[MS-EVEN6-Diff]:

EventLog Remoting Protocol Version 6.0

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation ("this documentation") for protocols, file formats, data portability, computer languages, and standards ~~as well as overviews of the interaction among each of these technologies~~support. Additionally, overview documents cover inter-protocol relationships and interactions.
- **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you ~~may~~can make copies of it in order to develop implementations of the technologies that are described in ~~the Open Specifications~~ this documentation and ~~may~~can distribute portions of it in your implementations ~~using~~that use these technologies or in your documentation as necessary to properly document the implementation. You ~~may~~can also distribute in your implementation, with or without modification, any ~~schema, IDL's~~schemas, IDLs, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications~~.~~ documentation.
- **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.
- **Patents**. Microsoft has patents that ~~may~~might cover your implementations of the technologies described in the Open Specifications~~.~~ documentation. Neither this notice nor Microsoft's delivery of ~~the~~this documentation grants any licenses under those patents or any other Microsoft patents. However, a given Open ~~Specification may~~Specifications document might be covered by the Microsoft Open Specifications Promise or the Microsoft Community Promise. If you would prefer a written license, or if the technologies described in ~~the Open Specifications~~this documentation are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **Trademarks**. The names of companies and products contained in this documentation ~~may~~might be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names**. The example companies, organizations, products, domain names, ~~e-mail~~email addresses, logos, people, places, and events that are depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than as specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications ~~do~~documentation does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments, you are free to take advantage of them. Certain Open Specifications documents are intended for use in conjunction with publicly available ~~standard~~standards specifications and network programming art~~,~~ and ~~assumes~~, as such, assume that the reader either is familiar with the aforementioned material or has immediate access to it.

Revision Summary

Date	Revision History	Revision Class	Comments
10/22/2006	0.01	New	Version 0.01 release
1/19/2007	1.0	Major	Version 1.0 release
3/2/2007	1.1	Minor	Version 1.1 release
4/3/2007	1.2	Minor	Version 1.2 release
5/11/2007	1.3	Minor	Version 1.3 release
6/1/2007	2.0	Major	Updated and revised the technical content.
7/3/2007	2.0.1	Editorial	Changed language and formatting in the technical content.
7/20/2007	2.0.2	Editorial	Changed language and formatting in the technical content.
8/10/2007	2.1	Minor	Clarified the meaning of the technical content.
9/28/2007	2.2	Minor	Clarified the meaning of the technical content.
10/23/2007	3.0	Major	Added clarification of server state.
11/30/2007	4.0	Major	Updated and revised the technical content.
1/25/2008	5.0	Major	Updated and revised the technical content.
3/14/2008	6.0	Major	Updated and revised the technical content.
5/16/2008	6.0.1	Editorial	Changed language and formatting in the technical content.
6/20/2008	6.1	Minor	Clarified the meaning of the technical content.
7/25/2008	6.2	Minor	Clarified the meaning of the technical content.
8/29/2008	6.3	Minor	Removed constants in IDL.
10/24/2008	6.3.1	Editorial	Changed language and formatting in the technical content.
12/5/2008	7.0	Major	Updated and revised the technical content.
1/16/2009	7.0.1	Editorial	Changed language and formatting in the technical content.
2/27/2009	7.0.2	Editorial	Changed language and formatting in the technical content.
4/10/2009	7.0.3	Editorial	Changed language and formatting in the technical content.
5/22/2009	7.1	Minor	Clarified the meaning of the technical content.
7/2/2009	7.1.1	Editorial	Changed language and formatting in the technical content.
8/14/2009	7.2	Minor	Clarified the meaning of the technical content.
9/25/2009	7.3	Minor	Clarified the meaning of the technical content.
11/6/2009	8.0	Major	Updated and revised the technical content.
12/18/2009	9.0	Major	Updated and revised the technical content.
1/29/2010	9.1	Minor	Clarified the meaning of the technical content.

Date	Revision History	Revision Class	Comments
3/12/2010	9.2	Minor	Clarified the meaning of the technical content.
4/23/2010	9.3	Minor	Clarified the meaning of the technical content.
6/4/2010	9.4	Minor	Clarified the meaning of the technical content.
7/16/2010	10.0	Major	Updated and revised the technical content.
8/27/2010	10.1	Minor	Clarified the meaning of the technical content.
10/8/2010	10.1	None	No changes to the meaning, language, or formatting of the technical content.
11/19/2010	11.0	Major	Updated and revised the technical content.
1/7/2011	12.0	Major	Updated and revised the technical content.
2/11/2011	13.0	Major	Updated and revised the technical content.
3/25/2011	14.0	Major	Updated and revised the technical content.
5/6/2011	15.0	Major	Updated and revised the technical content.
6/17/2011	15.1	Minor	Clarified the meaning of the technical content.
9/23/2011	15.1	None	No changes to the meaning, language, or formatting of the technical content.
12/16/2011	16.0	Major	Updated and revised the technical content.
3/30/2012	16.0	None	No changes to the meaning, language, or formatting of the technical content.
7/12/2012	16.0	None	No changes to the meaning, language, or formatting of the technical content.
10/25/2012	16.0	None	No changes to the meaning, language, or formatting of the technical content.
1/31/2013	16.0	None	No changes to the meaning, language, or formatting of the technical content.
8/8/2013	17.0	Major	Updated and revised the technical content.
11/14/2013	18.0	Major	Updated and revised the technical content.
2/13/2014	18.0	None	No changes to the meaning, language, or formatting of the technical content.
5/15/2014	18.0	None	No changes to the meaning, language, or formatting of the technical content.
6/30/2015	19.0	Major	Significantly changed the technical content.
10/16/2015	19.0	~~No Change~~None	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

1 Introduction

The EventLog Remoting Protocol Version 6.0, originally available in the Windows Vista operating system, is a **remote procedure call (RPC)**–based protocol that exposes RPC methods for reading **events** in both **live event logs** and **backup event logs** on remote computers. This protocol also specifies how to get general information for a log, such as number of **records** in the log, oldest records in the log, and if the log is full. It may also be used for clearing and backing up both types of **event logs**.

Sections 1.5, 1.8, 1.9, 2, and 3 of this specification are normative ~~and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in [RFC2119]. Sections 1.5 and 1.9 are also normative but do not contain those terms~~. All other sections and examples in this specification are informative.

1.1 Glossary

~~The~~This document uses the following terms ~~are specific to this document~~:

backup event log: An **event log** that cannot be written to, only read from. **Backup event logs** are typically used for archival purposes, or for copying to another computer for use by support personnel.

channel: A destination of **event** writes and a source for **event** reads. The physical backing store is a **live event log**.

cursor: The current position within a **result set**.

endpoint: A network-specific address of a remote procedure call (RPC) server process for remote procedure calls. The actual name and type of the endpoint depends on the **RPC** protocol sequence that is being used. For example, for RPC over TCP (RPC Protocol Sequence ncacn_ip_tcp), an endpoint might be TCP port 1025. For RPC over Server Message Block (RPC Protocol Sequence ncacn_np), an endpoint might be the name of a named pipe. For more information, see [C706].

event: A discrete unit of historical data that an application exposes that may be relevant to other applications. An example of an event would be a particular user logging on to the computer.

event descriptor: A structure indicating the kind of **event**. For example, a user logging on to the computer could be one kind of **event**, while a user logging off would be another, and these **events** could be indicated by using distinct **event descriptors**.

event log: A collection of records, each of which corresponds to an event.

event metadata: The metadata of an **event** provider including the **event** definition, **events**, **channels** the provider generates the events into, the unique identifier of the provider, and the localized string tables for this provider.

globally unique identifier (GUID): A term used interchangeably with **universally unique identifier (UUID)** in Microsoft protocol technical documents (TDs). Interchanging the usage of these terms does not imply or require a specific algorithm or mechanism to generate the value. Specifically, the use of this term does not imply or require that the algorithms described in [RFC4122] or [C706] must be used for generating the **GUID**. See also **universally unique identifier (UUID)**.

Interface Definition Language (IDL): The International Standards Organization (ISO) standard language for specifying the interface for remote procedure calls. For more information, see [C706] section 4.

live event log: An **event log** that can be written to and read from.

opnum: An operation number or numeric identifier that is used to identify a specific **remote procedure call (RPC)** method or a method in an interface. For more information, see [C706] section 12.5.2.12 or [MS-RPCE].

publisher: In the context of events: The source of event generation. An application or component that writes to one or more **event logs**. An application that publishes events.

publisher metadata: The metadata of an **event** that includes the predefined property values of one **event** and the **event** user-defined data definition.

query: A context-dependent term commonly overloaded with three meanings, defined as follows: The act of requesting **records** from a set of **records** or the request itself. The particular string defining the criteria for which **records** are to be returned. This string can either be an XPath, as specified in [XPATH], (for more information, see [MS-EVEN6] section 2.2.15) or a **structured XML query**, as specified in [XML10], (for more information, see [MS-EVEN6] section 2.2.16).

record: The data structure that contains an **event** that is currently represented in an **event log**.

remote procedure call (RPC): A context-dependent term commonly overloaded with three meanings. Note that much of the industry literature concerning RPC technologies uses this term interchangeably for any of the three meanings. Following are the three definitions: (*) The runtime environment providing remote procedure call facilities. The preferred usage for this meaning is "RPC runtime". (*) The pattern of request and response message exchange between two parties (typically, a client and a server). The preferred usage for this meaning is "RPC exchange". (*) A single message from an exchange as defined in the previous definition. The preferred usage for this term is "RPC message". For more information about RPC, see [C706].

result set: A set of **records** that are selected by a **query**.

RPC dynamic endpoint: A network-specific server address that is requested and assigned at run time, as described in [C706].

RPC endpoint: A network-specific address of a server process for remote procedure calls (RPCs). The actual name of the RPC endpoint depends on the RPC protocol sequence being used. For example, for the NCACN_IP_TCP RPC protocol sequence an RPC endpoint might be TCP port 1025. For more information, see [C706].

RPC protocol sequence: A character string that represents a valid combination of a **remote procedure call (RPC)** protocol, a network layer protocol, and a transport layer protocol, as described in [C706] and [MS-RPCE].

structured XML query: An XML document that specifies a **query** that ~~may~~can contain multiple **subqueries**. For more information, see section 2.2.16.

subquery: A component of a **structured XML query**. For more information, see section 2.2.16.

subscription filter: An XPath query expression used in a subscription to filter out events that do not meet certain criteria from the client.

universally unique identifier (UUID): A 128-bit value. UUIDs can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects in cross-process communication such as client and server interfaces, manager entry-point vectors, and **RPC** objects. UUIDs are highly likely to be unique. UUIDs are also known as **globally unique identifiers (GUIDs)** and these terms are used interchangeably in the Microsoft protocol technical documents (TDs). Interchanging the usage of these terms does not imply or require a specific algorithm or mechanism to generate the UUID. Specifically, the use of this term does not imply or require that the algorithms described in [RFC4122] or [C706] must be used for generating the UUID.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as defined in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, https://www2.opengroup.org/ogsys/catalog/c706

[ISO/IEC-8859-1] International Organization for Standardization, "Information Technology -- 8-Bit Single-Byte Coded Graphic Character Sets -- Part 1: Latin Alphabet No. 1", ISO/IEC 8859-1, 1998, http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=28245

Note There is a charge to download the specification.

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[MS-ERREF] Microsoft Corporation, "Windows Error Codes".

[MS-EVEN] Microsoft Corporation, "EventLog Remoting Protocol".

[MS-GPSI] Microsoft Corporation, "Group Policy: Software Installation Protocol Extension".

[MS-KILE] Microsoft Corporation, "Kerberos Protocol Extensions".

[MS-LSAD] Microsoft Corporation, "Local Security Authority (Domain Policy) Remote Protocol".

[MS-NLMP] Microsoft Corporation, "NT LAN Manager (NTLM) Authentication Protocol".

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions".

[MS-SPNG] Microsoft Corporation, "Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) Extension".

[PRA-CreateDirectory] Microsoft Corporation, "CreateDirectory function", ~~http://download~~https://msdn.microsoft.com/~~download/5/B/C/5BC37A4E-6304-45AB-8C2D-AE712526E7F7/CreateDirectory.pdf~~en-us/library/windows/desktop/aa363855(v=vs.85).aspx

[PRA-CreateFile] Microsoft Corporation, "CreateFile function", ~~http://download~~https://msdn.microsoft.com/~~download/5/B/C/5BC37A4E-6304-45AB-8C2D-AE712526E7F7/CreateFile.pdf~~en-us/library/windows/desktop/aa363858(v=vs.85).aspx

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC3986] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[RFC4122] Leach, P., Mealling, M., and Salz, R., "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005, http://www.ietf.org/rfc/rfc4122.txt

[RFC4234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005, http://www.rfc-editor.org/rfc/rfc4234.txt

[UNICODE] The Unicode Consortium, "The Unicode Consortium Home Page", 2006, http://www.unicode.org/

[XML10] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Third Edition)", February 2004, http://www.w3.org/TR/2004/REC-xml-20040204/

[XMLSCHEMA1.1/2:2008] Peterson, D., Biron, P.V., Malhotra, A., et al., Eds., "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes", W3C Working Draft, June 2008, http://www.w3.org/TR/2008/WD-xmlschema11-2-20080620/

[XPATH] Clark, J., and DeRose, S., "XML Path Language (XPath), Version 1.0", W3C Recommendation, November 1999, http://www.w3.org/TR/xpath/

1.2.2 Informative References

[MSDN-BNDHNDLS] Microsoft Corporation, "Binding Handles", http://msdn.microsoft.com/en-us/library/aa373566.aspx

[MSDN-CH] Microsoft Corporation, "Context Handles", http://msdn.microsoft.com/en-us/library/aa373605(VS.85).aspx

[MSDN-CONSUMEVTS] Microsoft Corporation, "Consuming Events", http://msdn.microsoft.com/en-us/library/dd996910.aspx

[MSDN-CreateFile] Microsoft Corporation, "CreateFile function", http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

[MSDN-EVENTRECORD] Microsoft Corporation, "EVENT_RECORD structure", http://msdn.microsoft.com/en-us/library/aa363769(VS.85).aspx

[MSDN-EVENTS] Microsoft Corporation, "Event Schema", http://msdn.microsoft.com/en-us/library/aa385201.aspx

[MSDN-EVENT] Microsoft Corporation, "Event Logging", http://msdn.microsoft.com/en-us/library/aa363652.aspx

[MSDN-EVENT_DESCRIPTOR] Microsoft Corporation, "EVENT_DESCRIPTOR structure", http://msdn.microsoft.com/en-us/library/aa363754(VS.85).aspx

[MSDN-EVENT_HEADER] Microsoft Corporation, "EVENT_HEADER structure", http://msdn.microsoft.com/en-us/library/aa363759(v=VS.85).aspx

[MSDN-EvntManifestSE] Microsoft Corporation, "EventManifest Schema Elements", http://msdn.microsoft.com/en-us/library/aa382753(v=VS.85).aspx

[MSDN-EVTLGCHWINEVTLG] Microsoft Corporation, "Event Logs and Channels in Windows Event Log", http://msdn.microsoft.com/en-us/library/aa385225.aspx

[MSDN-EVTSCT] Microsoft Corporation, "Event Schema Complex Types", http://msdn.microsoft.com/en-us/library/aa384343(v=VS.85).aspx

[MSDN-EVTSST] Microsoft Corporation, "Event Schema Simple Types", http://msdn.microsoft.com/en-us/library/aa385204(v=VS.85).aspx

[MSDN-FILEATT] Microsoft Corporation, "GetFileAttributes function", http://msdn.microsoft.com/en-us/library/aa364944.aspx

[MSDN-FMT] Microsoft Corporation, "FormatMessage function", http://msdn.microsoft.com/en-us/library/ms679351.aspx

[MSDN-GETTHDPREUILANG] Microsoft Corporation, "GetThreadPreferredUILanguages function", http://msdn.microsoft.com/en-us/library/dd318128(v=vs.85).aspx

[MSDN-MUIResrcMgmt] Microsoft Corporation, "MUI Resource Management", http://msdn.microsoft.com/en-us/library/dd319070(VS.85).aspx

[MSDN-ProcessTrace] Microsoft Corporation, "ProcessTrace function", http://msdn.microsoft.com/en-us/library/aa364093.aspx

[MSDN-ProvEvts] Microsoft Corporation, "Providing Events", http://msdn.microsoft.com/en-us/library/aa364098(v=VS.85).aspx

[MSDN-RpcAsyncCompleteCall] Microsoft Corporation, "RpcAsyncCompleteCall function", http://msdn.microsoft.com/en-us/library/aa375572(v=VS.85).aspx

[MSDN-stringTable] Microsoft Corporation, "stringTable (LocalizationType) Element", http://msdn.microsoft.com/en-us/library/aa384125(v=VS.85).aspx

[MSDN-WAIM] Microsoft Corporation, "Writing an Instrumentation Manifest", http://msdn.microsoft.com/en-us/library/dd996930.aspx

[MSDN-WPPST] Microsoft Corporation, "WPP Software Tracing", http://msdn.microsoft.com/en-us/library/ff556204.aspx

[MSKB-113996] Microsoft Corporation, "INFO: Mapping NT Status Error Codes to Win32 Error Codes", March 2005, http://support.microsoft.com/kb/113996

[PE-COFF] Microsoft Corporation, "Microsoft Portable Executable and Common Object File Format Specification", May2006, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

1.3 Overview

1.3.1 Background

Event logs allow applications or the operating system to store historical information that is of interest to administrators. The information is organized in separate, discrete pieces of information, which are referred to as events. An example of an event is a user logging on to the computer.

The events represented in an event log are referred to as records. The records in a log are numbered. The first event written has its record number (that is, a field in the record) set to 1, the second event has its record number set to 2, and so on. Logs can be configured to be circular. A circular log is one in which the oldest records are overwritten once the logs reach some maximum size. Once a record is written, it is never again updated and is thereafter treated as read-only.

A computer can have several event logs. One log might be devoted to security events while another can be for general application use.

Applications or components that write to event logs are known as **publishers**. A single event log might contain events from many publishers. A single publisher can write to multiple logs. Publishers play the role played by event sources in the EventLog Remoting Protocol [MS-EVEN].

Publishers write several kinds of events. For example, a user logging on to the computer could be one kind of event while a user logging off would be another. When a publisher writes an event, it specifies an **event descriptor**, which indicates what kind of event is being written. Event descriptors (section 1.8.3) subsume the **eventID** and **event category** fields used in the EventLog Remoting Protocol. Publishers also specify message files that are used to define localized messages that can be used to display events using localized strings.

An event log can be either a live event log or a backup event log. A live event log is one that can be used for both reading and writing. A live event log can be used to create a backup event log, which is a read-only snapshot of a live event log. Backup event logs are typically used for archival purposes or are copied to another computer for use by support personnel.

Each live event log corresponds to a **channel**. A channel is a logical data stream of event records. Publishers write to channels, and each channel has a live event log as its physical backing store. Events can be read from either a backup event log or a channel corresponding to a live event log. A backup event log cannot be associated with a channel.

1.3.2 EventLog Remoting Protocol Version 6.0

The EventLog Remoting Protocol Version 6.0 provides a way to access event logs on remote computers.

For both live logs and backup logs, the protocol exposes RPC (as specified in [MS-RPCE]) methods for reading event and for getting basic information about the log, such as the number of records in the log, the oldest record in the log, and whether the log is full, and therefore can no longer accept additional events. When reading events, a filter can be specified so that only desired records are returned.

The EventLog Remoting Protocol Version 6.0 does not support writing events to either live event logs or backup event logs.

For live event logs only, the protocol also exposes RPC methods for subscriptions, clearing logs, and creating backup logs. Subscriptions are similar to normal reading except the subscription can be used to get events asynchronously as they arrive.

The protocol provides the methods for reading publisher and event logs settings and it also provides the methods to change the settings of event logs. Additionally, the protocol provides methods for converting events into localized messages suitable for display to users.

A **query** can be done in which a filter is applied. The **result set** is the set of records that satisfy the filter. The **cursor** is the location in the result set that is the last record retrieved by the caller. A filter is composed by using selectors and suppressors. A selector specifies records to include, while a suppressor specifies records to exclude. Suppressors override selectors.

For more information and an overview of methods used, see section 3.1.4.

1.4 Relationship to Other Protocols

The EventLog Remoting Protocol Version 6.0 is dependent on RPC (as specified in [MS-RPCE]) for message transport.

The EventLog Remoting Protocol Version 6.0 is a replacement for the EventLog Remoting Protocol [MS-EVEN]. The EventLog Remoting Protocol Version 6.0 supports a number of new features not present in the original EventLog Remoting Protocol, such as query processing with filters, subscriptions, localized message support, and configuration support.

The EventLog Remoting Protocol Version 6.0 allows access to all the event logs accessible by the EventLog Remoting Protocol, plus some additional event logs not accessible via the EventLog Remoting Protocol.

The server-side dependency on the Local Security Authority (Domain Policy) Remote Protocol [MS-LSAD] is a shared-state dependency resulting from EventLog Remoting Protocol Version 6.0 depending on the Access Check algorithm pseudocode (as specified in Windows Data Types [MS-DTYP] section 2.5.3.2), which in turn depends on state in the Local Security Authority (Domain Policy) Remote Protocol.

1.5 Prerequisites/Preconditions

The EventLog Remoting Protocol Version 6.0 has the prerequisites, as specified in [MS-RPCE], that are common to protocols depending on RPC.

1.6 Applicability Statement

The EventLog Remoting Protocol Version 6.0 is well-suited for reading event logs.Event logs can be used for many purposes; for example, recording local security events and application start/stop events.

An eventlog user can retrieve events from an eventlog server, but an eventlog server cannot retrieve events from a remote publisher's eventlog server.

The EventLog Remoting Protocol Version 6.0 is typically preferred over the original EventLog Remoting Protocol whenever both parties support it because it offers numerous improvements, such as subscriptions and improved configurability, as specified in section 3.1.4.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- Protocol Version: A client wanting to use the EventLog Remoting Protocol Version 6.0 can attempt to connect to the **UUID** for the EventLog Remoting Protocol Version 6.0. If this UUID does not exist, the EventLog Remoting Protocol [MS-EVEN] UUID can still exist, and the client can attempt to connect to it.

  The EventLog Remoting Protocol Version 6.0 RPC interface has a single version number. The version number can change, but this version of the protocol requires it to be a specific value (for more information, see section 2.1.1). The EventLog Remoting Protocol Version 6.0 can be extended by adding RPC messages to the interface with **opnums** lying numerically beyond those defined here. An RPC client determines whether such methods are supported by attempting to invoke the method; if the method is not supported, the RPC run time returns an "opnum out of range" error, as specified in [C706] and [MS-RPCE]. Details on RPC versioning and capacity negotiation in this situation are specified in [C706] section 6.3 and [MS-RPCE] section 1.7.

- Security and Authentication Methods: RPC servers and clients in the EventLog Remoting Protocol Version 6.0 use an RPC authentication service, as specified in section 2.1.

- Localization: The EventLog Remoting Protocol Version 6.0 defines several methods that support localization. These methods each take a locale identifier (ID) (as specified in [MS-GPSI] Appendix A), which is used to determine the language preferences for localization.

1.8 Vendor-Extensible Fields

1.8.1 Channel Names

Each channel has a name that is a [UNICODE] string. This name MUST be unique across all channels on the same server. The set of channel names also includes all names of live event logs, as specified

in the original EventLog Remoting Protocol. Event logs are specified in section 3.1.1.2. Event log naming constraints are specified in section 1.8.2.

Channel names are treated in a case-insensitive manner, are limited to 255 characters, and cannot begin with the character \ (backslash). No restrictions other than these exist on the characters that are included in a channel name. However, channel names SHOULD<1> be prefixed with a unique value (such as the name of the entity that created the channel) so that the channels are easily identifiable and readable.

1.8.2 Publisher Names

Each publisher has a name that is a [UNICODE] string. This name MUST be unique across all publishers on the same server. Publisher names MUST be treated in a case-insensitive manner, MUST be limited to 255 characters, and MUST NOT begin with the backslash \. The set of publisher names also includes all event sources (for more information, see [MSDN-EVENTS]). Apart from these restrictions, there are no character restrictions on publisher names. However, publisher names SHOULD<2> be prefixed with a unique value (such as the name of the entity that created the publisher) so that the publishers are easily identifiable and readable.

1.8.3 Event Descriptor

Each publisher uses event descriptors to identify the different types of events that it writes. Publishers do not need to be concerned with using the same event descriptors as other publishers do, because the meaning of a particular event descriptor's value is determined on a per-publisher basis.

1.8.4 Error Codes

The EventLog Remoting Protocol Version 6.0 uses Win32 error codes, specifically, the subset designated as "NTSTATUS". These values are taken from the Windows error number space, as specified in [MS-ERREF] section 2.3. Vendors SHOULD reuse those values with their indicated meanings.<3> Choosing any other value runs the risk of a collision in the future. For a mapping of Windows NT operating system status error codes to Win32 error codes, see [MSKB-113996].

1.9 Standards Assignments

The EventLog Remoting Protocol Version 6.0 has no standards assignments, only private assignments made by Microsoft by using allocation procedures, as specified in other protocols.

Microsoft has allocated to the EventLog Remoting Protocol Version 6.0 an interface **GUID** by using the procedure specified in [C706] section 6.2.2. It also allocates an RPC **endpoint** name, as specified in [C706]. The assignments are as follows.

Parameter	Value
RPC Interface UUID	{F6BEAFF7-1E19-4FBB-9F8F-B89E2018337C}
RPC Endpoint Name	Eventlog

2 Messages

2.1 Transport

This protocol uses RPC as the transport protocol.

2.1.1 Server

The server interface is identified by UUID F6BEAFF7-1E19-4FBB-9F8F-B89E2018337C version 1.0, using the **RPC dynamic endpoint** EventLog. The server MUST specify RPC over TCP/IP (that is, ncacn_ip_tcp) as the **RPC protocol sequence** to the RPC implementation, as specified in [MS-RPCE]. The server MUST specify both the Simple and Protected GSS-API Negotiation Mechanism [MS-SPNG] (0x9) and Kerberos [MS-KILE] (0x10) as the RPC authentication service, as specified in [MS-RPCE].

The EventLog Remoting Protocol Version 6.0 allows any user to establish a connection to the RPC server. The server uses the underlying RPC protocol to retrieve the identity of the caller that made the method call, as specified in the second bullet of section 3.3.3.4.3 of [MS-RPCE]. The server SHOULD use this identity to perform method-specific access checks, as specified in section 3.1.4.

2.1.2 Client

The client MUST use RPC over TCP/IP (that is, ncacn_ip_tcp), as specified in [MS-RPCE], as the RPC protocol sequence to communicate with the server. The higher-level protocol or client application MUST specify the Simple and Protected GSS-API Negotiation Mechanism [MS-SPNG] (0x9), NTLM [MS-NLMP] (0xA), or Kerberos [MS-KILE] (0x10) as the RPC authentication service, as specified in [MS-RPCE], and the protocol client MUST pass this choice unmodified to the RPC layer.

2.2 Common Data Types

In addition to RPC base types, the following sections use the definitions of FILETIME, DWORD, and GUID, as specified in [MS-DTYP] Appendix A.

2.2.1 RpcInfo

The RpcInfo structure is used for certain methods that return additional information about errors.

```
typedef struct tag_RpcInfo {
  DWORDm_error,
  m_subErr,
  m_subErrParam;
} RpcInfo;
```

m_error: A Win32 error code that contains a general operation success or failure status. A value of 0x00000000 indicates success; any other value indicates failure. Unless noted otherwise, all failure values MUST be treated equally.

m_subErr: MUST be zero unless specified otherwise in the method using this structure. Unless noted otherwise, all nonzero values MUST be treated equally.

m_subErrParam: MUST be zero unless specified otherwise in the method using this structure. Unless noted otherwise, all nonzero values MUST be treated equally.

2.2.2 BooleanArray

The BooleanArray structure is defined as follows.

```
typedef struct _BooleanArray {
  [range(0, MAX_RPC_BOOL_ARRAY_COUNT)]
    DWORD count;
  [size_is(count)] boolean* ptr;
} BooleanArray;
```

count: A 32-bit unsigned integer that contains the number of BOOLEAN values pointed to by **ptr**.

ptr: A pointer to an array of BOOLEAN values.

2.2.3 UInt32Array

The UInt32Array structure is defined as follows.

```
typedef struct _UInt32Array {
  [range(0, MAX_RPC_UINT32_ARRAY_COUNT)]
    DWORD count;
  [size_is(count)] DWORD* ptr;
} UInt32Array;
```

count: An unsigned 32-bit integer that contains the number of unsigned 32-bit integers pointed to by ptr.

ptr: A pointer to an array of unsigned 32-bit integers.

2.2.4 UInt64Array

The UInt64Array structure is defined as follows.

```
typedef struct _UInt64Array {
  [range(0, MAX_RPC_UINT64_ARRAY_COUNT)]
    DWORD count;
  [size_is(count)] DWORD64* ptr;
} UInt64Array;
```

count: A 32-bit unsigned integer that contains the number of 64-bit integers pointed to by ptr.

ptr: A pointer to an array of unsigned 64-bit integers.

2.2.5 StringArray

The StringArray structure is defined as follows.

```
typedef struct _StringArray {
  [range(0, MAX_RPC_STRING_ARRAY_COUNT)]
    DWORD count;
  [size_is(count), string] LPWSTR* ptr;
} StringArray;
```

count: A 32-bit unsigned integer that contains the number of strings pointed to by ptr.

ptr: A pointer to an array of null-terminated Unicode (as specified in [UNICODE]) strings.

2.2.6 GuidArray

The GuidArray structure is defined as follows.

```
typedef struct _GuidArray {
  [range(0, MAX_RPC_GUID_ARRAY_COUNT)]
    DWORD count;
  [size_is(count)] GUID* ptr;
} GuidArray;
```

count: A 32-bit unsigned integer that contains the number of GUIDs pointed to by ptr.

ptr: A pointer to an array of GUIDs.

2.2.7 EvtRpcVariant

The EvtRpcVariant structure is defined as follows.

```
typedef struct tag_EvtRpcVariant {
  EvtRpcVariantType type;
  DWORD flags;
  [switch_is(type)] union {
    [case(EvtRpcVarTypeNull)]
      int nullVal;
    [case(EvtRpcVarTypeBoolean)]
      boolean booleanVal;
    [case(EvtRpcVarTypeUInt32)]
      DWORD uint32Val;
    [case(EvtRpcVarTypeUInt64)]
      DWORD64 uint64Val;
    [case(EvtRpcVarTypeString)]
      [string] LPWSTR stringVal;
    [case(EvtRpcVarTypeGuid)]
      GUID* guidVal;
    [case(EvtRpcVarTypeBooleanArray)]
      BooleanArray booleanArray;
    [case(EvtRpcVarTypeUInt32Array)]
      UInt32Array uint32Array;
    [case(EvtRpcVarTypeUInt64Array)]
      UInt64Array uint64Array;
    [case(EvtRpcVarTypeStringArray)]
      StringArray stringArray;
    [case(EvtRpcVarTypeGuidArray)]
      GuidArray guidArray;
  };
} EvtRpcVariant;
```

type: Indicates the actual type of the union.

flags: This flag MUST be set to either 0x0000 or 0x0001. If this flag is set to 0x0001, it indicates that an EvtRpcVariant structure has been changed by the client. For an example of how this flag might be set, suppose the client application retrieved an EvtRpcVariantList structure by calling EvtRpcGetChannelConfig, changed one or more EvtRpcVariant structures in the list, and then sent the list back to the server via EvtRpcPutChannelConfig. In this example, the server updates the values corresponding to the EvtRpcVariant structures with this flag set.

Value	Meaning
0x0000	A flag indicating that no instance of an EvtRpcVariant structure was changed by the client.
0x0001	A flag indicating that an EvtRpcVariant structure was changed by the client.

RpcVariant: The data type to be passed.

nullVal: MUST be set to 0x00000000.

booleanVal: A BOOLEAN value.

uint32Val: A 32-bit unsigned integer.

uint64Val: A 64-bit unsigned integer.

stringVal: A null-terminated UNICODE string.

guidVal: A GUID.

booleanArray: An array of BOOLEAN values that are stored as a BooleanArray.

uint32Array: An array of 32-bit unsigned integers that are stored as a UInt32Array.

uint64Array: An array of 64-bit unsigned integers that are stored as a UInt64Array.

stringArray: An array of strings that are stored as a StringArray.

guidArray: An array of GUIDs that are stored as a GuidArray.

2.2.8 EvtRpcVariantType

The EvtRpcVariantType enumeration is used by the EvtRpcVariant (section 2.2.7) type.

```
typedef [v1_enum] enum tag_EvtRpcVariantType
{
  EvtRpcVarTypeNull = 0,
  EvtRpcVarTypeBoolean,
  EvtRpcVarTypeUInt32,
  EvtRpcVarTypeUInt64,
  EvtRpcVarTypeString,
  EvtRpcVarTypeGuid,
  EvtRpcVarTypeBooleanArray,
  EvtRpcVarTypeUInt32Array,
  EvtRpcVarTypeUInt64Array,
  EvtRpcVarTypeStringArray,
  EvtRpcVarTypeGuidArray
} EvtRpcVariantType;
```

2.2.9 EvtRpcVariantList

The EvtRpcVariantList data type is a wrapper for multiple EvtRpcVariant (section 2.2.7) data types.

```
typedef struct tag_EvtRpcVariantList {
  [range(0, MAX_RPC_VARIANT_LIST_COUNT)]
    DWORD count;
  [size_is(count)] EvtRpcVariant* props;
} EvtRpcVariantList;
```

count: Number of EvtRpcVariant values pointed to by the **props** field.

props: Pointer to an array of EvtRpcVariant values.

2.2.10 EvtRpcAssertConfigFlags Enumeration

The EvtRpcAssertConfigFlags Enumeration members specify how the *path* and *channelPath* parameters (used by a number of the methods in 3.1.4) are to be interpreted.

```
typedef [v1_enum] enum tag_EvtRpcAssertConfigFlags
{
  EvtRpcChannelPath = 0,
  EvtRpcPublisherName = 1
} EvtRpcAssertConfigFlags;
```

EvtRpcChannelPath: The associated parameter string contains a path to a channel.

EvtRpcPublisherName: The associated parameter string contains a publisher name.

2.2.11 EvtRpcQueryChannelInfo

The format of the EvtRpcQueryChannelInfo data type is as follows.

```
typedef struct tag_EvtRpcQueryChannelInfo {
  LPWSTR name;
  DWORD status;
} EvtRpcQueryChannelInfo;
```

name: Name of the channel to which the status applies.

status: A Win32 error code that indicates the channel status. A value of 0x00000000 indicates success; any other value indicates failure. Unless otherwise noted, all failure values MUST be treated equally.

2.2.12 BinXml

BinXml is a token representation of text XML 1.0, which is specified in [XML10]. Here, BinXml encodes an XML document so that the original XML text can be correctly reproduced from the encoding. For information about the encoding algorithm, see section 3.1.4.7.

The binary format for all numeric values is always little-endian. No alignment is required for any data. The format is given in the following Augmented Backus-Naur Form (ABNF) example, as specified in [RFC4234]).

In addition to defining the layout of the binary XML binary large objects (BLOBs), the following ABNF example has additional annotations that suggest a way to convert the binary to text. To convert to text, a tool is needed to evaluate the BinXml according to ABNF and to emit text for certain key rules. That text is emitted before evaluating the rule. The actual text to emit is defined in the sections as noted.

When processing the Attribute rule, the text generated is as specified in section 2.2.12.2.

Note When the emit rules specify emitting a literal string, that string is surrounded by quotes. The quotation marks shown are not part of the output. They are included in the text to delineate the characters that are sent on the wire. For example, an instruction might specify that "/>" is output.

```
; ==== Top-level Definitions ======================================
;
Document = 0*1Prolog Fragment 0*1Misc EOFToken
Prolog = PI
Misc = PI
```

```
Fragment = 0*FragmentHeader ( Element / TemplateInstance )
FragmentHeader = FragmentHeaderToken MajorVersion MinorVersion Flags
MajorVersion = OCTET
MinorVersion = OCTET
Flags = OCTET

;
; ==== Basic XML Definitions ====================================
;
Element =
 ( StartElement CloseStartElementToken Content EndElementToken ) /
 ( StartElement CloseEmptyElementToken ) ; Emit using Element Rule
Content =
 0*(Element / CharData / CharRef / EntityRef / CDATASection / PI)
CharData = ValueText / Substitution
StartElement =
 OpenStartElementToken 0*1DependencyId ElementByteLength
 Name 0*1AttributeList
DependencyId = WORD
ElementByteLength = DWORD
AttributeList = AttributeListByteLength 1*Attribute
Attribute =
 AttributeToken Name AttributeCharData ; Emit using Attribute Rule
AttributeCharData =
 0*(ValueText / Substitution / CharRef / EntityRef)
AttributeListByteLength = DWORD
ValueText = ValueTextToken StringType LengthPrefixedUnicodeString
Substitution =
 NormalSubstitution / OptionalSubstitution
; Emit using Substitution Rule
NormalSubstitution =
 NormalSubstitutionToken SubstitutionId ValueType
OptionalSubstitution =
 OptionalSubstitutionToken SubstitutionId ValueType
SubstitutionId = WORD
CharRef = CharRefToken WORD ; Emit using CharRef Rule
EntityRef = EntityRefToken Name ; Emit using EntityRef Rule
CDATASection = CDATASectionToken LengthPrefixedUnicodeString
; Emit using CDATA Section Rule
PI = PITarget PIData
PITarget = PITargetToken Name ; Emit using PITarget Rule
PIData = PIDataToken LengthPrefixedUnicodeString
; Emit using PIData Rule
Name = NameHash NameNumChars NullTerminatedUnicodeString
NameHash = WORD
NameNumChars = WORD

;
; ==== Token Types ==============================================
;
EOFToken = %x00
OpenStartElementToken = %x01 / %x41
CloseStartElementToken = %x02 ;Emit using CloseStartElementToken Rule
CloseEmptyElementToken = %x03 ;Emit using CloseEmptyElementToken Rule
EndElementToken = %x04 ; Emit using EndElementToken Rule

ValueTextToken = %x05 / %x45
AttributeToken = %x06 / %x46
CDATASectionToken = %x07 / %x47
CharRefToken = %x08 / %x48
EntityRefToken = %x09 / %x49

PITargetToken = %x0A
PIDataToken = %x0B
TemplateInstanceToken = %x0C
NormalSubstitutionToken = %x0D
OptionalSubstitutionToken = %x0E
FragmentHeaderToken = %x0F

;
```

```
; ==== Template-related definitions ==============================
;
TemplateInstance =
 TemplateInstanceToken TemplateDef TemplateInstanceData
TemplateDef =
 %b0 TemplateId TemplateDefByteLength
 0*FragmentHeader Element EOFToken
TemplateId = GUID

;
; The full length of the value section of the TemplateInstanceData
; can be obtained by adding up all the lengths described in the
; value spec.
;
TemplateInstanceData =
 ValueSpec *Value; Emit using TemplateInstanceDataRule
ValueSpec = NumValues *ValueSpecEntry
NumValues = DWORD
ValueSpecEntry = ValueByteLength ValueType %x00
ValueByteLength = WORD

TemplateDefByteLength = DWORD

;
; ==== Value Types ===============================================
;
ValueType =
 NullType / StringType / AnsiStringType / Int8Type / UInt8Type /
 Int16Type / UInt16Type / Int32Type / UInt32Type / Int64Type /
 Int64Type / Real32Type / Real64Type / BoolType / BinaryType /
 GuidType / SizeTType / FileTimeType / SysTimeType / SidType /
 HexInt32Type / HexInt64Type / BinXmlType / StringArrayType /
 AnsiStringArrayType / Int8ArrayType / UInt8ArrayType /
 Int16ArrayType / UInt16ArrayType / Int32ArrayType / UInt32ArrayType/
 Int64ArrayType / UInt64ArrayType / Real32ArrayType /
 Real64ArrayType / BoolArrayType / GuidArrayType / SizeTArrayType /
 FileTimeArrayType / SysTimeArrayType / SidArrayType /
 HexInt32ArrayType / HexInt64ArrayType
NullType = %x00
StringType = %x01
AnsiStringType = %x02
Int8Type = %x03
UInt8Type = %x04
Int16Type = %x05
UInt16Type = %x06
Int32Type = %x07
UInt32Type = %x08
Int64Type = %x09
UInt64Type = %x0A
Real32Type = %x0B
Real64Type = %x0C
BoolType = %x0D
BinaryType = %x0E
GuidType = %x0F
SizeTType = %x10
FileTimeType = %x11
SysTimeType = %x12
SidType = %x13
HexInt32Type = %x14
HexInt64Type = %x15
BinXmlType = %x21
StringArrayType = %x81
AnsiStringArrayType = %x82
Int8ArrayType = %x83
UInt8ArrayType = %x84
Int16ArrayType = %x85
UInt16ArrayType = %x86
Int32ArrayType = %x87
UInt32ArrayType = %x88
Int64ArrayType = %x89
```

```
UInt64ArrayType = %x8A
Real32ArrayType = %x8B
Real64ArrayType = %x8C
BoolArrayType = %x8D
GuidArrayType = %x8F
SizeTArrayType = %x90
FileTimeArrayType = %x91
SysTimeArrayType = %x92
SidArrayType = %x93
HexInt32ArrayType = %x00 %x94
HexInt64ArrayType = %x00 %x95

;
; === Value Formats ================================================
;
Value =
 StringValue / AnsiStringValue / Int8Value / UInt8Value /
 Int16Value / UInt16Value / Int32Value / UInt32Value / Int64Value /
 UInt64Value / Real32Value / Real64Value / BoolValue / BinaryValue /
 GuidValue / SizeTValue / FileTimeValue / SysTimeValue / SidValue /
 HexInt32Value / HexInt64Value / BinXmlValue / StringArrayValue /
 AnsiStringArrayValue / Int8ArrayValue / UInt8ArrayValue /
 Int16ArrayValue / UInt16ArrayValue / Int32ArrayValue /
 UInt32ArrayValue / Int64ArrayValue / UInt64ArrayValue /
 Real32ArrayValue / Real64ArrayValue / BoolArrayValue /
 GuidArrayValue / SizeTArrayValue / FileTimeArrayValue /
 SysTimeArrayValue / SidArrayValue / HexInt32ArrayValue /
 HexInt64ArrayValue
StringValue = 0*WORD
AnsiStringValue = 0*OCTET
Int8Value = OCTET
UInt8Value = OCTET
Int16Value = 2*2OCTET
UInt16Value = 2*2OCTET
Int32Value = 4*4OCTET
UInt32Value = 4*4OCTET
Int64Value = 8*8OCTET
UInt64Value = 8*8OCTET
Real32Value = 4*4OCTET
Real64Value = 8*8OCTET
BoolValue = OCTET
BinaryValue = *OCTET
GuidValue = GUID
SizeTValue = UInt32Value / UInt64Value
FileTimeValue = 8*8OCTET
SysTimeValue = 16*16OCTET
SidValue = *OCTET
HexInt32Value = UInt32Value
HexInt64Value = UInt64Value
BinXmlValue = Fragment EOFToken

StringArrayValue = *NullTerminatedUnicodeString
AnsiStringArrayValue = *NullTerminatedAnsiString
Int8ArrayValue = *Int8Value
UInt8ArrayValue = *UInt8Value
Int16ArrayValue = *Int16Value
UInt16ArrayValue = *UInt16Value
Int32ArrayValue = *Int32Value
UInt32ArrayValue = *UInt32Value
Int64ArrayValue = *Int64Value
UInt64ArrayValue = *UInt64Value
Real32ArrayValue = *Real32Value
Real64ArrayValue = *Real64Value
BoolArrayValue = *BoolValue
GuidArrayValue = *GuidValue
SizeTArrayValue = *SizeTValue
FileTimeArrayValue = *FileTimeValue
SysTimeArrayValue = *SysTimeValue
SidArrayValue = *SidValue
HexInt32ArrayValue = *HexInt32Value
```

```
HexInt64ArrayValue = *HexInt64Value

;
; ==== Base Types ================================================
;
NullTerminatedUnicodeString = StringValue %x00 %x00
NullTerminatedAnsiString = AnsiStringValue %x00
LengthPrefixedUnicodeString = NumUnicodeChars StringValue
NumUnicodeChars = WORD
OCTET = %x0
WORD = 2*2OCTET
DWORD = 4*4OCTET
GUID = 16*16OCTET
```

Entity	Description
MajorVersion	The major version of BinXml. MUST be set to 1.
MinorVersion	The minor version of BinXml. MUST be set to 1.
Flags	The reserved value in the BinXml header. Not used currently and MUST be 0.
DependencyID	Specifies the index into the ValueSpec list of an instance of the TemplateDefinition (TemplateInstance). If the ValueType at that index is NullType, the element MUST NOT be included for rendering purposes. If the index is 0xFFFF, there is no dependency for the element.
ElementByteLength	The number of bytes that is after ElementByteLength and that makes up the entire element definition, including the EndElementToken or CloseEmptyElementToken for the element.
AttributeListByteLength	The number of bytes in the attribute list that is after AttributeListByteLength and is up to, but not including, the CloseStartElementToken or CloseEmptyElementToken; typically used for jumping to the end of the enclosing start element tag.
AttributeCharData	The character data that appears in an attribute value.
SubstitutionId	A 0-based positional identifier into the set of substitution values. Zero indicates the first substitution value; 1 indicates the second substitution value, and so on.
CharRef	An XML 1.0 character reference value.

NameHash	The low order 16 bits of the value that is generated by performing a hash of the binary representation of Name (in which NameNumChars * 2 is the hash input length).
	The hash function is implemented by initially setting the value of the hash to zero. For each character in Name, multiply the previous value of the hash by 65599 and add the binary representation of the character to the hash value.
	The following pseudocode shows how to implement this hash function.
	`hash(str)`
	`{`
	`  hashVal = 0;`
	`  for(i=0; i < strLen; i++ )`
	`  hashVal = hashVal*65599 + str[i];`
	`  return hashVal;`
	`}`

NameNumChars	The number of Unicode characters for the NameData, not including the null terminator.
OpenStartElementToken	A value of 0x01 indicates that the element start tag contains no elements; a value of 0x41 indicates that an attribute list can be expected in the element start tag.

Entity	Description
ValueTextToken	A value of 0x45 indicates that more data can be expected to follow in the current content of the element or attribute; a value of 0x05 indicates that no more such data follows.
AttributeToken	A value of 0x46 indicates that there is another attribute in the attribute list; a value of 0x06 indicates that no more attributes exist.
CDATASectionToken	A value of 0x47 indicates that more data can be expected to follow in the current content of the element or attribute; a value of 0x07 indicates that no more such data follows.
CharRefToken	A value of 0x48 indicates that more data can be expected to follow in the current content of the element or attribute; a value of 0x08 indicates that no more such data follows.
EntityRefToken	A value of 0x49 indicates that more data can be expected to follow in the current content of the element or attribute; a value of 0x09 indicates that no more such data follows.
TemplateId	The raw data of the GUID that identifies a template definition.
NumValues	The number of substitution values that make up the Template Instance Data.
ValueByteLength	The length, in bytes, of a substitution value as it appears in the Template Instance Data.
TemplateDefByteLength	The number of bytes after the TemplateDefByteLength up to and including the EOFToken (end of fragment or document) element for the template definition.
ValueType	The type of a substitution value, as it appears in the Template Instance Data.
Value	The raw data of the substitution value.
NumUnicodeChars	The number of wide characters in LengthPrefixedUnicodeString. The Length MUST include the null terminator if one is present in the string; however, length-prefixed strings are not required to have a null terminator.

2.2.12.1 Emitting Instruction for the Element Rule

Before emitting anything, the tool SHOULD determine whether there is an optional substitution that is NULL. If there is such a substitution, the tool MUST NOT emit anything for this element. The DependencyId rule (as specified in 2.2.12) determines whether there are any optional substitutions. If there are optional substitutions, the tool MUST emit the character "<" and the text, as specified by the Name rule (as specified in 2.2.12), as defined in the StartElement rule (also specified in 2.2.12). If the element contains array data (for more information, see section 3.1.4.7.5), the tool MUST emit multiple instances of the element, with one instance for each element of the array.

2.2.12.2 Emitting Instruction for the Attribute Rule

Before emitting anything, the tool SHOULD verify that the attribute data, as specified by the AttributeCharData rule in 2.2.12, is not empty. If the attribute data is empty, the tool SHOULD NOT emit anything. If the attribute data is not empty, emit the space character " " and the text, as specified by the Name rule in 2.2.12, the character "=", the character "'", the text, as specified by the AttributeCharData rule in 2.2.12, and, finally, the character "'".

2.2.12.3 Emitting Instruction for the Substitution Rule

BinXml uses templates, as specified in section 3.1.4.7.1. Substitutions are done only inside a template instance definition. Any data needed for substitutions is in the template instance data, which comes immediately after the template instance definition. The template instance definition is defined by the TemplateDef rule in 2.2.12, and the template instance data is defined by the TemplateInstanceData rule in 2.2.12.

To emit a substitution in a template, the tool needs to extract the string value from the instance data section. The tool can use the TemplateDefByteLength (specified in 2.2.12) to locate the template instance data quickly.

One special case is when the substitution is of type BinXml. In that case, the tool MUST use the BinXmlValue rule, which is a recursive call. A typical BinXml document contains a template that contains another template in itself.

The other data types MUST be output as follows.

Type	Output format
NullType	Empty string
StringType	Text
AnsiStringType	Text
Int8Type	Signed integer
UInt8Type	Unsigned integer
Int16Type	Signed integer
UInt16Type	Unsigned integer
Int32Type	Signed integer
UInt32Type	Unsigned integer
Int64Type	Signed integer
UInt64Type	Unsigned integer
Real32Type	Signed value having the form [-]dddd.dddd, where ddd3 is one or more decimal digits.
Real64Type	Signed value having the form [-]dddd.dddd, where ddd3 is one or more decimal digits.
BoolType	"true" or "false"
BinaryType	Each byte is displayed as a hexadecimal number with a single space separating each pair of bytes.
GuidType	GUID. Definitions of the fields are as specified in [MS-DTYP]. The text format is {aaaa-bb-cc-dddddd} where aaaa is the hexadecimal value of Data1; bb is the hexadecimal value of Data2; cc is the hexadecimal value of Data3; and dddddd is the hexadecimal value of Data4. For each of the hexadecimal values, all the digits MUST be shown, even if the value is 0.
SizeTType	Hexadecimal integer. The number portion is preceded by the characters "0x". For example, the number 18 is displayed as 0x12.
FileTimeType	Four-digit year "-", 2-digit month "-", 2-digit day "T", 2-digit hour ":", 2-digit seconds "." and 3-digit milliseconds "Z". For example, 2006-10-20T03:23:54.248Z is 3:23:34 am of October 20, 2006.

Type	Output format
SysTimeType	Same as FileTimeType.
SidType	Security ID. A SID type description including the text representation is specified in [MS-DTYP].
HexInt32Type	Hexadecimal integer. The number portion is preceded by the characters "0x". For example, the number 18 is displayed as 0x12.
HexInt64Type	Hexadecimal integer. The number portion is preceded by the characters "0x". For example, the number 18 is displayed as 0x12.

2.2.12.4 Emitting Instruction for the CharRef Rule

Emit the characters "&" and "#" and the decimal string representation of the value.

2.2.12.5 Emitting Instruction for the EntityRef Rule

Emit the character "&" and the text, as specified by the Name rule in 2.2.12.

2.2.12.6 Emitting Instruction for the CDATA Section Rule

Emit the text "<[CDATA[" followed by the text (as specified by the NullTerminatedUnicodeString rule in 2.2.12), and then the string "]]".

2.2.12.7 Emitting Instruction for the PITarget Rule

Emit the text "<?", the text (as specified by the Name rule in 2.2.12), and then the space character " ".

2.2.12.8 Emitting Instruction for the PIData Rule

Emit the text (as specified by the NullTerminatedUnicodeString rule in 2.2.12), and then the text "?>".

2.2.12.9 Emitting Instruction for the CloseStartElement Token Rule

Emit the character ">".

2.2.12.10 Emitting Instruction for the CloseEmptyElement Token Rule

Emit the text "/>".

2.2.12.11 Emitting Instruction for the EndElement Token Rule

Emit the character "<" followed by the text for the element name, and then the text "/>".

2.2.12.12 Emitting Instruction for the TemplateInstanceData Rule

Emitting is suppressed by this rule or any rules invoked recursively.

2.2.13 Event

The Event type is specified to be well-formed XML fragments, as specified in [XML10]. The Event type MUST also conform to the following XML schema, as specified in [XMLSCHEMA1.1/2:2008].

The protocol does not interpret any of the fields in the XML fragment. Client applications (that is, the higher-layer application using the protocol client) that call EvtRpcMessageRender or EvtRpcMessageRenderDefault MUST extract the values specified in the EVENT_DESCRIPTOR structure specified in [MS-DTYP] section 2.3.1. But client applications do not need to interpret these values to call these functions.

```
<xs:schema
targetNamespace=
"http://schemas.microsoft.com/win/2004/08/events/event"
elementFormDefault=
"qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:evt=
"http://schemas.microsoft.com/win/2004/08/events/event">
  <xs:simpleType name="GUIDType">
    <xs:restriction base="xs:string">
      <xs:pattern
value="\{[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-
[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="DataType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="Name" type="xs:string" use="optional"/>
        <xs:attribute name="Type" type="xs:QName" use="optional"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:simpleType name="HexInt32Type">
    <xs:annotation>
      <xs:documentation> Hex 1-8 digits in size</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:pattern value="0[xX][0-9A-Fa-f]{1,8}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="HexInt64Type">
    <xs:annotation>
      <xs:documentation> Hex 1-16 digits in size</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:pattern value="0[xX][0-9A-Fa-f]{1,16}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="ComplexDataType">
    <xs:sequence>
      <xs:element name="Data" type="evt:DataType" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional"/>
  </xs:complexType>
  <xs:complexType name="SystemPropertiesType">
    <xs:sequence>
      <xs:element name="Provider">
        <xs:complexType>
          <xs:attribute name="Name" type="xs:anyURI"
                                    use="optional"/>
          <xs:attribute name="Guid" type="evt:GUIDType"
                                    use="optional"/>
          <xs:attribute name="EventSourceName" type="xs:string"
use="optional"/>
        </xs:complexType>
```

```
      </xs:element>
      <xs:element name="EventID">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:unsignedShort">
              <xs:attribute name="Qualifiers"
                            type="xs:unsignedShort"
           use="optional"/>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="Version" type="xs:byte" minOccurs="0"/>
      <xs:element name="Level" type="xs:byte" minOccurs="0"/>
      <xs:element name="Task" type="xs:unsignedShort"
                              minOccurs="0"/>
      <xs:element name="Opcode" type="xs:byte" minOccurs="0"/>
      <xs:element name="Keywords" type="evt:HexInt64Type"
                                  minOccurs="0"/>
      <xs:element name="TimeCreated" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="SystemTime" type="xs:dateTime"
use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="EventRecordID" minOccurs="0">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="xs:unsignedLong"/>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="Correlation" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="ActivityID" type="evt:GUIDType"
use="optional"/>
          <xs:attribute name="RelatedActivityID"
                        type="evt:GUIDType"
            use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="Execution" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="ProcessID" type="xs:unsignedInt"
use="required"/>
          <xs:attribute name="ThreadID" type="xs:unsignedInt"
use="required"/>
          <xs:attribute name="ProcessorID" type="xs:byte"
use="optional"/>
          <xs:attribute name="SessionID" type="xs:unsignedInt"
use="optional"/>
          <xs:attribute name="KernelTime" type="xs:unsignedInt"
use="optional"/>
          <xs:attribute name="UserTime" type="xs:unsignedInt"
use="optional"/>
          <xs:attribute name="ProcessorTime" type="xs:unsignedInt"
use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="Channel" type="xs:anyURI" minOccurs="0"/>
      <xs:element name="Computer" type="xs:string"/>
      <xs:element name="Security" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="UserID" type="xs:string"
use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:any namespace="##other" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
```

```
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
  <xs:complexType name="EventDataType">
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Data" type="evt:DataType"/>
        <xs:element name="ComplexData" type="evt:ComplexDataType"/>
      </xs:choice>
      <xs:element name="Binary" type="xs:hexBinary" minOccurs="0">
        <xs:annotation>
          <xs:documentation>Classic eventlog binary data
</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="optional"/>
  </xs:complexType>
  <xs:complexType name="UserDataType">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
  <xs:complexType name="DebugDataType">
    <xs:sequence>
      <xs:element name="SequenceNumber" type="xs:unsignedInt"
minOccurs="0"/>
      <xs:element name="FlagsName" type="xs:string" minOccurs="0"/>
      <xs:element name="LevelName" type="xs:string" minOccurs="0"/>
      <xs:element name="Component" type="xs:string"/>
      <xs:element name="SubComponent" type="xs:string"
minOccurs="0"/>
      <xs:element name="FileLine" type="xs:string" minOccurs="0"/>
      <xs:element name="Function" type="xs:string" minOccurs="0"/>
      <xs:element name="Message" type="xs:string"/>
      <xs:any namespace="##other" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
  <xs:complexType name="ProcessingErrorDataType">
    <xs:sequence>
      <xs:element name="ErrorCode" type="xs:unsignedInt"/>
      <xs:element name="DataItemName" type="xs:string"/>
      <xs:element name="EventPayload" type="xs:hexBinary"/>
      <xs:any namespace="##other" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
  <xs:complexType name="RenderingInfoType">
    <xs:sequence>
      <xs:element name="Message" type="xs:string" minOccurs="0"/>
      <xs:element name="Level" type="xs:string" minOccurs="0"/>
      <xs:element name="Opcode" type="xs:string" minOccurs="0"/>
      <xs:element name="Task" type="xs:string" minOccurs="0"/>
      <xs:element name="Channel" type="xs:string" minOccurs="0"/>
      <xs:element name="Publisher" type="xs:string" minOccurs="0"/>
      <xs:element name="Keywords" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Keyword" type="xs:string"
                                       minOccurs="0"
                        maxOccurs="64"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:any namespace="##other" minOccurs="0"
maxOccurs="unbounded"/>
```

```
    </xs:sequence>
    <xs:attribute name="Culture" type="xs:language" use="required"/>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
  <xs:complexType name="EventType">
    <xs:sequence>
      <xs:element name="System" type="evt:SystemPropertiesType"/>
      <xs:choice>
        <xs:element name="EventData" type="evt:EventDataType"
 minOccurs="0">
          <xs:annotation>
            <xs:documentation>Generic event</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="UserData" type="evt:UserDataType"
 minOccurs="0">
          <xs:annotation>
            <xs:documentation>Custom event</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="DebugData" type="evt:DebugDataType"
 minOccurs="0">
          <xs:annotation>
            <xs:documentation>WPP debug event</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="BinaryEventData" type="xs:hexBinary"
                                           minOccurs="0">
          <xs:annotation>
            <xs:documentation>
               Non schematized event
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="ProcessingErrorData"
type="evt:ProcessingErrorDataType" minOccurs="0">
          <xs:annotation>
            <xs:documentation>Instrumentation event
</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:any namespace="##other" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:choice>
      <xs:element name="RenderingInfo"
                  type="evt:RenderingInfoType"
          minOccurs="0"/>
      <xs:any namespace="##other" minOccurs="0"
                                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>
  <xs:element name="Event" type="evt:EventType"/>
</xs:schema>
```

2.2.14 Bookmark

The bookmark type specifies the cursor position in the event query or subscription result set. Note that bookmarks are passed from the client to the server by using the XML representation that is specified in this section. In contrast, the server passes a binary representation of bookmarks to the client as specified in section 2.2.16.

The bookmark type is specified to be well-formed XML fragments as specified in [XML10] and that conforms to the following XML schema as specified in [XMLSCHEMA1.1/2:2008].

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="BookmarkType">
<xs:attribute name="Channel" type="xs:anyURI" use="required"/>
<xs:attribute name="RecordId" type="xs:long" use="required"/>
<xs:attribute name="IsCurrent" type="xs:boolean" use="optional"
default="false"/>
</xs:complexType>
<xs:complexType name="BookmarkListType">
<xs:sequence>
<xs:element name="Bookmark" type="BookmarkType"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element name="BookmarkList" type="BookmarkListType"/>
</xs:schema>


<!-- Example bookmark for the Application Log: -->

<BookmarkList>
   <Bookmark Channel="Application" RecordId="2004" IsCurrent="True"/>
</BookmarkList>
```

Elements	
BookmarkList	The top-level element that contains a list of bookmarks for the individual event logs.
Bookmark	Defines the cursor position in an event log specified by the **Channel** attribute.

Attributes	
Channel	The name of the channel.
RecordId	The logical event record number in the event log specified by the **Channel** attribute.
IsCurrent	Specifies if the event at the cursor position corresponds to the channel corresponding to the element. A subscription or query can apply to several channels. In this case, there is a Bookmark element for each channel. However, only one channel can be the one with the last event, and its IsCurrent attribute MUST be set to true.

2.2.15 Filter

The filter type is an XPath filter used to select events in the event logs, and is specified to be a subset of XPath 1.0, as specified in [XPATH].

2.2.15.1 Filter XPath 1.0 Subset

The filter type supports the following XPath 1.0 subset:

Location Paths:

- Axis
- Child

- Attribute
- Node tests"*" - wildcard
- NCName
- text

Expressions:

- or
- and
- =, !=
- <=, <, >=, >
- "('Expr')"
- Literal
- Number
- FunctionCall

Core Function Library: position()

The data model supported by the EventLog filter for representing XML events is a restricted form of what is used for XPath 1.0. Evaluation of each event MUST be restricted to forward-only, in-order, depth-first traversal of the XML.

The data model used differs in the following specific ways:

- Because only the child and attribute axes are supported, generating a string value for nodes is not supported.
- Generating an expanded name for nodes is not supported.
- Evaluation of nodes in forward document order is supported, but reverse document order is not.
- Node sets are not supported.
- The stream of XML events is represented as the set of top-level elements in a "virtual" document. The root of this document is implied and does not have a formal name. This implies that absolute location paths are not supported. The current context node at the start/end of each XML event evaluation is this root node.
- The evaluation context is a restricted version of that for XPath 1.0. It contains a current context node and a nonzero positive integer representing the current position. It does not contain the context size, nor does it contain variable bindings. It has a smaller function library, and it has no namespace scoping support.
- Namespace, processing, and comment nodes are not supported.

2.2.15.2 Filter XPath 1.0 Extensions

This protocol's filter type defines the following functions that are not part of the set defined by the XPath 1.0 specification, but are specific to this protocol.

Core Function Library:

- Boolean band(bitfield, bitfield)

  The band(bitfield, bitfield) bitwise AND function takes two bitfield arguments, performs a bitwise AND.

- number timediff(SYSTEM_TIME)

  The timediff(SYSTEM_TIME) function calculates the difference in milliseconds between the argument-supplied time and the current system time. The result MUST be positive if the system time is greater than the argument time, zero if the system time is equal to the argument time, and negative if the system time is less than the argument time.

- number timediff(SYSTEM_TIME, SYSTEM_TIME)

  The timediff(SYSTEM_TIME, SYSTEM_TIME) function calculates the difference in milliseconds between the first and second argument-supplied times. The result MUST be positive if the second argument is greater than the first, zero if they are equal, and negative if the second argument is less than the first.

Data Model:

This protocol's filter supports an expanded set of data types. These are:

- Unicode (as specified in [UNICODE]) string
- ANSI string. In this specification, ANSI strings refer to multi-byte strings in which the encoding is controlled by the current system code page. One of the most common code pages is ANSI Latin-1, as specified in [ISO/IEC-8859-1].
- BOOLEAN
- Double
- UINT64, which is an unsigned 64-bit integer
- GUID, as specified in [MS-RPCE]
- SID, as specified in [MS-DTYP]
- SYSTEMTIME, as specified in [MS-DTYP]
- FILETIME, as specified in [MS-DTYP]
- Binary large object (BLOB)
- Bitfield (64 bits)

In XPath expressions, the additional data types are expressed as strings and converted to the wanted type for expression evaluation. The conversion is based on the syntax of the string literal.

During evaluation of an XPath expression, a data string is determined to represent one of these additional types if it conforms to the syntactical representation for that type. The scopes of syntactic representations overlap such that it is possible for a string to have a valid representation as more than one type. In this case, a representation for each such type is retained and used in accordance with the following implicit conversion rules at event evaluation time.

The GUID type is converted to and from a string, as specified in [RFC4122]. The SID type is converted as specified in [MS-DTYP].

The ABNF for the remaining types is as follows, where DIGIT and HEXDIGIT are as specified in [RFC4234] Appendix B.

```
Double = 0*1(SIGN) 0*(DIGIT) 0*1("." 1*(DIGIT))
 0*1(("d" / "D" / "e" / "E") 0*1(SIGN) 0*1(DIGIT))
SIGN = "+" / "-"
UINT64 = "0" ("x" / "X") 1*DIGIT
SYSTEMTIME = FILETIME
FILETIME = date-time
date-fullyear   = 4DIGIT
date-month      = 2DIGIT  ; 01-12
date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on month-year
time-hour       = 2DIGIT  ; 00-23
time-minute     = 2DIGIT  ; 00-59
time-second     = 2DIGIT  ; 00-59
time-msecs      = "." 1*3DIGIT
time-offset     = "Z"
partial-time    = time-hour ":" time-minute ":" time-second [time-msecs]
full-date       = date-fullyear "-" date-month "-" date-mday
full-time       = partial-time time-offset
date-time       = full-date "T" full-time
BinaryBlob = 1*HEXDIG
bitfield = UINT64
```

Additionally, if the string is determined to be of a numeric type, it is determined to be of Boolean type with value false if its numeric value is zero, and true otherwise. If the string is not of numeric type but is a string of value "true" or "false", it is determined to be of Boolean type with value true or false, respectively.

FILETIME and SYSTEMTIME are interpreted as GMT times.

All of the comparison operators are type-wise aware of the additional data types. For the cases of string (both Unicode and ASCII), Boolean, and Double, evaluation is the same as for XPath 1.0.

For the remaining types, implicit type coercion in the expression L1 op L2 is governed by the following exhaustive rule set:

- If L2 is a string, L1 MUST be converted to a string.
- If L2 is a Boolean, L1 MUST be converted to a Boolean.
- If L2 is a GUID, SID, SYSTEMTIME, or FILETIME, L1 MUST be converted to a literal of the same type, if possible. If the conversion cannot be performed, the result of the evaluation MUST be false.
- If L2 is of numeric type, including bitfield, and L1 is of type double, L2 MUST be converted to double.
- If L2 is of numeric type, including bitfield, and L1 is of an unsigned integral type, L2 MUST be converted to an unsigned type.

2.2.16 Query

The query type specifies an XML document used to select events in the event log by using well-formed XML (as specified in [XML10]) and is defined by the following XSD (as specified in [XMLSCHEMA1.1/2:2008]).

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace=
  "http://schemas.microsoft.com/win/2004/08/events/eventquery"
elementFormDefault="qualified"
xmlns="http://schemas.microsoft.com/win/2004/08/events/eventquery"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="QueryType">
    <xs:choice maxOccurs="unbounded">
```

```
        <xs:element name="Select">
          <xs:complexType mixed="true">
            <xs:attribute name="Path" type="xs:anyURI"
             use="optional"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Suppress">
          <xs:complexType mixed="true">
            <xs:attribute name="Path" type="xs:anyURI"
              use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:attribute name="Id" type="xs:long" use="optional"/>
      <xs:attribute name="Path" type="xs:anyURI" use="optional"/>
    </xs:complexType>
    <xs:complexType name="QueryListType">
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="Query" type="QueryType"/>
      </xs:sequence>
    </xs:complexType>
    <xs:element name="QueryList" type="QueryListType"/>
  </xs:schema>
```

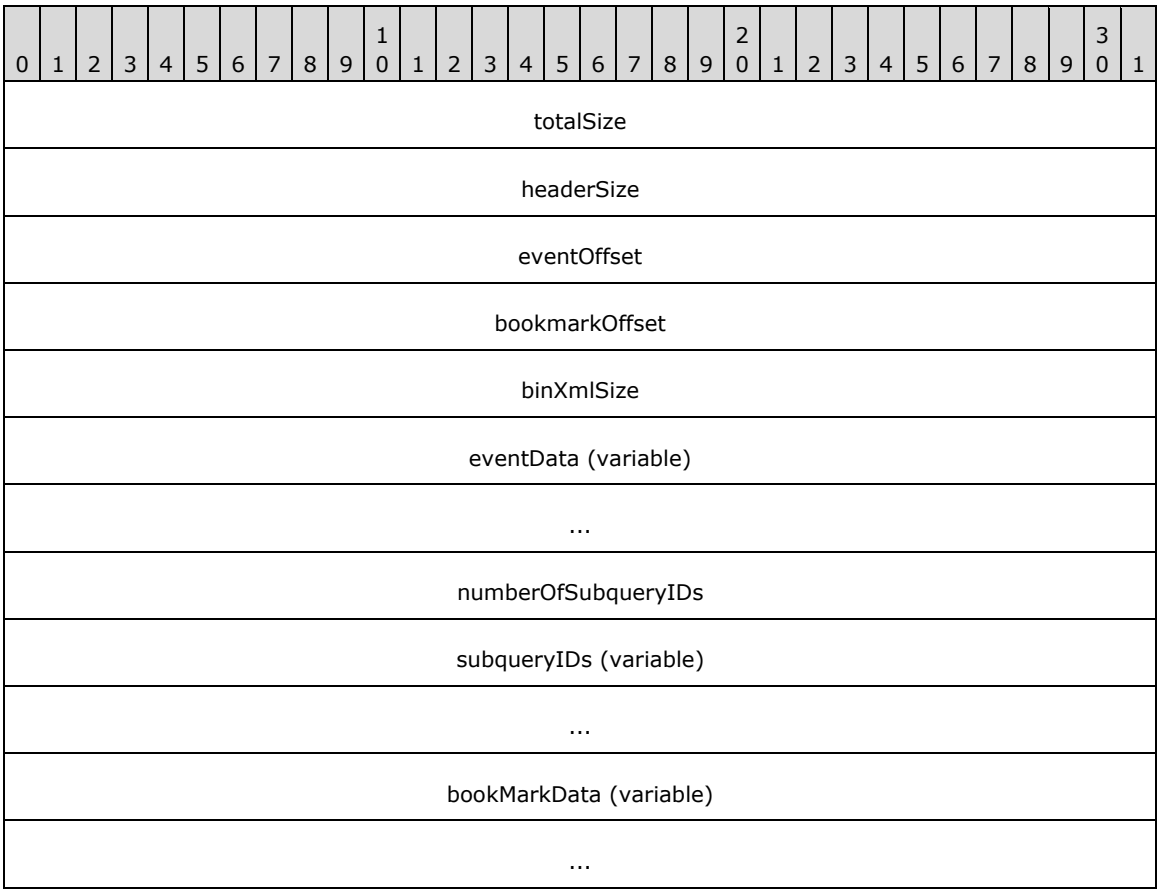
Elements	Description
QueryList	Lists the query elements. The event query result set contains events matched by any of the query elements.
Query	Defines a set of selectors and suppressors. Query elements are referred to as **subqueries**.
Select	Defines an event filter for events included in the result set (unless rejected by a suppressor in the same query element), as specified in section 2.2.15.
Suppress	Defines an event filter for events omitted from the result set (even if the same events were selected by a selector in the same query element), as specified in section 2.2.15.

Attributes	Description
ID	Defines the ID of a subquery so that a consumer can determine what subquery out of many caused the record to be included in a result set. Multiple subqueries using the same IDs are not distinguished in the result set. For information on subquery IDs, see section 2.2.17.
Path	Specifies either the name of a channel or a path to a backup event log for query elements, selectors, and suppressors. A path specified for the query element applies to the selectors and suppressors it contains that do not specify a path of their own. If a path begins with file://, it MUST be interpreted as a Uniform Resource Identifier (URI) path to a backup event log file, as specified in [RFC3986], that uses file as a scheme; for example, file://c:/dir1/dir2/file.evt. Otherwise, a path MUST be interpreted as a channel name.

2.2.17 Result Set

An event query or subscription returns multiple events in the result set. The result set is a buffer containing one or more variable length **EVENT_DESCRIPTOR** structures, as specified in [MS-DTYP] section 2.3.1. Methods that return multiple events always return an array of offsets into the buffer for the individual events.

The records are transferred as a set of bytes. All integer fields in this structure MUST be in little-endian byte order (that is, least significant byte first).



totalSize (4 bytes): A 32-bit unsigned integer that contains the total size in bytes of this structure, including the header.

headerSize (4 bytes): This MUST always be set to 0x00000010.

eventOffset (4 bytes): This MUST always be set to 0x00000010.

bookmarkOffset (4 bytes): A 32-bit unsigned integer that contains the byte offset from the start of this structure to **bookMarkData**.

binXmlSize (4 bytes): Size in bytes of the BinXml data in the **eventData** field.

eventData (variable): A byte-array that contains variable length BinXml data.

numberOfSubqueryIDs (4 bytes): Number of **subqueryIDs** fields that follow. This is 0 if the event is selected by an XPath expression (rather than a **structured XML query**).

subqueryIDs (variable): An array of subquery IDs. Events that are selected using a structured XML query can be selected by one or more subqueries. Each subquery has either an ID specified in the XML element that defines the subquery, or defaults to 0xFFFFFFFF. This list has an entry for each subquery that matches the event. If two subqueries select the event, and both use the same ID, the ID only is listed once.

bookMarkData (variable): A byte-array that contains variable length bookmark data, as specified:

A query can refer to several channels or backup event logs. A subscription can refer to several channels. To accurately record the state of a query, it is necessary to know where the file cursor (bookmark) is with respect to those channels or backup event logs. The bookmark data is encoded as follows. Note that all integer fields in this structure MUST be in little-endian byte order (that is, least significant byte first).

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
bookmarkSize																															
headerSize																															
channelSize																															
currentChannel																															
readDirection																															
recordIdsOffset																															
logRecordNumbers (variable)																															
...																															

bookmarkSize (4 bytes): A 32-bit unsigned integer that contains the total size in bytes of the bookmark, including the header and **logRecordNumbers**.

headerSize (4 bytes): A 32-bit unsigned integer, and MUST be set to 0x00000018.

channelSize (4 bytes): A 32-bit unsigned integer that contains the number of channels in the query. This is the number of elements in **logRecordNumbers**.

currentChannel (4 bytes): A 32-bit unsigned integer that indicates what channel the current event is from.

readDirection (4 bytes): A 32-bit unsigned integer that contains the read direction. 0x00000000 indicates chronological order based on time written, and 0x00000001 indicates reverse order.

recordIdsOffset (4 bytes): A 32-bit unsigned integer that contains the byte offset from the start of the header to **logRecordNumbers**.

logRecordNumbers (variable): An array of 64-bit unsigned integers that contain the record numbers for each of the channels or backup event logs. The order of the record numbers MUST match the order of the channels or backup event logs in the query (for example, the first channel in the query corresponds to the first member of the array).

2.2.18 BinXmlVariant Structure

Some of the methods use the following structures for returning data. In particular, the BinXmlVariant structure is used for returning information about a channel or backup event log. This structure is custom-marshaled. The integer fields in this structure MUST be in little-endian byte order (that is, least significant byte first).

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
union																															
...																															
count																															
type																															

union (8 bytes): 8 bytes of data. Interpretation is based on type.

count (4 bytes): Not used. Can be set to any arbitrary value when sent and MUST be ignored on receipt.

type (4 bytes): Specifies the union type.

Value	Meaning
BinXmlVarUInt32 0x00000008	The union field contains an unsigned long int, followed by 4 bytes of arbitrary data that MUST be ignored.
BinXmlVarUInt64 0x0000000A	The union field contains an unsigned __int64.
BinXmlVarBool 0x0000000D	The union field contains an unsigned long int, followed by 4 bytes of arbitrary data that MUST be ignored.
BinXmlVarFileTime 0x00000011	The union field contains a **FILETIME**, as specified in [MS-DTYP] Appendix A.

2.2.19 error_status_t

The error_status_t return type is used for all methods. This is a Win32 error code.

This type is declared as follows:

```
typedef unsigned long error_status_t;
```

2.2.20 Handles

The following handles are used when a client connects to the server.

```
typedef [context_handle] void* PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION;
typedef [context_handle] void* PCONTEXT_HANDLE_LOG_QUERY;
typedef [context_handle] void* PCONTEXT_HANDLE_LOG_HANDLE;
typedef [context_handle] void* PCONTEXT_HANDLE_OPERATION_CONTROL;
typedef [context_handle] void* PCONTEXT_HANDLE_PUBLISHER_METADATA;
typedef [context_handle] void* PCONTEXT_HANDLE_EVENT_METADATA_ENUM;
```

For information on handle security, see section 5.1.

2.2.21 Binding Handle

This protocol reuses the RPC binding handle as the logical connection between the client and server. Numerous methods described in section 3.1.4 take the binding handle as the first parameter.

This type is declared as follows:

```
typedef I_RPC_HANDLE RPC_BINDING_HANDLE;
```

This data type declares a binding handle containing information that the RPC run-time library uses to access binding information. For more information about the RPC binding handle, see [MSDN-BNDHNDLS].

2.3 Message Syntax

2.3.1 Common Values

The following common values are used throughout this specification.

Name	Value
MAX_PAYLOAD	(2 * 1024 * 1024)
MAX_RPC_QUERY_LENGTH	(MAX_PAYLOAD / sizeof(WCHAR))
MAX_RPC_CHANNEL_NAME_LENGTH	512
MAX_RPC_QUERY_CHANNEL_SIZE	512
MAX_RPC_EVENT_ID_SIZE	256
MAX_RPC_FILE_PATH_LENGTH	32768
MAX_RPC_CHANNEL_PATH_LENGTH	32768
MAX_RPC_BOOKMARK_LENGTH	(MAX_PAYLOAD / sizeof(WCHAR))
MAX_RPC_PUBLISHER_ID_LENGTH	2048
MAX_RPC_PROPERTY_BUFFER_SIZE	MAX_PAYLOAD
MAX_RPC_FILTER_LENGTH	MAX_RPC_QUERY_LENGTH
MAX_RPC_RECORD_COUNT	1024
MAX_RPC_EVENT_SIZE	MAX_PAYLOAD
MAX_RPC_BATCH_SIZE	MAX_PAYLOAD
MAX_RPC_RENDERED_STRING_SIZE	MAX_PAYLOAD
MAX_RPC_CHANNEL_COUNT	8192
MAX_RPC_PUBLISHER_COUNT	8192
MAX_RPC_EVENT_METADATA_COUNT	256
MAX_RPC_VARIANT_LIST_COUNT	256
MAX_RPC_BOOL_ARRAY_COUNT	(MAX_PAYLOAD / sizeof(BOOL))

Name	Value
MAX_RPC_UINT32_ARRAY_COUNT	(MAX_PAYLOAD / sizeof(UINT32))
MAX_RPC_UINT64_ARRAY_COUNT	(MAX_PAYLOAD / sizeof(UINT64))
MAX_RPC_STRING_ARRAY_COUNT	(MAX_PAYLOAD / 512)
MAX_RPC_GUID_ARRAY_COUNT	(MAX_PAYLOAD / sizeof(GUID))
MAX_RPC_STRING_LENGTH	(MAX_PAYLOAD / sizeof(WCHAR))

3 Protocol Details

3.1 Server Details

The server handles client requests for any of the messages specified in section 2, and operates on the logs and configuration on the server. For each of those messages, the behavior of the server is specified in section 3.1.4.

3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This specification does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this specification.

3.1.1.1 Events

An event is an entity that describes some occurrence in the system. All events in the system can be represented as XML (though in the protocol they only appear as BinXml, as specified in section 3.1.4.7).

An event is identified by a numeric code (EventID) and a set of attributes (qualifiers, task, opcode, level, version, and keywords). It also contains its publisher name and originating channel, and can also contain event specific data. See [MSDN-EVENTRECORD], [MSDN-EVENT_HEADER], and [MSDN-EVENT_DESCRIPTOR], for a description of the layout of event data structures which contains this information.

3.1.1.2 Publishers

Events are raised to the system by a publisher (though this is not through the EventLog Remoting Protocol Version 6.0). For more information on how to provide events, see [MSDN-ProvEvts].

The publisher is registered within the system and has an identifier and a publisher name. The identifier of a publisher is a UUID, which is 16 bytes. For example, {10ccdb74-baf6-4164-b765-c292096626df} can be served as the identifier for a publisher. The publisher resource file contains all the metadata information for the publisher (as specified in section 3.1.1.14). The message file of a publisher contains all the description strings of a publisher (as specified in section 3.1.1.14). The publisher parameter file is another message file that is used for parameter substitution (as specified in section 3.1.1.14). The publisher name is an arbitrary Unicode string. The publisher provides the events into channels (as specified in section 3.1.1.4), it declares the channels that can be used during the registration. The server saves all the channels a publisher declares and associates them with the publisher.

The following is an example of two registered publishers and their associated channels:

Publisher1: Publisher1 Name, Publisher1 Identifier, Publisher1 Resource File, Publisher1 Message File, Publisher1 Parameter File

- Channel A: Channel A reference ID, Channel A reference flag, Channel A start index
- Channel B: Channel B reference ID, Channel B reference flag, Channel B start index

Publisher2: Publisher2 Name, Publisher2 Identifier

- Channel C: Channel C reference ID, Channel C reference flag, Channel C start index

- Channel B: Channel B reference ID, Channel B reference flag, Channel B start index

In this example, Publisher1 indicates that it will provide events to two channels, Channel A and Channel B. Publisher2 indicates that it will provide events to Channel C and Channel B. So in this case, Channel B is shared between two publishers and it can contain the events from both publishers.

The channel reference ID is the relative index number of a channel, while the channel start index is an absolute start offset for the channel in the channel table. For example, publisher 1 declares two channels: Channel A and Channel B. Thus the channel A reference ID is 0 because it is the first channel of publisher 1 and the channel B reference ID is 1 since it is the second one relative to the publisher. But at the same time, the declared channels have to be registered in the channel table (as specified in section 3.1.1.5). The same channel will get a different index number when the channels are registered to the channel table. For example, when the channel table contains 10 channels already, and when channel A is registered, it will get an index of 11, and channel B will get an index 12. As such, the start index of channel A is 11 and the start index of channel B is 12.

The channel reference flag is a numeric value used for the purpose of extension. There is no recommendation on how the server SHOULD use it. By default, all the channel reference flags are 0. The server can use it for any special purpose.

The publisher is only registered on the local machine. It cannot be registered to a remote server.

Publisher identifiers can be obtained through the protocol and from events that conform to the event XSD, as specified in section 2.2.13. Publisher identifiers MUST be unique.

Also, publishers can have additional metadata registered in the system, consisting of a set of attribute/value pairs, as specified in sections 3.1.4.25 and 3.1.4.26. This can be obtained through the protocol by using the publisher identifier. This metadata typically includes message tables that are used for displaying localized messages. The metadata information is registered to the server when the publisher is installed on the server. Installing a publisher is not in the scope of this protocol.

Note A subset of the set of publishers is logically shared with the abstract data model of the obsolete EventLog Remoting Protocol, if it is also supported. That is, all event sources registered with the original EventLog Remoting Protocol can be enumerated via this protocol (the EventLog Remoting Protocol Version 6.0), but not vice versa.

3.1.1.3 Publisher Tables

A publisher table is an array of registered publishers on the server. Each publisher in the table SHOULD contain the publisher name, the publisher identifier, and the channels to which the publisher will write events.

A typical publisher table structure is as follows:

<Publisher1 Name><Publisher1 Identifier><Channel List for Publisher 1><Resource File of Publisher 1><Parameter File of Publisher 1><Message File of Publisher 1>

<Publisher2 Name><Publisher2 Identifier><Channel List for Publisher 2><Resource File of Publisher 2><Parameter File of Publisher 2><Message File of Publisher 2>

<Publisher n Name><Publisher n Identifier><Channel List for Publisher n><Resource File of Publisher n><Parameter File of Publisher n><Message File of Publisher n>

The channel list for each publisher is a list of channels. The channel list format is specified in the publisher examples in section 3.1.1.2.

The publisher table is saved on the server's disk and is permanent. Adding or removing entries in this table can only be executed by the server or some automatic configuration tool provided by the server.

The server reads the table from disk at start up and loads it into memory for fast processing and lookup. The client MAY be able to change some information in the memory but cannot touch the information saved on disk. A changed memory snapshot can only be applied toward the copy on disk through the **EvtRpcAssertConfig** method (as specified in section 3.1.4.29) or **EvtRpcRetractConfig** (as specified in section 3.1.4.30).

The server can also add access control of the publisher table entry, or the server MAY define the access rights for the table entry. By doing this, the server can control the client access rights to the publishers in the table. For example, the server can set the publisher 1's information to be accessed only by administrators. If a non-administrator client wants to get the information for publisher 1, the server can deny access with an error.

3.1.1.4 Channels

A channel is a named stream of events. It serves as a logical pathway for transporting events from the event publisher to a log file and possibly a subscriber. It is a sink that collects events.

Publishers declare the channels they are going to generate events into. The channels they declare MUST have an identifier and that identifier MUST be unique. The publishers can also import the existing channels in the server simply by referencing the channel identifier. Each channel MUST have a unique name (also called a Channel Path). The name of the channel is a string and the server SHOULD register the channel with the identifier and its name. The server keeps the table of registered channels.

A channel name can be obtained through the protocol method **EvtRpcGetChannelList** as specified in section 3.1.4.20 and from events that conform to event schema, as specified in section 2.2.13.

Channels have a set of configurable properties (as specified in section 3.1.4.21) that affect the behavior of the channel within the system. The configurable properties are the channel interface to the client. The channel data structure SHOULD contain these properties as internal data fields so that the server can track the changes of the value of these properties and adjust the behavior based on the latest property values. The required channel properties are specified in the following table:

Name	Meaning
Enabled	If true, the channel can accept new events. If false, any attempts to write events into this channel are automatically dropped.
Channel Isolation	One of three values: **0:** Application. Use security setting (channel access property) of Application channel. **1:** System. Use security setting (channel access property) of System channel. **2:** Custom. The channel has its own explicit security settings.
type	One of four values: **0:** Admin **1:** Operational **2:** Analytic **3:** Debug For more information, see [MSDN-EVTLGCHWINEVTLG].
OwningPublisher	Name of the publisher that defines and registers the channel with the system.
Classic	If true, the channel represents an event log created according to the EventLog Remoting Protocol, not this protocol (EventLog Remoting Protocol Version 6.0).
Access	A Security Descriptor Description Language (SDDL) string, as specified in [MS-DTYP], which represents access permissions to the channels. The server uses the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2) to perform the access control.

Name	Meaning
	A client with read access to the channel can read the properties of the channel and read the events from the channel. A client with write access to the channel can change the properties of the channel and write the events into the channel. A client with clear access to the channel can perform the clear log operation toward the channel. Note that the only access permissions defined for channels are read, write, and clear.
Retention	If set to true, events can never be overwritten unless explicitly cleared. If set to false, events are overwritten as needed when the event log is full.
AutoBackup	When set to true, the event log file associated with the channel is closed as soon as it reaches the maximum size specified by the MaxSize property, and a new file is opened to accept new events. If the new file reaches maximum size, another new file will be generated and the previous new file will be backed up. The events in backed up files cannot be queried from this channel in the server unless the client specifies the backup log file names in a separate query.
MaxSize	The value that indicates at which point the size (in bytes) of the event log file stops increasing. When the size is greater than or equal to this value, the file growth stops.
LogFilePath	File path to the event log file for the channel.
Level	Events with a level property less than or equal to this specified value are logged to the channel.
Keywords	Events with a keyword bit contained in the Keywords bitmask set are logged to the channel.
ControlGuid	A GUID value. For more information on the server behavior for this property, see section 3.1.4.22
BufferSize	Size of the events buffer (in kilobytes) used for asynchronous event delivery. This property is for providing events. Typically the events generated by a publisher are first written to memory buffers on the server. Once the buffer used is full, that buffer is written to a disk file. The BufferSize is used to specify the size of the buffer. The server allocates buffers according to the BufferSize value. The number of buffers the server can allocate is controlled by the MinBuffers and MaxBuffers properties. The server's specific implementation can allocate any number of buffers between MinBuffers and MaxBuffers.
MinBuffers	The minimum number of buffers used for asynchronous event delivery. For more information, see the preceding BufferSize description.
MaxBuffers	The maximum number of buffers used for asynchronous event delivery. For more information, see the preceding BufferSize description.
Latency	The number of seconds of inactivity (if events are delivered asynchronously and no new events are arriving) after which the event buffers MUST be flushed to the server. As specified in the description for BufferSize property, the server keeps a number of buffers when writing events. If the buffers are full, the server writes the buffers to disk file. However, if a certain amount of time elapses and the buffers are still not full, the server SHOULD write the buffers to disk. That certain amount of time is the latency property.
ClockType	One of two values: ▪ 0: SystemTime. Use the system time. When set to this value, the server uses the system time type (which is low-resolution on most platforms) for a time stamp field of any event it writes into this channel. ▪ 1: Query Performance Counter. The server uses a high-resolution time type for the time stamp field of any event it writes into this channel.
SIDType	One of two values:

Name	Meaning
	▪ 0: The events written by the server to this channel will not include the publisher's SID. ▪ 1: The events written by the server to this channel will include the publisher's SID.
PublisherList	List of publishers that can raise events into the channel. For more information on this field, see section 3.1.4.24.
FileMax	Maximum number of log files associated with an analytic or debug channel. When the number of logs reaches the specified maximum, the system begins to overwrite the logs, beginning with the oldest. A FileMax value of 0 or 1 indicates that only one file is associated with this channel. A FileMax of 0 is default.

These properties can be observed or modified through this protocol. The methods to observe and modify the channel properties are **EvtRpcGetChannelConfig** (as specified in section 3.1.4.21) and **EvtRpcPutChannelConfig** (as specified in section 3.1.4.22).

3.1.1.5 Channel Table

A channel table is an array of registered channels. Each channel's table item SHOULD contain the channel identifier which is a Unicode string of the channel name and its properties (as specified in section 3.1.4.21).

A typical channel table structure is as follows:

<Channel1 Identifier><Properties list>

<Channel2 Identifier><Properties list>

<Channel n Identifier><Properties list>

Each properties list is a list of configurable channel properties (as specified in section 3.1.4.22).

The channel table is saved on the server's disk and is permanent. Adding or removing entries in this table can only be executed by the server or some automatic configuration tool provided by the server.

The server reads the table from disk at start up and loads it into memory for fast processing and lookup. The client MAY be able to change some information in the memory but can't touch the information saved on disk. A changed memory snapshot can only be applied toward the copy on disk through the **EvtRpcAssertConfig** method (as specified in section 3.1.4.29) or **EvtRpcRetractConfig** (as specified in section 3.1.4.30).

3.1.1.6 Logs

A log is a file system file containing events. Any channel has a log associated with it. In this case, the log is identified by using the channel identifier and is a live event log.

A log can also exist as a standalone file. In this case, the log is identified by using the file system path of that log file, which has no associated channel.

Logs have a set of properties which can be retrieved through the protocol method **EvtGetLogFileInfo** (as specified in section 3.1.4.15). Such properties are log creation time, last access time, last written time, log file size, extra attributes of the log, number of events in the log, oldest event record in the log, and the log full flag.

A log file usually consists of file header and file body. The header SHOULD contain metadata information of the event log itself. A recommended header SHOULD at least contain the following:

BOOL isLogFull: This flag indicates the log is full.

unsigned__int64 oldestRecordNumber: The oldest event log record ID in the log file.

unsigned__int64 numberOfRecords: The number of event log records in the log file.

unsigned__int64 curPhysicalRecordNumber: The physical record number of the latest record in the log file.

The server does not maintain these fields. These fields are maintained by publishers as events are added to the log file.

The log body consists of all the event records in binXML format (as specified in section 3.1.4.7).

The log file associated with a channel is maintained and updated by the server. This protocol assumes that the log files associated with channels are in the format described above. Note that rules for creating such files are out of scope for this protocol.

Through this protocol, the events in a channel (which are stored in a live event log file) can be exported into a standalone log file by the method **EvtRpcExportLog** as specified in section 3.1.4.17. This protocol defines other methods by which clients can manage log files and obtain information about them.

Note A subset of the log files is logically shared with the abstract data model of the obsolete Eventlog Remote Protocol (as specified in [MS-EVEN]), if it is also supported. That is, all log files accessible with the original Eventlog Remote Protocol (as specified in [MS-EVEN]) are also accessible via this protocol (Eventlog Remote Protocol Version 6.0), but not necessarily vice versa.

3.1.1.7 Localized Logs

A log file that is exported from a channel does not contain the message strings for event levels, opcodes, tasks, keywords, and event descriptions. Support engineers often need to read the events from a customer's exported event log, including these strings, in order to diagnose issues on the customer's system. This protocol provides for localizing an exported log file in order to provide localized versions of those strings in whatever locale the support engineer requires. The EvtRpcLocalizeExportLog (section 3.1.4.18) protocol method generates the locale-dependent file.

A localized log is composed of two files. One is the standalone exported log file, as specified in section 3.1.1.6. The other is a locale-dependent file that contains all the localized strings.

The locale-dependent file groups localized strings into sections that correspond to the fields of events. For each publisher represented in the file, there is a section for event levels, opcodes, tasks, keywords, and event descriptions. Within each section, localized strings are presented in ordered records that include additional information, such that tools for viewing localized log files (which are out of scope for this protocol) can match each string with its corresponding event in the standalone exported log file. Following the final publisher in the file, is an event section that contains the localized event description strings for each event in the standalone exported log file.

The locale-dependent file has the following structure:

[Publisher Section]

<Publisher Name 0>

  [Level Section]

    <0> <Level value 0> <MessageID of level 0> <Localized string for Level 0>

    ...

<n> <Level value n> <MessageID of level n> <Localized string for Level n>

[Tasks Section]

<0> <Task value 0> <MessageID of task 0> <Localized string for task 0>

…

<m> <Task value m> <MessageID of task m> <Localized string for task m>

[Opcodes Section]

<0> <Opcode value 0> <MessageID of opcode 0> <Localized string for opcode 0>

…

<k> <Opcode value k> <MessageID of opcode k> <Localized string for opcode k>

[Keywords Section]

<0> <Keyword value 0> <MessageID of keyword 0> <Localized string of keyword 0>

…

<t> <Keyword value t> <MessageID of keyword t> <Localized string of keyword t>

…

<Publisher Name n>

[Level Section]

<0> <Level value 0> <MessageID of level 0> <Localized string for Level 0>

...

<n> <Level value n> <MessageID of level n> <Localized string for Level n>

[Tasks Section]

<0> <Task value 0> <MessageID of task 0> <Localized string for task 0>

…

<m> <Task value m> <MessageID of task m> <Localized string for task m>

[Opcodes Section]

<0> <Opcode value 0> <MessageID of opcode 0> <Localized string for opcode 0>

…

<k> <Opcode value k> <MessageID of opcode k> <Localized string for opcode k>

[Keywords Section]

<0> <Keyword value 0> <MessageID of keyword 0> <Localized string of keyword 0>

…

<t> <Keyword value t> <MessageID of keyword t> <Localized string of keyword t>

[Event Section]

<Event Record Id 0> <Localized event description string for event 0>

…

<Event Record Id n> <Localized event description string for event n>

3.1.1.8 Queries

Events within log files can be queried through the protocol. The protocol methods for querying the events are **EvtRpcRegisterLogQuery** (as specified in section 3.1.4.12) and **EvtRpcQueryNext** (as specified in section 3.1.4.13). An event query is an expression string that selects events within the log file or files. Because all events in the system have an event XML representation, the expression string can be based on this representation.

The syntax of the filter for a query is specified in sections 2.2.15 and 2.2.16.

3.1.1.9 Subscriptions

Clients can be notified of events occurring on the system through this protocol by using a subscription. Here, a subscriber establishes interest in a set of events selected through an event query. The system delivers the selected events when they occur to the subscriber through the protocol. In this protocol, a client can use the **EvtRpcRegisterRemoteSubscription** method (as specified in section 3.1.4.8) to set up the subscription. There are two types of subscriptions: pull and push subscriptions. The server knows the type of the subscription from the client subscription flag when it is created. The system delivers the events through the methods **EvtRpcRemoteSubscriptionNext** (as specified in section 3.1.4.9) for pull subscription or **EvtRpcRemoteSubscriptionNextAsync** (as specified in section 3.1.4.10) for push subscription.

3.1.1.10 Control Object

The control object is used by the client to cancel a server call that is taking too long to return any result to the client. A control object is an object which is created on the server side when a client registers some heavy operations such as subscription or query. The following example shows a typical workflow such as a subscription or a query:

1. A client tries to register a query job by calling EvtRpcRegisterLogQuery (as specified in section 3.1.4.12).

2. That method returns a control object through the PCONTEXT_HANDLE_OPERATION_CONTROL context handle (as specified in section 3.1.4.7).

3. The client issues the call EvtRpcQueryNext (as specified in section 3.1.4.13) to query the events, and this operation takes a long time before returning a result to the client. If the client wants to abandon the operation, it can call EvtRpcCancel (as specified in section 3.1.4.34) to cancel the query operation by providing the context handle it receives from the server in the first step.

4. Since this protocol describes only one cancel operation for the control object, the control object SHOULD keep the pointer of the operation object, such as a subscription, and a Boolean flag to indicate whether the operation is canceled or not.

For information on how many types of operations can be canceled, see section 3.1.4.6.

3.1.1.11 Context Handles

Sometimes operations such as querying a channels' subscription to new events for that channel can't be finished with one method call from clients. The server maintains the state information for the clients across several method calls to finish a complete workflow. The state information is server-related details and SHOULD NOT be seen by the clients. Thus, the server passes back a handle to

clients and that handle maps the state information in the server. Such handles are called context handles. When clients pass back the context handle, the server knows this handle and knows the state information so that it can serve the subsequent method calls from clients.

This protocol uses the following types of context handles, and does not allow handles of different types to be interchanged:

- PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION: This context handle is dedicated to subscription operation for the client. The client specifies a set of events to the server when setting up the subscription connection. For example, the client can ask to receive all the future events from a specified channel. The client passes that to the server and the server passes a context handle back to the client. Subsequently, the client can keep using this handle for requesting delivery of new events from the server for as long as that subscription connection is established. For this handle, the server logically needs to create an object to stand for this context handle to serve the client requests. In this protocol, this object is called the subscription object. A subscription object is a logical representation of a subscription in server memory. For detail content information on the subscription object, see the processing rules in section 3.1.4. A subscription object SHOULD contain the following information:

  1. HandleType: An integer value that describes the type of the context handle this object stands for. For the subscription object, the server SHOULD always keep this value to be the predefined type value for the type of PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION. For example, if the server decides to use 1 as the type value for PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION, it sets the HandleType to 1.

  2. Channels: An array of client subscribed channels. Each channel contains its name, log file path, and configuration properties. All the channel information can be retrieved from the corresponding channel table entry. As described in section 3.1.1.4, the channels are all registered to the server and kept in the server's channel table.

  3. IsPullType: A Boolean value to indicate pull or push subscription.

  4. Filter: The XPath query expression serving as the filter for delivering the events which meet certain criteria as specified by the filter expression. The filter is the subscription filter.

  5. Positions: An array of numeric LONGLONG values to indicate the next record ID of events for each channel that is to be delivered to the client. This array has the same size as the channels array.

  A subscription object is created by the server and cast to PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION context handle by the server before being passed to the client. The server tracks the client state inside the subscription object so that the server knows how to serve the next requests from the client.

- PCONTEXT_HANDLE_LOG_QUERY: This context handle serves the query operation. To maintain the client state, the server SHOULD create a corresponding log query object. A log query object is a class instance that resides in the server's memory to represent the query object for the client. Inside the query object, the server SHOULD maintain the following data fields:

  1. HandleType: An integer value that describes the type of the context handle this object stands for. For the query object, the server SHOULD always set this value to the predefined type value for the type of PCONTEXT_HANDLE_LOG_QUERY. For example, if the server uses 2 as the type value for PCONTEXT_HANDLE_LOG_QUERY, it sets the HandleType to 2.

  2. The channel path: The channel name or the log file name.

  3. The query filter: The XPATH query from the client input.

  4. Position: The numeric value served as a cursor to indicate where the next return event SHOULD be. The cursor SHOULD be set to 0 initially and be updated each time the client calls

the **EvtRpcQuerySeek** (as specified in section 3.1.4.14) or **EvtRpcQueryNext** (as specified in section 3.1.4.13).

The log query object is created by the server and cast into the PCONTEXT_HANDLE_LOG_QUERY as a handle. When the client passes back the context handle, the server can cast it back to a log query object and all the internal state information is then retrieved by the server.

- PCONTEXT_HANDLE_LOG_HANDLE: This context handle serves all the operations related to channels or logs. To do this, the server SHOULD maintain a log object mapped for this handle. The object SHOULD contain following fields:

  1. HandleType: An integer value that describes the type of the context handle this object stands for. For the query object, the server SHOULD always set this value to the predefined type value for the type of PCONTEXT_HANDLE_LOG_HANDLE. For example, if the server uses 3 as the type value for PCONTEXT_HANDLE_LOG_HANDLE, it sets the HandleType to 3.

  2. LogType: A numeric integer type value that indicates the log information object is a live channel or a backup event log.

  3. Channel: A void* pointer. When the object is a live channel, the pointer points to the channel entry in the server's channel table. When it is a backup event log, the pointer is the handle of the opened event log file.

  The object is created by the server and cast to the PCONTEXT_HANDLE_LOG_HANDLE. When the client returns the handle, the server has all the internal state information by casting it back to the log object.

- PCONTEXT_HANDLE_PUBLISHER_METADATA: The state includes the identity of the publisher as well as the locale to be used. Sometimes the client is interested in information about a publisher that produces an event. The client sends the server the publisher's name to monitor and thus retrieves information on that publisher by issuing subsequent method calls. The server MUST open the publisher's information when the client sends the publisher's name and maintain that opened information so that subsequent methods from the client can pass back publisher information. To do this, the server maintains a **publisher metadata** object. The publisher metadata object is a logical representation of publisher metadata (as specified in section 3.1.1.14). The publisher metadata object SHOULD contain the following data fields:

  1. HandleType: An integer value that describes the type of the context handle this object stands for. For the query object, the server SHOULD always set this value to the predefined type value for the type of PCONTEXT_HANDLE_PUBLISHER_METADATA. For example, if the server uses 4 as the type value for PCONTEXT_HANDLE_PUBLISHER_METADATA, it sets the HandleType to 4.

  2. ResourceFile: A Unicode string for the publisher resource file name (as specified in 3.1.1.14).

  3. MessageFile: A Unicode string for the message file name (as specified in 3.1.1.14).

  4. ParameterFile: A Unicode string for the parameter file name (as specified in 3.1.1.14).

  5. Locale: An unsigned short value for the requested locale.

  6. ResourceFileHandle: The opened resource file handle so that the server can read the file.

  The publisher metadata object is created by the server and cast into the PCONTEXT_HANDLE_PUBLISHER_METADATA handle type. When the client passes the handle, the server can cast it back to the publisher metadata object.

- PCONTEXT_HANDLE_EVENT_METADATA_ENUM: The state includes the identity of the publisher as well as the position of the enumeration. As events provider, the publisher MUST save its events in a predefined XML template (see [MSDN-ProvEvts] for information on providing events). This

template is the **event metadata**. When clients want to enumerate the metadata of an event from a publisher, they MUST get the PCONTEXT_HANDLE_EVENT_METADATA_ENUM context handle. The server MUST maintain the client enumeration status and the publisher identity to complete this task. To maintain that state, the server maintains an event metadata object. The event metadata object is a logical representation of event metadata (as specified in section 3.1.1.14). The event metadata object SHOULD contain the following data fields:

1. HandleType: An integer value that describes the type of the context handle this object stands for. For the query object, the server SHOULD always set this value to the predefined type value for the type of PCONTEXT_HANDLE_EVENT_METADATA_ENUM. For example, if the server uses 5 as the type value for PCONTEXT_HANDLE_EVENT_METADATA_ENUM, it sets the HandleType to 5.

2. EventsMetaData: A memory buffer that holds the data content of the events information section of a publisher resource file (as specified in 3.1.1.14).

3. Enumerator: A numeric integer value serving as a cursor to track the position in the event metadata section.

The event metadata object SHOULD contain the following data fields: the events information section of a publisher resource plus a cursor (numeric value) to track the position in the event metadata section. The event metadata object is created by the server and cast into the PCONTEXT_HANDLE_EVENT_METADATA handle type. When the client passes the handle, the server can cast it back to the event metadata object.

- PCONTEXT_HANDLE_OPERATION_CONTROL: The state includes the method calls that can be controlled by the handle. Sometimes, the client sends a method that has a long run time on the server. The client can cancel such calls that take too long to return. The client can use the PCONTEXT_HANDLE_OPERATION_CONTROL context handle to control whether the server keeps serving the call or cancels it. The server uses this context handle to maintain which operation it is serving any given client so that calls from that client are acted on. The server SHOULD maintain a control object (as specified in section 3.1.1.10) for this handle. The control object SHOULD contain the following data fields:

  1. HandleType: An integer value that describes the type of the context handle this object stands for. For the query object, the server SHOULD always set this value to be the predefined type value for the type of PCONTEXT_HANDLE_OPERATION_CONTROL. For example, if the server uses 6 as the type value for PCONTEXT_HANDLE_OPERATION_CONTROL, it sets the HandleType to 6.

  2. OperationPointer: A pointer that points to a server operation object such as a query object or a subscription object.

  3. Canceled: A Boolean value indicating whether the client has required the server to cancel the operation.

  The operation control object is created by the server and cast into the PCONTEXT_HANDLE_OPERATION_CONTROL handle type. When the client passes the handle, the server can cast it back to the control object.

These context handles are defined as pointers as specified in section 2.2.20.

3.1.1.12 Handle Table

The server MAY have a table of context handles it creates for the client. When the client passes a handle to the server and the server cannot find the handle in the table, it is designated an invalid handle.<4>

3.1.1.13 Localized String Table

The server MUST have a table of localized strings for each publisher and a default table. A table of localized strings is declared by the publisher in the event manifest (event manifest is specified in [MSDN-EvntManifestSE]). The string table declaration is specified in [MSDN-stringTable].

The declared string table is built into the publisher's language-specific resource file. There is one language-specific resource file for each language. See [MSDN-MUIResrcMgmt] for more information about the language-specific resource file. The language-specific resource files are stored on the server pre-generated. language-specific resource file generation is out of the scope of this protocol.

Since there is one language-specific resource file for each language, there is one string table for each language. The default table is the language-specific resource file for the default language. If the server's default language is English, the default string table is the English string table. If the server's default language is simplified Chinese, the default string table is the simplified Chinese string table.

3.1.1.14 Publisher Resource, Message, and Parameter Files

The server MUST keep the resource files for all the publishers who register themselves to the server. It is the publisher's responsibility to register itself to the server, registering a publisher to the server is not in the scope of this protocol. The publisher resource files are DLL files that are generated by the publishers when they designate events. In order to designate events, the publisher needs to specify the channel, provider, and events information in a manifest. For each event, it also needs to specify the level, opcode, task, keyword, and event description information. All this information is written into an instrumentation manifest file. For information on writing the manifest file, see [MSDN-WAIM]. Next, the information is compiled and saved into the publisher resource file. All publishers MUST have a DLL file with a name like publishername.dlll that saves all the events and channels as well as its own name. This file MUST also store the description strings for those events and channels.

The description strings can be localizable. To make localized sets of strings, all the description strings are moved from the publisher resource file (as specified in section 3.1.1.13) and packed into a localized string table and then moved into a corresponding language-specific resource files resource file for each language (as specified in [MSDN-MUIResrcMgmt]). The publisher resource file only saves a *messageId*, which is used as an index to locate the real string in the language-specific resource file for each description string. Then all the levels descriptions are packed into a level table with the level information and the *messageId*. Similarly, all the opcodes, tasks, keywords, event descriptions, and channels are packed as tables in the language-specific resource file.

The following shows a typical instrumentation part of a publisher resource file:

[Publisher Information]

<Publisher Identifier><MessageId for publisher name string><Publisher helper link string>

[Channel Information]

<Channel Identifier 1><Channel 1 name><MessageId for channel 1 description string>

<Channel Identifier n><Channel n name><MessageId for channel n description string>

[Events information]

<Event 1 Identifier><version><MessageId for event 1 description string>

<Level Value of Event 1><Level Name of Event 1><MessageId for level description string>

<Opcode Value of Event 1><Opcode Name of Event 1><MessageId for opcode description string>

<Task Value of Event 1><Task Name of Event 1><MessageId for task description string>

<Keyword Value of Event 1><Keyword Name of Event 1><MessageId for keyword description string>

<event 1 definition template>

<channel identifier>

<publisher identifier>

…

<Event m Identifier><version><MessageId for event m description string>

<Level Value of Event m><Level Name of Event m><MessageId for level description string>

<Opcode Value of Event m><Opcode Name of Event m><MessageId for opcode description string>

<Task Value of Event m><Task Name of Event m><MessageId for task description string>

<Keyword Value of Event m><Keyword Name of Event m><MessageId for keyword description string>

<event m definition template>

<channel identifier>

<publisher identifier>

In a publisher's language-specific resource file, the real strings are saved and indexed by the *messageId*.

All the information in this section for a publisher is also called publisher metadata, and the part of each event in Events information is called event metadata.

Published resource files are DLL or EXE files, as specified in [PE-COFF]. The format of publisher resource files is outside the scope of this protocol.

By default, the server SHOULD provide a default publisher and that publisher can provide common channels, events, and so forth. The default publisher is built in with the server and does not have to be installed. The format of its resource file is the same as any publisher resource file. Although recommended, the server does not have to provide a default publisher.

The publisher resource file only saves the message Id of any string. The server SHOULD have another file that saves all the real strings. That would be a publisher message file, which takes the following form:

<MessageId 1><The language neutral string 1>

…

<MessageId n><The language-neutral string n>

These language-neutral strings can then be translated into different languages and put into localized string tables (section 3.1.1.13).

The publisher parameter file has the same format as the publisher message file and can take this form:

<Value 1><String 1>

…

<Value n><String n>

The strings in the publisher parameter file cannot be localized. It is used for parameter substitution. For example, if a publisher defines the description string of an event as "The system has found %%2", when the server tries to expand the string with the **EvtRpcMessageRender** method (section 3.1.4.31), it sees %%2 and knows that this part is replaced with a real string from the publisher's parameter file. It then uses 2 as the index and finds the string for the value 2 in the parameter file and replaces %%2 with that string.

3.1.2 Timers

None.

3.1.3 Initialization

The EventLog Remoting Protocol Version 6.0 server MUST be initialized by registering the RPC interface and listening on the **RPC endpoint**, as specified in section 2.1. Then, the server MUST wait for client requests.

3.1.4 Message Processing Events and Sequencing Rules

Because the server MUST make access control decisions as part of responding to EventLog Remoting Protocol Version 6.0 requests, the client MUST authenticate to the server. This is the responsibility of the lower-layer protocol, RPC over TCP/IP (as specified in [C706]). The access control decisions affecting the EventLog Remoting Protocol Version 6.0 are made based on the identity conveyed by this lower-layer protocol.

The following sections first provide an informative overview of the message sequences before giving the prescriptive details of processing for each message.

The following table lists the **IDL** members in opcode order.

Methods in RPC Opnum Order

Method	Description
EvtRpcRegisterRemoteSubscription	Used by a client to create either a push or a pull subscription. Opnum: 0
EvtRpcRemoteSubscriptionNextAsync	Used by a client to request asynchronous delivery of events that are delivered to a subscription. Opnum: 1
EvtRpcRemoteSubscriptionNext	Used for pull subscriptions in which the client polls for events. Opnum: 2
EvtRpcRemoteSubscriptionWaitAsync	Used to enable the client to only poll when results are likely. Opnum: 3
EvtRpcRegisterControllableOperation	Obtains a CONTEXT_HANDLE_OPERATION_CONTROL handle that can be used to cancel other operations. Opnum: 4
EvtRpcRegisterLogQuery	Used to query one or more channels. It can also be used to query a specific file. Opnum: 5
EvtRpcClearLog	Instructs the server to clear a live event log. Opnum: 6

Method	Description
EvtRpcExportLog	Instructs the server to create a backup event log at a specified file name. Opnum: 7
EvtRpcLocalizeExportLog	Used by a client to add localized information to a previously created backup event log. Opnum: 8
EvtRpcMessageRender	Used by a client to get localized descriptive strings for an event. Opnum: 9
EvtRpcMessageRenderDefault	Used by a client to get localized strings for common values of opcodes, tasks, or keywords, as specified in section 3.1.4.31. Opnum: 10
EvtRpcQueryNext	Used by a client to get the next batch of records from a query result set. Opnum: 11
EvtRpcQuerySeek	Used by a client to move a query cursor within a result set. Opnum: 12
EvtRpcClose	Used by a client to close context handles opened by other methods in this protocol. Opnum: 13
EvtRpcCancel	Used by a client to cancel another method. Opnum: 14
EvtRpcAssertConfig	Indicates to the server that publisher or channel configuration has been updated. Opnum: 15
EvtRpcRetractConfig	Indicates to the server that publisher or channel configuration is to be removed. Opnum: 16
EvtRpcOpenLogHandle	Used by a client to get information on a live or backup log. Opnum: 17
EvtRpcGetLogFileInfo	Used by a client to get information on an event log. Opnum: 18
EvtRpcGetChannelList	Used to enumerate the set of available channels. Opnum: 19
EvtRpcGetChannelConfig	Used by a client to get the configuration for a channel. Opnum: 20
EvtRpcPutChannelConfig	Used by a client to update the configuration for a live event log. Opnum: 21
EvtRpcGetPublisherList	Used by a client to get the list of publishers. Opnum: 22
EvtRpcGetPublisherListForChannel	Used by a client to get the list of publishers that write events to a particular live event log. Opnum: 23

Method	Description
EvtRpcGetPublisherMetadata	Used by a client to open a handle to publisher metadata. It also gets some initial information from the metadata. Opnum: 24
EvtRpcGetPublisherResourceMetadata	Used by a client to obtain information from the publisher metadata. Opnum: 25
EvtRpcGetEventMetadataEnum	Used by a client to obtain a handle for enumerating a publisher's event metadata. Opnum: 26
EvtRpcGetNextEventMetadata	Used by a client to get details on a particular possible event, and also returns the next event metadata in the enumeration. Opnum: 27
EvtRpcGetClassicLogDisplayName	Used to obtain a descriptive name of a channel. Opnum: 28

All methods MUST NOT throw exceptions. All return values use the NTSTATUS numbering space (as specified in [MS-ERREF] section 2.3) and, in particular, a value of 0x00000000 indicates success, and any other return value indicates an error. For a mapping of Windows NT operating system status error codes to Win32 error codes, see [MSKB-113996]. All error values MUST<5> be treated the same, unless specified otherwise.

Within the sections that follow this one, methods are presented in the order typically implemented to accomplish the following operations:

- Subscription
- Queries
- Log Maintenance
- Configuration and Metadata
- Message Rendering
- Miscellaneous Operations

3.1.4.1 Subscription Sequencing

Subscriptions can be either pull or push model. The pull model is essentially a polling model in which the client requests new events; in the push mode, the server delivers events as they occur.

In all models, the subscription starts with a client application (that is, the higher layer above the protocol client) calling the EvtRpcRegisterRemoteSubscription (section 3.1.4.8) method to get a CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle. The subscription ends when that handle is closed by using the EvtRpcClose (section 3.1.4.33) method.

In between these calls, the two models vary. All methods used in the two models use the CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle.

In the pull model, the client loops by using the EvtRpcRemoteSubscriptionNext (section 3.1.4.10) method to get events. Optionally, the client can use the EvtRpcRemoteSubscriptionWaitAsync (section 3.1.4.11) method to delay calling the EvtRpcRemoteSubscriptionNext (section 3.1.4.10) method until events are ready. The server

completes the EvtRpcRemoteSubscriptionWaitAsync (section 3.1.4.11) method call when new events are ready.

In the push model, the client loops by using the EvtRpcRemoteSubscriptionNextAsync (section 3.1.4.9) method to get events. The call MUST be completed by the server when a new event is ready.

Note that there is also a CONTEXT_HANDLE_OPERATION_CONTROL handle returned by the EvtRpcRegisterRemoteSubscription (section 3.1.4.8) method. The sequencing and use of these handles are specified in section 3.1.4.6.

The application ends the subscription by passing the CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle to the EvtRpcClose (section 3.1.4.33) method.

3.1.4.2 Query Sequencing

Queries begin with a client application calling the EvtRpcRegisterLogQuery (section 3.1.4.12) method, which returns a handle of type CONTEXT_HANDLE_LOG_QUERY.

The client application can then use the handle for subsequent calls to the EvtRpcQueryNext (section 3.1.4.13) method or the EvtRpcQuerySeek (section 3.1.4.14) method.

The application then closes the handle at the end of the query using EvtRpcClose.

Note that there is also a CONTEXT_HANDLE_OPERATION_CONTROL handle returned by EvtRpcRegisterLogQuery. The sequencing and use of these handles are specified in section 3.1.4.6.

3.1.4.3 Log Information Sequencing

To get information on a log, a client application calls the EvtRpcOpenLogHandle (section 3.1.4.19) method first to get a handle of type CONTEXT_HANDLE_LOG_HANDLE.

The application can then use the handle for subsequent calls to the EvtRpcGetLogFileInfo (section 3.1.4.15) method.

Finally, the application closes the handle by using the EvtRpcClose (section 3.1.4.33) method.

3.1.4.4 Publisher Metadata Sequencing

To get information on a publisher, a client application calls the EvtRpcGetPublisherMetadata (section 3.1.4.25) method to get a handle of type CONTEXT_HANDLE_PUBLISHER_METADATA.

The application can then use the handle for subsequent calls to the EvtRpcMessageRender (section 3.1.4.31), EvtRpcGetPublisherResourceMetadata (section 3.1.4.26), and EvtRpcGetEventMetadataEnum (Opnum 26) methods.

Finally, the application closes the handle by using the EvtRpcClose (section 3.1.4.33) method.

3.1.4.5 Event Metadata Enumerator Sequencing

To enumerate information on a publisher's events, a client application calls the EvtRpcGetEventMetadataEnum (Opnum 26) method to get a handle of type CONTEXT_HANDLE_EVENT_METADATA_ENUM.

The application can then use the handle for subsequent calls to the EvtRpcGetNextEventMetadata (section 3.1.4.28) method.

Finally, the application closes the handle by using the EvtRpcClose (section 3.1.4.33) method.

3.1.4.6 Cancellation Sequencing

A client application can use CONTEXT_HANDLE_OPERATION_CONTROL to cancel a method by passing CONTEXT_HANDLE_OPERATION_CONTROL to the EvtRpcCancel (section 3.1.4.34) method. The EvtRpcClose (section 3.1.4.33) method is then used when the application no longer needs the handle.

3.1.4.6.1 Canceling Subscriptions

The CONTEXT_HANDLE_OPERATION_CONTROL handle is obtained at the same time a CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle is obtained by calling the EvtRpcRegisterRemoteSubscription (Opnum 0) (section 3.1.4.8) method. Any calls to the EvtRpcRemoteSubscriptionNext (Opnum 2) (section 3.1.4.10), EvtRpcRemoteSubscriptionNextAsync (Opnum 1) (section 3.1.4.9), and EvtRpcRemoteSubscriptionWaitAsync (Opnum 3) (section 3.1.4.11) methods using the CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle can be canceled by using the CONTEXT_HANDLE_OPERATION_CONTROL handle in a call to the EvtRpcCancel (section 3.1.4.34) method.

3.1.4.6.2 Canceling Queries

The CONTEXT_HANDLE_OPERATION_CONTROL handle is obtained at the same time a CONTEXT_HANDLE_LOG_QUERY handle is obtained by calling the EvtRpcRegisterLogQuery (Opnum 5) (section 3.1.4.12) method. Any calls to the EvtRpcQueryNext (Opnum 11) method or the EvtRpcQuerySeek (Opnum 12) (section 3.1.4.14) method by using the CONTEXT_HANDLE_LOG_QUERY handle can be canceled by using the CONTEXT_HANDLE_OPERATION_CONTROL handle in a call to the EvtRpcCancel (section 3.1.4.34) method.

3.1.4.6.3 Canceling Clear or Export Methods

Any calls to the EvtRpcClearLog (Opnum 6) (section 3.1.4.16), EvtRpcExportLog (Opnum 7) (section 3.1.4.17), and EvtRpcLocalizeExportLog (Opnum 8) (section 3.1.4.18) methods can be canceled by using a CONTEXT_HANDLE_OPERATION_CONTROL handle. Normally, the CONTEXT_HANDLE_OPERATION_CONTROL handle used for these functions is obtained via the EvtRpcRegisterControllableOperation (Opnum 4) (section 3.1.4.35) method.

There is a single type of CONTEXT_HANDLE_OPERATION_CONTROL handle. A handle obtained by the EvtRpcRegisterRemoteSubscription (Opnum 0) (section 3.1.4.8) method or the EvtRpcRegisterLogQuery (Opnum 5) (section 3.1.4.12) method can be used to cancel the EvtRpcClearLog (Opnum 6) (section 3.1.4.16), EvtRpcExportLog (Opnum 7) (section 3.1.4.17), and EvtRpcLocalizeExportLog (Opnum 8) (section 3.1.4.18) methods. There is no restriction on how many methods a CONTEXT_HANDLE_OPERATION_CONTROL handle can control. One handle can be used to cancel any number of calls.

3.1.4.7 BinXml

The event information returned by the query and subscription methods is in a binary format named BinXml. BinXml is a token representation of text XML 1.0, as specified in [XML10].

Here BinXml encodes an XML document so that the original XML text can be correctly reproduced from the encoding. There is no requirement for a server to use or understand the text XML. The protocol can be implemented end to end by treating BinXml as a method to transmit name-value pairs, instead of as an encoding of XML. However, after the data has been received, it is common for third-party applications to convert from binary XML to text XML independent of the protocol. Therefore, for informative purposes only, an overview of the relationship is provided.

Note that this translation is not required by either the client or the server in this protocol.

What follows is a greatly simplified example of a fragment of text XML encoding in binary XML.

Text	Binary
<SomeEvent>	01 SomeEvent 02
<PropA> 99 </PropA>	01 PropA 02 05 "99" 04
<PropB> 101 </PropB>	01 PropB 02 05 "101" 04
</SomeEvent>	04 00

The binary bytes in the preceding example have the following meaning.

```
00 - eof
01 - open start tag
02 - close start tag
04 - end tag
05 - value text
```

BinXml also includes more information that allows for fast navigation of the XML. For example, lengths of elements and attribute lists allow the user to jump forward in the BinXml stream. Another example is that BinXml encoding of Names includes length and hash values that allow for fast comparisons of the XML names.

3.1.4.7.1 BinXml Templates

BinXml encoding supports a way to use a template of a BinXml fragment and apply it to a set of values. A BinXml template describes the format and contents of an event independent of the values contained in a specific instance of the event being described. It contains property names and placeholders for the event properties.

The primary advantage of this is that the values (set of data) can remain in native form and only need to be converted to text if the BinXml encoding is actually rendered into XML text.

A BinXml encoding of an XML fragment that uses templates and a set of substitution values is referred to as a Template Instance. A Template Definition is a BinXml fragment that contains substitution tokens, and the Template Instance Data refers to the set of values.

Continuing the example from BinXml (section 3.1.4.7) with a possible sample Template Definition, note the following:

- This example uses two substitution tokens, %1 and %2, that are replaced by specific event values at rendering time.
- These tokens map to substitution identifiers in the BinXML template.
- The substitution identifiers are 0-based, whereas the substitution tokens are 1-based.
- The Text column of the following table shows the XML representation of the various fields.
- The Binary column of the following table shows the binary representation of those fields encoded in BinXML.

Text	Binary
<SomeEvent>	01 SomeEvent 02
<PropA> %1 </PropA>	01 PropA 02 05 0D 00 04

Text	Binary
<PropB> %3 </PropB>	01 PropB 02 05 0D 01 04
</SomeEvent>	04 00

Where the substitution token is 0D, and is followed by a substitution identifier (00 or 01 in the example).

This template definition can be combined with raw UINT8 values { 0x63, 0x65 } to form a Template Instance such as the following example. The ordering of the values is significant: the first value encountered maps to the first substitution identifier; the second value maps to the second substitution identifier, and so on. In the following example, the value 0x63 replaces the identifier 00 at rendering time, and the 0x65 replaces identifier 01.

Text	Binary
	0C
<SomeEvent>	01 SomeEvent 02
<PropA> %1 </PropA>	01 PropA 02 05 0D 00 04
<PropB> %2 </PropB>	01 PropB 02 05 0D 01 04
</SomeEvent>	04 00
	0 01 04 01 04
	63 65 00

Note The beginning Template Instance token (0x0C) and the trailing end of fragment or document token (EOFToken, 0x00) for the Template Instance immediately following the Template Definition is information about the type and length of the values that make up the Template Instance data. This is called the Value Spec of the Template Instance. In this example, there are 2 values, each of UINT8 integer type (04) and each of length 1.

If the BinXml in the preceding example is rendered as XML text, it looks identical to the first example, as follows.

```
<SomeEvent>
<PropA> 99 </PropA>
<PropB> 101 </PropB>
</SomeEvent>
```

Substitutions can occur in attribute values as well as any other place where XML character data is allowed (for example, in element content). Within these regions, there are no restrictions on the number of substitutions that can exist.

3.1.4.7.2 Optional Substitutions

Another feature of BinXml templates is that substitutions can be specified such that the enclosing element or attribute MUST be omitted from rendered XML text (or other processing) if the value identified by the substitution is NULL in the Template Instance data. If this type of rendering from the BinXml is wanted, the substitution needs to be specified by using an optional substitution token. The optional substitution token is 0x0E, as compared to the normal substitution token 0x0D.

The server MAY determine whether to use the optional substitution token based on the event definition.<6>

The following table contains an example where %1 and %2 represent the optional substitution tokens.

Text	Binary
	0C
<SomeEvent>	01 SomeEvent 02
<PropA> %1 </PropA>	01 PropA 02 05 0E 00 04
<PropB> %2 </PropB>	01 PropB 02 05 0E 01 04
</SomeEvent>	04 00
	02 00 00 01 04
	65 00

This tells any processor of the encoded BinXml to use the following XML representation.

```
<SomeEvent>
<PropB> 101 </PropB>
</SomeEvent>
```

Note The preceding Value Spec for the optional element PropA specifies that the substitution value is NULL (see Type System (section 3.1.4.7.3)), and this is how the BinXml processor knows to omit this element.

The optional substitution applies only to the element or attribute immediately enclosing it.

Note If an element contains an optional substitution, and that substitution value is NULL, the element cannot appear in rendered XML, even if that element has attributes containing content, other (nonNULL) substitution values, and so on.

3.1.4.7.3 Type System

Each value (in BinXml encoding) of templates has an accompanying type. Likewise, each value has an accompanying byte length. This is redundant for fixed size types, but is necessary for variable-length types such as strings and binary large objects (BLOBs).

Each BinXml type has a canonical XML representation. This is the format used to represent the value when the BinXml is rendered as XML text. The following table gives the meaning of each type and also lists its canonical XML representation by association with XSD types. An XS: prefix specifies the XML Schema namespace (as specified in [XMLSCHEMA1.1/2:2008]), and an EVT: prefix specifies types defined in the event.xsd (as specified in [MSDN-EVENTS]).

The binary encoding of all number types MUST be little-endian. Additionally, no alignment is assumed in the binary encoding for any of these types.

This table can be used to convert between BinXml and canonical XML. That is, if an application converts BinXml to text, the binary form on the left is replaced with the type on the right, where the type column corresponds to a field in the ABNF, as specified in section 2.2.12.

BinXml type	Meaning	Canonical XML representation
NullType	No value.	"" (Empty String)
StringType	A sequence of [UNICODE] characters. Not assumed to be null terminated. The string length is derived from the byte length accompanying value.	xs:String

BinXml type	Meaning	Canonical XML representation
AnsiStringType	A sequence of ANSI characters. Not assumed to be null terminated. The string length is derived from the Byte length accompanying value.	xs:String
Int8Type	A signed 8-bit integer.	xs:byte
UInt8Type	An unsigned 8-bit integer.	xs:unsignedByte
Int16Type	A signed 16-bit integer.	xs:short
UInt16Type	An unsigned 16-bit integer.	xs:unsignedShort
Int32Type	A signed 32-bit integer.	xs:int
UInt32Type	An unsigned 32-bit integer.	xs:unsignedInt
HexInt32Type	An unsigned 32-bit integer.	evt:hexInt32
Int64Type	A signed 64-bit integer.	xs:long
UInt64Type	An unsigned 64-bit integer.	xs:unsignedLong
HexInt64Type	An unsigned 64-bit integer.	evt:hexInt64
Real32Type	An IEEE 4-byte floating point number.	xs:float
Real64Type	An IEEE 8-byte floating point number.	xs:double
BoolType	An 8-bit integer that MUST be 0x00 or 0x01 (mapping to true or false, respectively).	xs:Boolean
BinaryType	A variable size sequence of bytes.	xs:hexBinary
GuidType	A 128-bit UUID, as specified in [C706], for example, {2d4d81d2-94bd-4667-a2af-2343f9d83462}. The canonical form in XML contains braces.	evt:GUID
SizeTType	A 32-bit unsigned integer, if the server is a 32-bit platform; or a 64-bit unsigned integer, if the server is a 64-bit platform.	evt:hexInt32 or Evt::hexInt64
FileTimeType	An 8-byte FILETIME structure, as specified in [MS-DTYP] Appendix A.	xs:dateTime
SysTimeType	A 16-byte SYSTEMTIME structure, as specified [MS-DTYP].	xs:dateTime
SidType	A binary representation of the SID, as specified in [MS-DTYP]. While the structure has variable size, its length is contained within the data itself. The canonical form is the output of the Security Descriptor Definition Language (SDDL) string SID representation, as specified in [MS-DTYP].	xs:string
BinXmlType	Specified in section 3.1.4.7.4.	

3.1.4.7.4 BinXml Type

The BinXml type MUST be used for values that are themselves BinXml encoding of XML fragments or TemplateInstances. This allows embedding of TemplateInstances.

The byte length for the value specifies the length of the BinXml fragment, up to and including its EOF token.

This type MUST only be used when substituting into element content. For example, given the following template instance.

Text	Binary
	0C
<InnerTemplate>	01 InnerTemplate 02
<PropA> %1 </PropA>	01 PropA 02 05 0D 00 04
<PropB> %2 </PropB>	01 PropB 02 05 0D 01 04
</InnerTemplate>	04 00
	02 01 04 01 04
	63 65 00

And the following outer template definition.

Text	Binary
<OuterTemplate>	01 OuterTemplate 02
<PropA> %1 </PropA>	01 PropA 02 05 0D 00 04
<PropB> %2 </PropB>	01 PropB 02 05 0D 01 04
</OuterTemplate>	

If the set of values for the outer template instance is the BinXml for InnerTemplate TemplateInstance, and the UINT8 value is 0x67, the resultant BinXml is the following.

```
0C
01 OuterTemplate 02
01 PropA 02 05 0D 00 04
01 PropB 02 05 0D 01 04
04 00
02 30 21 01 04           <- value spec for Outer Template
0C
01 InnerTemplate 02
01 PropA 02 05 0D 00 04
01 PropB 02 05 0D 01 04
04 00
02 01 04 01 04           <- value spec for inner template
63 65 00
67 00
```

Note the value spec for the Inner Template. The template is 0x30 long, and is of type 0x21 ("BinXmlType"). It is followed by one UINT8 value.

3.1.4.7.5 Array Types

In addition to the base types, arrays of most base types can be specified in BinXml encoding. The only basic types that are not allowed are binary, non-null-terminated AnsiStringType string, non-null-terminated StringType string, and BinXml.

The array itself is considered a single value of the set of values that make up the Template Instance. As with all values, there is an accompanying type and a byte length. Elements of an array MUST all be of the same type.

The binary representation of the array MUST be the serialized representation of each element of the array.

The byte length MUST be used to derive the number of elements in the array. This is trivial for fixed size types. Arrays of variable length types are supported, but only if the length of each element in the array can be derived for the data itself. The only variable length types that can be used in arrays are:

- Null-terminated ANSI strings
- Null-terminated [UNICODE] strings
- SIDs

Consider the original template example. If the set of values is { [97,99], 101} (where the first value is an array of two elements of type UINT8, and the second value is of type UINT8), the resultant BinXml is shown in the following table.

Text	0C
<SomeEvent>	01 SomeEvent 02
<PropA> %1 </PropA>	01 PropA 02 05 0D 00 04
<PropB> %2 </PropB>	01 PropB 02 05 0D 01 04
</SomeEvent>	04 00
	02 02 84 01 04
61 63 65 00	

And the resultant XML text representation of this encoding is the following.

```
<SomeEvent>
<PropA> 97 </PropA>
<PropA> 99 </PropA>
<PropB> 101 </PropB>
</SomeEvent>
```

3.1.4.7.6 Prescriptive Details

The server MUST return all event information encoded in BinXml format according to the BinXml ABNF.

Additionally, the server MUST organize the data encoded by the BinXml in such a way that if the BinXml is transformed to XML text, this XML text is valid according to the Event.xsd Schema (as specified in [MSDN-EVENTS]).

The Type System table (for more information, see section 3.1.4.7.3) MUST be used to map Substitution values onto XSD Schema types (as specified in [MSDN-EVTSST] and [MSDN-EVTSCT]).

3.1.4.8 EvtRpcRegisterRemoteSubscription (Opnum 0)

The EvtRpcRegisterRemoteSubscription (Opnum 0) method is used by a client to create either a push or a pull subscription. In push subscriptions, the server calls the client when new events are ready. In pull subscriptions, the client polls the server for new events. Subscriptions can be to either a single channel and its associated log, or to multiple channels and their logs.

A client can use bookmarks to ensure a reliable subscription even if the client is not continuously connected. A client can create a bookmark locally based on the contents of an event that the client has processed. If the client disconnects and later reconnects, it can use the bookmark to pick up where it left off. For information on bookmarks, see section 2.2.14.

```
error_status_t EvtRpcRegisterRemoteSubscription(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in, range(1, MAX_RPC_QUERY_LENGTH), string]
    LPCWSTR query,
  [in, unique, range(0, MAX_RPC_BOOKMARK_LENGTH), string]
    LPCWSTR bookmarkXml,
  [in] DWORD flags,
  [out, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION* handle,
  [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* control,
  [out] DWORD* queryChannelInfoSize,
  [out, size_is(, *queryChannelInfoSize), range(0, MAX_RPC_QUERY_CHANNEL_SIZE)]
    EvtRpcQueryChannelInfo** queryChannelInfo,
  [out] RpcInfo* error
);
```

binding: An RPC binding handle as specified in section 2.2.21.

channelPath: A pointer to a string that contains a channel name or is a null pointer. In the case of a null pointer, the *query* field indicates the channels to which the subscription applies.

query: A pointer to a string that contains a query that specifies events of interest to the application. The pointer MUST be either an XPath filter, as specified in section 2.2.15, or a query as specified in section 2.2.16.

bookmarkXml: Either NULL or a pointer to a string that contains a bookmark indicating the last event that the client processed during a previous subscription. The server MUST ignore the bookmarkXML parameter unless the flags field has the bit 0x00000003 set.

flags: Flags that determine the behavior of the query.

Value	Meaning
EvtSubscribeToFutureEvents 0x00000001	Get events starting from the present time.
EvtSubscribeStartAtOldestRecord 0x00000002	Get all events from the logs, and any future events.
EvtSubscribeStartAfterBookmark 0x00000003	Get all events starting after the event indicated by the bookmark.

The following bits control other aspects of the subscription. These bits are set independently of the flags defined for the lower two bits, and independently of each other.

Value	Meaning
EvtSubscribeTolerateQueryErrors 0x00001000	The server does not fail the function as long as there is one valid channel.
EvtSubscribeStrict 0x00010000	Fail if any events are missed for reasons such as log clearing.

Value	Meaning
EvtSubscribePull 0x10000000	Subscription is going to be a pull subscription. A pull subscription requires the client to call the **EvtRpcRemoteSubscriptionNext** (as specified in section 3.1.4.10) method to fetch the subscribed events. If this flag is not set, the subscription is a push subscription. A push subscription requires the client to call the **EvtRpcRemoteSubscriptionNextAsync** (as specified in section 3.1.4.9) to receive notifications from the server when the subscribed events arrive.

handle: A context handle for the subscription. This parameter is an RPC context handle, as specified in [C706], Context Handles.

control: A context handle for the subscription. This parameter is an RPC context handle, as specified in [C706], Context Handles.

queryChannelInfoSize: A pointer to a 32-bit unsigned integer that contains the number of EvtRpcQueryChannelInfo structures returned in *queryChannelInfo*.

queryChannelInfo: A pointer to an array of EvtRpcQueryChannelInfo (section 2.2.11) structures that indicate the status of each channel in the subscription.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to nonzero values if the error is related to parsing the query. If the method succeeds, the server MUST set all of the values in the structure to 0.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST fail the method if any of the following conditions occur:

- The *flags* parameter specifies that the *bookmarkXML* parameter is used; and the *bookmarkXML* parameter is NULL or does not contain a valid bookmark. For more information, see section 2.2.14. The server SHOULD return ERROR_INVALID_PARAMETER (0x00000057) in this case.<7>
- The *channelPath* argument specifies a channel that does not exist. The server MAY return ERROR_EVT_INVALID_CHANNEL_PATH (0x00003A98).<8>
- The *query* parameter is syntactically incorrect. The server SHOULD return ERROR_EVT_INVALID_QUERY (0x00003A99) in this case. The *query* argument MUST be either of the following:
  - A simple XPath

    For information on the specification of the protocol's support of XPath, see section 2.2.15.
  - A query

    For more information, see section 2.2.16). The server MUST verify the validity of any channel names specified in the query, and any invalid channels MUST be returned via the *QueryChannelInfo* parameter.

    If there is at least one invalid channel path and the 0x00010000 bit (**EvtSubscribeStrict**) is set in the *flags* parameter, the server MUST fail the method with the error ERROR_EVT_INVALID_CHANNEL_PATH (0x00003A98).

    If the client specifies the 0x00000003 bit (**EvtSubscribeStartAfterBookmark**) and the 0x00010000 bit (**EvtSubscribeStrict**) is set in the flags parameter, the server MUST fail the

method with the error **ERROR_EVT_INVALID_QUERY** (0x00003A99) if the events position specified by the bookmark is missing in the event channel.

If the 0x00001000 bit (**EvtSubscribeTolerateQueryErrors**) is set, the function SHOULD NOT fail if the channelPath is valid although the query parameter is invalid. The method also SHOULD NOT fail if the query parameter is a structure query (as specified in section 2.2.16) that contains at least one valid channel.

If both **EvtSubscribeStrict** and **EvtSubscribeTolerateQueryErrors** are specified in the flags parameter, the server ignores the **EvtSubscribeTolerateQueryErrors** and only uses the **EvtSubscribeStrict** flag.

Next, the server MUST verify that the caller has read access to the files, and MUST fail the method if the caller does not have read access with the error code ERROR_ACCESS_DENIED (0x00000005).

- If *bookmarkXML* is non-NULL and the **EvtSubscribeStartAfterBookmark** flag is set, and the log has been cleared or rolled over since the bookmark was obtained, the server MUST fail the method and return ERROR_EVT_QUERY_RESULT_STALE (0x00003AA3).
- The server SHOULD fail the method if both the path and query parameters are NULL.<9>

The server SHOULD fail the method with the error code ERROR_INVALID_PARAMETER (0x00000057) if the flags parameter is 0 or does not contain one of the following values:

- 0x00000001 (**EvtSubscribeToFutureEvents**)
- 0x00000002 (**EvtSubscribeStartAtOldestRecord**)
- 0x00000003 (**EvtSubscribeStartAfterBookmark**)

If the above checks all succeed, the server MUST attempt to do the following:

- Create a CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle to the subscription.
- Create a CONTEXT_HANDLE_OPERATION_CONTROL handle.

The server MUST set the name element to the name of the *queryChannelInfo* parameter to the name of the channels in question, and the status element to the status for that particular channel. For example, if the query contains the "Application" channel, the server MUST return an EvtRpcQueryChannelInfo struct with the name set to "Application"; if the query against that channel was successfully registered, the server MUST set the status element to ERROR_SUCCESS (0x00000000); if the query for that channel failed, the server MUST set the status element to an NTSTATUS error code indicating the reason for failure.

When creating the CONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle, the server SHOULD create a subscription object. A subscription object is a class instance that logically stands for a subscription in the server memory as specified in section 3.1.1.11. The server SHOULD set the positions in the subscription object where the events SHOULD start in the channels based on the bookmark value if the bookmark is provided. If the bookmark is not provided, the position values are set either to be the beginning of channels if the flags contains **EvtSubscribeStartAtOldestRecord**, or set to be the latest position values of channels if the flags contain **EvtSubscribeToFutureEvents**. If the client specifies the EvtSubscribePull bit in the *flags* parameter, the server SHOULD set the IsPullType field in the subscription object to be true, otherwise the value SHOULD be false. The channel array in the subscription object SHOULD be set as all the client subscribed channels and the **subscription filter** is set to be the XPath query expression from the parameter.

When creating the CONTEXT_HANDLE_OPERATION_CONTROL handle, the server SHOULD create an operation control object. The server SHOULD set the operational pointer of the control object to be the pointer of the subscription object that the server creates so that it can perform control operations on

that object. The server SHOULD also set the canceled field in the control object initially to false. If the client waits too long for the subscription, it can use the EvtRpcCancel method (as specified in section 3.1.4.34) to cancel the subscription. Since the operation control object contains the subscription pointer, it can request the subscription to stop on the server side.

The server SHOULD only fail the creation of handles in the case of not enough memory and return ERROR_OUTOFMEMORY (0x0000000E). The server SHOULD add the newly created handles to its handle table in order to track them.

The server MUST return a value that indicates success or failure for this operation.

3.1.4.9 EvtRpcRemoteSubscriptionNextAsync (Opnum 1)

The EvtRpcRemoteSubscriptionNextAsync (Opnum 1) method is used by a client to request asynchronous delivery of events that are delivered to a subscription.

```
error_status_t EvtRpcRemoteSubscriptionNextAsync(
  [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle,
  [in] DWORD numRequestedRecords,
  [in] DWORD flags,
  [out] DWORD* numActualRecords,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataIndices,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataSizes,
  [out] DWORD* resultBufferSize,
  [out, size_is(, *resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
    BYTE** resultBuffer
);
```

handle: A handle to the subscription. This parameter is an RPC context handle, as specified in [C706], Context Handles.

numRequestedRecords: A 32-bit unsigned integer that contains the number of events to return.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<10>

numActualRecords: A pointer to a 32-bit unsigned integer that contains the value that, on success, MUST be set to the number of events retrieved. This might be used, for example, if the method times out without receiving the full number of events specified in *numRequestedRecords*.

eventDataIndices: A pointer to an array of 32-bit unsigned integers that contain the offsets for the event. An event's offset is its position relative to the start of *resultBuffer*.

eventDataSizes: A pointer to an array of 32-bit unsigned integers that contain the event sizes in bytes.

resultBufferSize: A pointer to a 32-bit unsigned integer that contains the number of bytes of data returned in *resultBuffer*.

resultBuffer: A pointer to a byte-array that contains the result set of one or more events. The events MUST be in binary XML format, as specified in section 2.2.17.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD fail the operation if the handle is not valid. The server SHOULD save the handle value it creates and returns to the client via the handle parameter in the

EvtRpcRegisterRemoteSubscription method (as specified in section 3.1.4.8) in its handle table (as specified in section 3.1.1.12) and compare it with the handle passed here to perform the check.<11> The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid.

After the server validates the handle, it casts the handle value to the subscription object. The server then MUST check whether the subscription object is a push subscription. Since the subscription object contains the type of subscription, the server checks its type and SHOULD fail the method if it is not a push type subscription with the error ERROR_INVALID_OPERATION(0x000010DD).

If the preceding check succeeds, the server MUST determine whether there are any events the client has not received that pass the subscription filters. The subscription filters are the XPath queries that the client specifies in the query parameter in the **EvtRpcRegisterRemoteSubscription** method (as specified in section 3.1.4.8). For information on how the server applies the filter, see [MSDN-CONSUMEVTS]. The server MUST wait until there is at least one event the client has not received before completing this call. Once there is at least one event, the server MUST return the event or events, and then update its subscription object state to keep track of what events have been delivered to the subscription. As specified in section 3.1.4.8, the subscription object contains the position where the events start in the channels, once the new event is delivered to the client, the server is able to update the position value it saves in the subscription object so that it can perform the tracking task of the events delivery. The server SHOULD track the new events generation from any of the registered publishers in order for it to deliver the coming events to the client in a timely manner. See [MSDN-ProcessTrace] for a suggested implementation.

The server returns the result in the five output parameters: *numActualRecords*, *eventDataIndices*, *eventDataSizes*, *resultBufferSize*, and *resultBuffer*. On successful return, the *numActualRecords* contains the number of events in the *resultBuffer*. All the returned events are in BinXML format and they are packed as one binary blob in the *resultBuffer*. The total size of all these events are marked by *resultBufferSize*. Since all the events are packed together, there is a need to identify where the separator is for each event in the result. To do this, the server fills two arrays: eventDataIndices and eventDataSizes. Both arrays contain the *numActualRecords* of elements. For the eventDataIndices array, each array element is a 32-bit value which is the start position of each event in the *resultBuffer*. For the eventDataSizes array, each element is a 32-bit value which is the size of every event.

The server SHOULD be notified by the underlying network that the connection is lost from the client if the client abnormally terminates the connection. The server abandons its operation for the client in such a case. The server releases the subscription object it creates and free all associated resources. The associated resources are described in EvtRpcRegisterRemoteSubscription (Opnum 0) (section 3.1.4.8).

The server MUST return a value that indicates success or failure for this operation.

3.1.4.10 EvtRpcRemoteSubscriptionNext (Opnum 2)

This EvtRpcRemoteSubscriptionNext (Opnum 2) method is a synchronous request for events that have been delivered to a subscription. This method is only used for pull subscriptions in which the client polls for events. The EvtRpcRemoteSubscriptionWaitAsync (section 3.1.4.11) method can be used along with this method to minimize the frequency of polling.

```
error_status_t EvtRpcRemoteSubscriptionNext(
  [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle,
  [in] DWORD numRequestedRecords,
  [in] DWORD timeOut,
  [in] DWORD flags,
  [out] DWORD* numActualRecords,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataIndices,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataSizes,
  [out] DWORD* resultBufferSize,
  [out, size_is(,*resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
```

```
    BYTE** resultBuffer
);
```

handle: A handle to a subscription. This parameter is an RPC context handle, as specified in [C706] Context Handles.

numRequestedRecords: A 32-bit unsigned integer that contains the maximum number of events to return.

timeOut: A 32-bit unsigned integer that contains the maximum number of milliseconds to wait before returning.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<12>

numActualRecords: A pointer to a 32-bit unsigned integer that contains the value that, on success, MUST be set to the number of events that are retrieved. This is useful in the case in which the method times out without receiving the full number of events specified in *numRequestedRecords*. If the method fails, the client MUST NOT use the value.

eventDataIndices: A pointer to an array of 32-bit unsigned integers that contain the offsets for the events. An event offset is its position relative to the start of *resultBuffer*.

eventDataSizes: A pointer to an array of 32-bit unsigned integers that contain the event sizes in bytes.

resultBufferSize: A pointer to a 32-bit unsigned integer that contains the number of bytes of data returned in *resultBuffer*.

resultBuffer: A pointer to a byte-array that contains the result set of one or more events. The events MUST be in binary XML format, as specified in section 2.2.17.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success. The method MUST return ERROR_TIMEOUT (0x000005b4) if fewer than *numRequestedRecords* records are found within the time-out period. Otherwise, it MUST return a different implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST do the following:

- Validate the handle. For processing rules for handle validation, see the remarks in section 3.1.4.9. The server MUST fail the method with the return code ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid or there is no state for this handle on the server.
- After the server validates the handle, it casts the handle value to the subscription object. The server then MUST check whether the subscription object is a push subscription. Because the subscription object contains the type of subscription, the server checks its type and SHOULD fail the method if it is not a push type subscription with the error ERROR_INVALID_OPERATION(0x000010DD).
- If the handle passes the check, the server MUST determine if the log file contains events to send to the client. These events pass the subscription filters but have not been sent to the client. The subscription filters are the XPath queries that the client specifies in the query parameter in the **EvtRpcRegisterRemoteSubscription** method (as specified in section 3.1.4.8). For information on how the server applies the filter, see [MSDN-CONSUMEVTS].
- If the log file contains events to send to the client, EvtRpcRemoteSubscriptionNext (Opnum 2) starts collecting events to send to the client. Three factors determine the number of events that the server sends to the client:

- The maximum number of records to send to the client. This value is specified by using the *numRequestedRecords* parameter.
- The *timeout interval*. This value is specified by using the *timeOut* parameter and defines the maximum time interval that the caller will wait for a query result. Complex queries and queries that inspect large log files are most likely to encounter the limit specified by the *timeout* value. If the execution time for delivering the next batch of events through the subscription exceeds the timeout value, the server MUST stop working and SHOULD return to the client immediately with the return code ERROR_TIMEOUT (0x000005B4). The server SHOULD treat a *timeout* parameter value of 0xFFFFFFFF as infinite, and process up to the limit of *numRequestedRecords* or the end of the log file regardless of the amount of time such processing takes.
- The end of the log file.

- If the server collects the maximum number of events to send to the client before reaching the end of the log file and before the *timeout interval* expires, the server MUST send the number of events specified in *numRequestedRecords* to the client.
- If the *timeout interval* expires before the server reaches the end of the log file, the server MUST send the collected events to the client. The number of events is less than or equal to the number of events specified in *numRequestedRecords*.
- If the server reaches the end of the log file before the *timeout interval* expires, the server MUST send the collected events to the client. The number of events is less than or equal to the number of events specified in *numRequestedRecords*.

The server returns the result in the five output parameters: *numActualRecords*, *eventDataIndices*, *eventDataSizes*, *resultBufferSize*, and *resultBuffer*. On successful return, the *numActualRecords* contains the number of events in the *resultBuffer*. All the returned events are in BinXML format and they are packed as one binary blob in the *resultBuffer*. The total size of all these events are marked by *resultBufferSize*. Since all the events are packed together, there is a need to identify where the separator is for each event in the result. To do this, the server fills two arrays: eventDataIndices and eventDataSizes. Both arrays contain the *numActualRecords* of elements. For the eventDataIndices array, each array element is a 32-bit value which is the start position of each event in the *resultBuffer*. For the eventDataSizes array, each element is a 32-bit value which is the size of every event.

The server MUST update the position value in the subscription object to keep track of the events received by the client so that subsequent calls can retrieve the rest of the result set. As specified in section 3.1.4.8, the subscription object keeps the positions where the events SHOULD start in the channels. Then the server can update the position value so that it can perform the task of tracking the delivered events. The entire result set in the log file can be retrieved by making a series of calls using EvtRpcRemoteSubscriptionNext (Opnum 2), including entries added to the log file during retrieval of the result set.

The server SHOULD be notified by the underlying network that the connection is lost from the client if the client abnormally terminates the connection. The server abandons its operation for the client in such a case. The server SHOULD release the subscription object it creates and free all associated resources. The associated resources are described in EvtRpcRegisterRemoteSubscription (Opnum 0) (section 3.1.4.8).

The server MUST return a value that indicates success or failure for this operation.

3.1.4.11 EvtRpcRemoteSubscriptionWaitAsync (Opnum 3)

Pull subscriptions are subscriptions in which the client requests records. The requests can be done by using a polling mechanism. The EvtRpcRemoteSubscriptionWaitAsync (Opnum 3) method can be used to enable the client to only poll when results are likely, and is typically used in conjunction with the EvtRpcRemoteSubscriptionNext (Opnum 2) (section 3.1.4.10) method, which is a blocking call; so this

asynchronous method is used to provide a way for the caller to not have to block or continuously poll the server.

```
error_status_t EvtRpcRemoteSubscriptionWaitAsync(
  [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle
);
```

handle: A handle to a subscription, as obtained from the EvtRpcRegisterRemoteSubscription (section 3.1.4.8) method. This parameter MUST be an RPC context handle, as specified in [C706] Context Handles.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. For processing rules for handle validation, see the remarks in section 3.1.4.9. The server SHOULD fail the method with the return code ERROR_INVALID_PARAMETER (0x00000057) if it has no state for the handle.<13>

After the server validates the handle, it casts the handle value to the subscription object. The server then MUST check whether the subscription object is a push subscription. Because the subscription object contains the type of subscription, the server checks its type and SHOULD fail the method if it is not a push type subscription with the error ERROR_INVALID_OPERATION(0x000010DD).

If the preceding check is successful, the server MUST determine whether there are any events the client has not received that pass the subscription filters. The subscription object contains the information of the last event since its last delivery to the client. If there is no new event from the last time the server returned events to the client until the current moment, the server does not complete the call and SHOULD return anything to the client. If there are new events coming in, the server applies the subscription filters and if those events pass the filters, the server SHOULD call RpcAyncCompleteCall (see [MSDN-RpcAsyncCompleteCall]) to complete the async call so that the client will receive notification. Then the client will use **EvtRpcRemoteSubscriptionNext** (as specified in section 3.1.4.10) to get those new events from the server. The subscription filters are the XPath queries that the client specifies in the query parameter in the **EvtRpcRegisterRemoteSubscription** method (as specified in section 3.1.4.8). For information on how the server applies the filter, see [MSDN-CONSUMEVTS]. If there are no events meeting that criteria, the server MUST NOT complete this operation.

The server SHOULD be notified by the underlying network that the connection is lost from the client if the client abnormally terminates the connection. The server abandons its operation for the client in such a case. The server SHOULD release the subscription object it creates and free all associated resources. The associated resources are described in EvtRpcRegisterRemoteSubscription (Opnum 0) (section 3.1.4.8).

The server MUST return a value indicating success or failure for this operation.

3.1.4.12 EvtRpcRegisterLogQuery (Opnum 5)

The EvtRpcRegisterLogQuery (Opnum 5) method is used to query one or more channels. It can also be used to query a specific file. Actual retrieval of events is done by subsequent calls to the EvtRpcQueryNext (section 3.1.4.13) method.

```
error_status_t EvtRpcRegisterLogQuery(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, unique, range(0, MAX_RPC_CHANNEL_PATH_LENGTH), string]
    LPCWSTR path,
  [in, range(1, MAX_RPC_QUERY_LENGTH), string]
    LPCWSTR query,
```

```
  [in] DWORD flags,
  [out, context_handle] PCONTEXT_HANDLE_LOG_QUERY* handle,
  [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* opControl,
  [out] DWORD* queryChannelInfoSize,
  [out, size_is(,*queryChannelInfoSize), range(0, MAX_RPC_QUERY_CHANNEL_SIZE)]
    EvtRpcQueryChannelInfo** queryChannelInfo,
  [out] RpcInfo* error
);
```

binding: An RPC binding handle as specified in section 2.2.21.

path: A pointer to a string that contains a channel or file path.

query: A pointer to a string that contains a query that specifies events of interest to the application. The pointer MUST be either an XPath filter, as specified in section 2.2.15, or a query, as specified in section 2.2.16.

flags: The flags field MUST be set as follows. The first two bits indicate how the *path* argument MUST be interpreted. Callers MUST specify one and only one value.

Value	Meaning
EvtQueryChannelPath 0x00000001	*Path* specifies a channel name.
EvtQueryFilePath 0x00000002	*Path* specifies a file name.

These bits control the direction of the query. Callers MUST specify one and only one value.

Value	Meaning
0x00000100	Events are read from oldest to newest.
0x00000200	Events are read from newest to oldest.

The following bit can be set independently of the previously mentioned bits.

Value	Meaning
0x00001000	Specifies to return the query result set even if one or more errors result from the query. For example, if a structured XML query specifies multiple channels, some channels are valid while others are not. A query that is used on many computers might be sent to a computer that is missing one or more channels in the query. If this bit is not set, the server MUST fail the query. If this bit is set, the query MUST succeed even if all channels are not present.

handle: A pointer to a query handle. This parameter MUST be an RPC context handle, as specified in [C706], Context Handles.

opControl: A pointer to a control handle. This parameter MUST be an RPC context handle, as specified in [C706], Context Handles.

queryChannelInfoSize: A pointer to a 32-bit unsigned integer that contains the number of EvtRpcQueryChannelInfo structures returned in *queryChannelInfo*.

queryChannelInfo: A pointer to an array of EvtRpcQueryChannelInfo structures, as specified in section 2.2.9.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to nonzero values if the error is related to parsing the query; in addition, the server MAY set the structure fields to nonzero values for errors unrelated to query parsing (for example, for an invalid channel name). All nonzero values MUST be treated the same. If the method succeeds, the server MUST set all the fields in the structure to 0.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST fail the method if the *path* parameter is non-NULL and invalid. The server checks the syntax of the query by checking whether the query string is either a valid XPath query (specified in section 2.2.15) or an XML query (specified in section 2.2.16). The server MUST interpret the *path* to be either a channel name or file path name, depending on the *flags* parameter.

The server SHOULD validate all flag values and return the error ERROR_INVALID_PARAMETER (0x00000057) if any of the following conditions occur:<14>

- Neither EvtQueryChannelPath or EvtQueryFilePath is set.
- Both EvtQueryChannelPath and EvtQueryFilePath are set.
- Neither 0x00000100 nor 0x00000200 is set.
- Both 0x00000100 and 0x00000200 are set.
- Any flag that is not specifically defined in this list is set.
- Both the query and the path parameters are NULL.

The server MUST fail the method if the *query* argument is syntactically incorrect. The server checks the syntax of the query by checking if the query string is either a valid XPath query (specified in section 2.2.15) or an XML query (specified in section 2.2.16). The server MAY not validate the semantics of the query.

For example, a client could compose a query that was intended to select all events concerning squares with more than five corners. This is an impossible situation, and the query will never return matching events. But the server has no inherent knowledge about squares; therefore, it has no way to determine that the query is invalid.

The *query* argument MUST be either a simple XPath (for information on the specification of the protocol's support of XPath, see section 2.2.15) or a query (for more information, see section 2.2.16). If the query is not a valid XPath (as specified in section 2.2.15) or allowed query (as specified in section 2.2.16), the server MUST fail the method with the return error ERROR_EVT_INVALID_QUERY (0x00003A99).

In the case of a query, the server MUST verify the validity of any channels or file paths specified in the query. The server SHOULD check whether the given channel is registered or the file path exists for performing the validation of channel or file path. The server SHOULD return the status of all channels found in the query via the *QueryChannelInfo* parameter, along with a return code that specifies the results of querying against that channel (that is, ERROR_SUCCESS (0x00000000) if the channel exists and is accessible). <15> If there is at least one invalid channel or file path in the query and the 0x00001000 bit is not set in the *flags* parameter, the server MUST fail the method either with the error ERROR_EVT_INVALID_CHANNEL_PATH (0x00003A98) or the error ERROR_EVT_INVALID_QUERY (0x00003A99), respectively.

Next, the server MUST verify that the caller has read access to the channel or the specified event log file and MUST fail the method if the caller does not have read access with the error code ERROR_ACCESS_DENIED (0x00000005). To perform the access check, the server SHOULD first

determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the read access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the preceding checks succeed, the server MUST attempt to create a CONTEXT_HANDLE_LOG_QUERY and return it to the caller via the *handle* parameter, and attempt to create a CONTEXT_HANDLE_OPERATION_CONTROL and return it in the *opControl* parameter. When creating the CONTEXT_HANDLE_LOG_QUERY, the server SHOULD create a log query object. The log query object is a class instance that resides in the server's memory to represent the query object for the client (as specified in section 3.1.1.11). Inside the query object, the server SHOULD set the channel path to the channel name or the event log file name the client specified, set the query filter as the XPATH query from the query parameter, and set the position to 0 initially. The position SHOULD be updated each time the client calls **EvtRpcQuerySeek** (as specified in section 3.1.4.14) or **EvtRpcQueryNext** (as specified in section 3.1.4.13). When creating the CONTEXT_HANDLE_OPERATION_CONTROL handle, the server SHOULD create an operation control object (as specified in section 3.1.1.10). The server SHOULD set the operational pointer of the control object to be the pointer of the log query object that the server creates so that it can perform control operations on that object. The server SHOULD also initially set the canceled field in the control object to false. If successful, the server MUST add the created handles to its handle table to track the issued handles. If any of the preceding checks fail, the server MUST NOT create the context handles or add them to the handle table.

The server SHOULD fail to create the two handles only in the case of memory limitation, and the server SHOULD return ERROR_OUTOFMEMORY(0x0000000E) in such case.

The server MUST return a value indicating success or failure for this operation.

3.1.4.13 EvtRpcQueryNext (Opnum 11)

The EvtRpcQueryNext (Opnum 11) method is used by a client to get the next batch of records from a query result set.

```
error_status_t EvtRpcQueryNext(
  [in, context_handle] PCONTEXT_HANDLE_LOG_QUERY logQuery,
  [in] DWORD numRequestedRecords,
  [in] DWORD timeOutEnd,
  [in] DWORD flags,
  [out] DWORD* numActualRecords,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataIndices,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataSizes,
  [out] DWORD* resultBufferSize,
  [out, size_is(,*resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
    BYTE** resultBuffer
);
```

logQuery: A handle to an event log. This parameter is an RPC context handle, as specified in [C706], Context Handles.

numRequestedRecords: A 32-bit unsigned integer that contains the number of events to return.<16>

timeOutEnd: A 32-bit unsigned integer that contains the maximum number of milliseconds to wait before returning, starting from the time the server begins processing the call.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<17>

numActualRecords: A pointer to a 32-bit unsigned integer that contains the value that, on success, MUST be set to the number of events that are retrieved. This is useful when the method times out without receiving the full number of events specified in *numRequestedRecords*. If the method fails, the client MUST NOT use the value.

eventDataIndices: A pointer to an array of 32-bit unsigned integers that contain the offsets (in bytes) within the *resultBuffer* for the events that are read.

eventDataSizes: A pointer to an array of 32-bit unsigned integers that contain the sizes (in bytes) within the *resultBuffer* for the events that are read.

resultBufferSize: A pointer to a 32-bit unsigned integer that contains the number of bytes of data returned in *resultBuffer*.

resultBuffer: A pointer to a byte-array that contains the result set of one or more events. The events MUST be in binary XML format, as specified in section 2.2.17.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success. The method MUST return ERROR_TIMEOUT (0x000005bf) if no records are found within the time-out period. The method MUST return ERROR_NO_MORE_ITEMS (0x00000103) once the query has finished going through all the log(s); otherwise, it MUST return a different implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if the handle is invalid. The server SHOULD save the log handle value it creates in the **EvtRpcRegisterLogQuery** method (as specified in section 3.1.4.12) in its handle table (as specified in section 3.1.1.12) and compare it with the handle passed here to perform the check.<18>

The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid.

If the above check is successful, the server MUST attempt to read through the event log(s) and copy any events that pass the filter into *resultBuffer*. As mentioned in section 3.1.4.12, the context handle corresponds to the log query object on the server side. So the server casts the logQuery handle to its internal log query object after the validation of the handle. The log query object contains the position which indicates how many records the client has already received. The server reads the next record after the position in the event log file. For each record it reads, it tries to match the query filter. If the event passes the filter, the server copies that event record into the client resultBuffer. The server MUST continue the operation until the number of events copied equals the number of events specified by the *numRequestedRecords* parameter, or until the duration of the call exceeds the number of milliseconds specified by the *timeOutEnd* parameter, or until there are no more records to be read. The server MUST update its position in the log query object to keep track of the next event record the server needs to return the next time a client calls this method. If the *timeOutEnd* parameter is 0xFFFFFFFF, the server SHOULD ignore the time-out and process the call as long as it needs without checking the time-out.

If the server cannot find any records in the time specified by the *timeOutEnd* parameter, it MUST return ERROR_TIMEOUT (0x000005bf).

If the server cannot find any records because it reached the end of the file, it MUST return ERROR_NO_MORE_ITEMS (0x00000103).

The server returns the result in the five output parameters: *numActualRecords*, *eventDataIndices*, *eventDataSizes*, *resultBufferSize*, and *resultBuffer*. On successful return, the *numActualRecords* contains the number of events in the *resultBuffer*. All the returned events are in BinXML format and they are packed as one binary blob in the *resultBuffer*. The total size of all these events are marked by *resultBufferSize*. Since all the events are packed together, there is a need to identify where the separator is for each event in the result. To do this, the server fills two arrays: eventDataIndices and eventDataSizes. Both arrays contain the *numActualRecords* of elements. For the eventDataIndices

array, each array element is a 32-bit value which is the start position of each event in the *resultBuffer*. For the eventDataSizes array, each element is a 32-bit value which is the size of every event.

The server MUST return a value indicating success or failure for this operation.

3.1.4.14 EvtRpcQuerySeek (Opnum 12)

The EvtRpcQuerySeek (Opnum 12) method is used by a client to move a query cursor within a result set.

```
error_status_t EvtRpcQuerySeek(
  [in, context_handle] PCONTEXT_HANDLE_LOG_QUERY logQuery,
  [in] __int64 pos,
  [in, unique, range(0, MAX_RPC_BOOKMARK_LENGTH), string]
    LPCWSTR bookmarkXml,
  [in] DWORD timeOut,
  [in] DWORD flags,
  [out] RpcInfo* error
);
```

logQuery: A handle to an event log. This parameter is an RPC context handle, as specified in [C706], Context Handles.

pos: The number of records in the result set to move by. If the number is positive, the movement is the same as the direction of the query that was specified in the EvtRpcRegisterLogQuery (section 3.1.4.12) method call that was used to obtain the handle specified by the *logQuery* parameter. If the number is negative, the movement is in the opposite direction of the query.

bookmarkXml: A pointer to a string that contains a bookmark.

timeOut: A 32-bit unsigned integer that MUST be set to 0x00000000 when sent and MAY be ignored on receipt.

flags: This MUST be set as follows: this 32-bit unsigned integer contains flags that describe the absolute position from which EvtRpcQuerySeek (Opnum 12) starts its seek. The origin flags (the first four flags that follow) are mutually exclusive; however, the last flag can be set independently. The *pos* parameter specifies the offset used in the definitions of these flags.

Value	Meaning
EvtSeekRelativeToFirst 0x00000001	The offset is relative to the first entry in the result set and SHOULD be nonnegative. Therefore, if an offset of 0 is specified, the cursor is moved to the first entry in the result set.
EvtSeekRelativeToLast 0x00000002	The offset is relative to the last entry in the result set and SHOULD be nonpositive. Therefore, if an offset of 0 is specified, the cursor is moved to the last entry in the result set.
EvtSeekRelativeToCurrent 0x00000003	The offset is relative to the current cursor location. If an offset of 0 is specified, the cursor is not to be moved. A positive or negative number can be used in this case to move the cursor to any other location.
EvtSeekRelativeToBookmark 0x00000004	The offset is relative to the bookmark location. If an offset of 0 is specified, the cursor is positioned at the bookmark. A positive or negative number can be used in this case to move the cursor to any other location. The server MUST fail the operation if the *bookmarkXml* parameter does not specify a valid position in the log. The presence of the EvtSeekStrict flag SHOULD influence the behavior of this flag, as specified below.

Value	Meaning
EvtSeekStrict 0x00010000	If this is set, the query fails if the seek cannot go to the record indicated by the other flags/parameters. If not set, the seek uses a best effort. For example, if 99 records remain in the result set and the *pos* parameter specifies 100 with the EvtSeekRelativeToCurrent flag set, the 99th record is selected.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo structure fields MUST be set to nonzero values if the error is related to parsing the query. In addition, the server MAY set the structure fields to nonzero values for errors unrelated to query parsing. All nonzero values MUST be treated the same by the client.

If the method succeeds, the server MUST set all the values in the structure to zero.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. For processing rules for handle validation, see the remarks in section 3.1.4.13. The server SHOULD fail the method with the return code ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid.

The server SHOULD<19> validate that the sign of the *pos* parameter makes sense for the search direction. That is, the server ~~SHOULD return~~returns ERROR_INVALID_PARAMETER (0x00000057) if a negative *pos* value is specified along with the EvtSeekRelativeToFirst flag and if a positive *pos* value is specified along with the EvtSeekRelativeToLast flag~~.<19>~~.

The server SHOULD validate that the *bookmarkXML* parameter has the correct syntax for the book mark. If it is not correct, the server SHOULD return ERROR_INVALID_PARAMETER (0x00000057).

The server MUST validate that one and only one of the mutually exclusive flags are specified and return ERROR_INVALID_PARAMETER (0x00000057) if this condition is not met. The mutually exclusive flags are:

- EvtSeekRelativeToFirst
- EvtSeekRelativeToLast
- EvtSeekRelativeToCurrent
- EvtSeekRelativeToBookmark

If validation succeeds, the server uses the address of the logQuery context handle as a pointer to the log query object, with implementation-specific typecasting as necessary. Then the following operations SHOULD be done:

1. Set the position value in the log query object to the initial value based on the flags.
    1. EvtSeekRelativeToFirst: Position set to 0.
    2. EvtSeekRelativeToLast: Position set to the number of records in the channel.
    3. EvtSeekRelativeToCurrent: Position unchanged.
    4. EvtSeekRelativeToBookmark: Read the event record Id from the bookmark XML, read every event from the beginning and try to find the same event record Id as specified in the bookmark XML. The position is the value of how many records the server has read before finding the same event record Ids.
2. When *pos* parameter is bigger than 0, the server reads one event record from its current position and increments the position value by 1. With the event record, the server tries to match the query

filter (the XPath expression). If the event matches the filter requirement, the server decreases the *pos* value by 1. If the event does not match, the *pos* value is kept the same. Then the server reads the next record, and repeats the process until the *pos* value becomes 0. Then the server returns to the client indicating the seek operation is finished.

3. When *pos* parameter is a negative value, the server reads the event record in reverse order. It reads the previous event record from its current position and decrements the position value by 1 each time it reads a previous record. With the record it reads, it tries to match the query filter (the XPath expression). If the event matches the filter requirement, the server increases the *pos* value by 1. If the event does not match, the *pos* value is kept the same. Next, the server reads the next previous record. This process is repeated until the *pos* value becomes 0. Then the server returns the value to the client indicating that the seek operation is finished.

If the client specifies the EvtSeekRelativeToBookmark flag and the server can't find the event record Id that matches the record Id in the bookmark XML, the server SHOULD return ERROR_NOT_FOUND (0x00000490) if the client specifies the EvtSeekStrict at the same time. Otherwise, the server tries to set the position to the nearest record matching the record Id specified in the bookmark. For example, if the record Id in the bookmark is 1000, and the event records in the log only has 999, 1002, 1003 as the record Ids, the server SHOULD stops at the event record whose record Id is 999.

In the previous server mutually exclusive flags validation, steps 2 or 3, if the server reaches either the beginning or the end of the event log file before the *pos* parameter reaches 0, the server SHOULD check if the client has specified the flag EvtSeekSrict. If so, the server will not return error. Otherwise, the server SHOULD return ERROR_NOT_FOUND (0x00000490).

The server MUST return a value indicating success or failure for this operation.

3.1.4.15 EvtRpcGetLogFileInfo (Opnum 18)

The EvtRpcGetLogFileInfo (Opnum 18) method is used by a client to get information about a live channel or a backup event log.

```
error_status_t EvtRpcGetLogFileInfo(
  [in, context_handle] PCONTEXT_HANDLE_LOG_HANDLE logHandle,
  [in] DWORD propertyId,
  [in, range(0, MAX_RPC_PROPERTY_BUFFER_SIZE)]
    DWORD propertyValueBufferSize,
  [out, size_is(propertyValueBufferSize)]
    BYTE* propertyValueBuffer,
  [out] DWORD* propertyValueBufferLength
);
```

logHandle: A handle to an event log. This parameter is an RPC context handle, as specified in [C706], Context Handles. For more information about the server-side object that maps to this handle, see section 3.1.4.19.

propertyId: A 32-bit unsigned integer that indicates what log file property (as specified in section 3.1.1.6) needs to be retrieved.

Value	Meaning
EvtLogCreationTime 0x00000000	A FILETIME containing the creation time of the file. This is the creation time of a log file associated with the channel or the creation time of the backup event log file in the server's file system.
EvtLogLastAccessTime 0x00000001	A FILETIME containing the last access time of the file. This is the last access time of a log file associated with the channel or the last access time of the backup event log file in the server's file system.

Value	Meaning
EvtLogLastWriteTime 0x00000002	A FILETIME containing the last write time of the file. This is the last written time of a log file associated with the channel or the last written time of the backup event log file in the server's file system.
EvtLogFileSize 0x00000003	An unsigned 64-bit integer containing the size of the file. This is the file size of a log file associated with the channel or the file size of the backup event log file in the server's file system.
EvtLogAttributes 0x00000004	An unsigned 32-bit integer containing the attributes of the file. The attributes are implementation-specific, and clients MUST<20> treat all values equally. The attributes are tracked by the server's file system and SHOULD be able to be retrieved from the file system.
EvtLogNumberOfLogRecords 0x00000005	An unsigned 64-bit integer containing the number of records in the file. See the following processing rules for how the server gets this value.
EvtLogOldestRecordNumber 0x00000006	An unsigned 64-bit integer containing the oldest record number in the file. See the following processing rules for how the server gets this value.
EvtLogFull 0x00000007	A BOOLEAN value; MUST be true if the log is full, and MUST be false otherwise. See the following processing rules for how the server gets this value.

propertyValueBufferSize: A 32-bit unsigned integer that contains the length of caller's buffer in bytes.

propertyValueBuffer: A byte-array that contains the buffer for returned data.

propertyValueBufferLength: A pointer to a 32-bit unsigned integer that contains the size in bytes of the returned data.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success. The method MUST return ERROR_INSUFFICIENT_BUFFER (0x0000007A) if the buffer is too small; otherwise, it MUST return a different implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the log handle value it creates in the EvtRpcOpenLogHandle (section 3.1.4.19) method in its handle table (as specified in section 3.1.1.12) so that it can compare that value with the value in the *logHandle* parameter to perform the check. If the values differ, the handle is invalid.<21> The server MUST fail the operation if the handle is invalid with the error code ERROR_INVALID_PARAMETER (0x00000057).

Next, the server MUST verify the *propertyId* value as one specified in the preceding *propertyId's* fields table. Otherwise, it SHOULD return ERROR_INVALID_PARAMETER (0x00000057).

If *propertyValueBufferSize* is too small, the server MUST return the size needed in the *propertyValueBufferLength* parameter and fail the method with a return code of ERROR_INSUFFICIENT_BUFFER (0X0000007A).

If the preceding checks succeed, the server MUST attempt to return the request information. The server SHOULD first cast the logHandle into the log object. The server SHOULD decide if the Channel pointer points to a live channel or the handle to a backup event log file based on the LogType field. If it is a live channel, the server SHOULD get the associated log file path and open the file to get a file handle. If it is a backup event log file, the log object contains the handle to the file. Then the server SHOULD get the *EvtLogCreationTime*, *EvtLogLastAccessTime*, *EvtLogLastWriteTime*, *EvtLogFileSize*, and *EvtLogAttributes* information by querying the file system to get the creation time, last access time, last written time, file size, and file attributes of the specified log file (if channel is specified, the log file is the disk file which associates with the channel).

Note This information is tracked by the file system automatically and the server does not need to touch any files for any operation, such as exporting events from the channel or clearing events in a channel.

The server keeps the number of event records, the oldest event record, and the log full flag in its live channel file (log file associated with the channel) or backup event log file header (as specified in section 3.1.1.6). The server reads the information directly when returning the mentioned properties to the client.

The server MUST pack the return data into a single BinXmlVariant structure, as specified in section 2.2.18, and copy it into the buffer that is pointed to by the *propertyValueBuffer* parameter. The server MUST NOT update its state.

The server MUST return a value indicating success or failure for this operation.

3.1.4.16 EvtRpcClearLog (Opnum 6)

The EvtRpcClearLog (Opnum 6) method instructs the server to clear all the events in a live channel, and optionally, to create a backup event log before the clear takes place.

```
error_status_t EvtRpcClearLog(
  [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control,
  [in, range(0, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in, unique, range(0, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR backupPath,
  [in] DWORD flags,
  [out] RpcInfo* error
);
```

control: A handle to an operation control object. This parameter is an RPC context handle, as specified in [C706], Context Handles.

channelPath: A pointer to a string that contains the path of the channel to be cleared.

backupPath: A pointer to a string that contains the path of the file in which events are to be saved before the clear is performed. A value of NULL indicates that no backup event log is to be created (the events to be cleared are not to be saved).

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<22>

error: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].<23>

Return Values: The method returns 0 (ERROR_SUCCESS) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

The server does not validate the control handle passed to **EvtRpcClearLog** and it SHOULD assume that this parameter is always valid when the method is invoked.

The server MUST verify that the *channelPath* parameter specifies a correct channel name by looking up the channel name in its channel table. The server SHOULD fail the call if the *channelPath* parameter is not an entry in its channel table with the error code ERROR_EVT_CHANNEL_NOT_FOUND (0x00003A9F).

If the *backupPath* parameter is non-NULL and non-empty, the server MUST validate the path and fail the call if it is not a file path (an illegal file path for the server's file system) or if it specifies a file that already exists. If the path is an illegal file path, the server SHOULD return the error

ERROR_INVALID_PARAMETER (0x00000057). If the path specifies a file which exists on the server, the server SHOULD return the error ERROR_FILE_EXISTS (0x00000050).

Next, the server MUST verify if the client has write and clear access to the channel and write access to the backup file if specified. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2). The server MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the client does not have write and clear access to the channel.

If the client specifies the backupPath, the server SHOULD first impersonate the identity of the caller. For information on how to impersonate the client's identity for the purpose of performing an authorization or security check, see [MS-RPCE] (section 3.3.3.4.3). Then the server SHOULD call the file system component to attempt to create the backup file. Once the server impersonates the client's identity, it can determine whether the client has write access because the file creation will fail with ERROR_ACCESS_DENIED (0x00000005) if the client does not have write access. If the server fails to create the backup file, it MUST return the error (a nonzero value as specified in [MS-ERREF]) reported by the underlying file system component.<24> Otherwise, the server MUST successfully create the backup file.

If the *backupPath* parameter is valid, the server MUST attempt to back up the log to the path specified in the *backupPath* parameter before the log is cleared. The method MUST fail and not clear the log if the backup does not succeed with any possible implementation-based error code.

If the *backupPath* parameter is NULL or empty, the method MUST NOT attempt to back up the event log but SHOULD still clear the events in the channel.

If the previous checks are successful and if there are no problems in creating a backup log, the server MUST attempt to clear the associated event log. All events MUST be removed during clearing. During this process, the server SHOULD check the **Canceled** field of the operation control object in the *control* parameter periodically, for example, once every 100 milliseconds. If the **Canceled** field becomes TRUE and the clearing operation has not been finished, the server SHOULD abandon the current operation and return to the client immediately with the error code ERROR_CANCELLED (0x000004C7) without updating any state. On a successful return, the server SHOULD reset the **numberOfRecords** to 0 and **isLogFull** to false for the header of its associated log file for the channel. The server does not need to update the **curPhysicalRecordNumber** and **oldestEventRecordNumber**. The **LogCreationTime**, **LogLastAccessTime**, **LogLastWriteTime**, **LogFileSize**, and **LogAttributes** attributes of the associated log file for the channel are tracked by the server's file system.

If all events are successfully deleted ("cleared"), the server MUST return ERROR_SUCCESS to indicate success. This method SHOULD only fail in extreme conditions, such as lack of system resources, file system error, or hardware error, and such issues are not part of the processing for the EventLog Remoting Protocol Version 6.0. In these cases, the method MUST return an implementation-specific error, as specified in [MS-ERREF], from lower level components unchanged. Depending on the server's implementation detail, the protocol has no specific error return recommendation other than it MUST come from [MS-ERREF].

Note The server does not need to update any state or information for the created backup event log file.

3.1.4.17 EvtRpcExportLog (Opnum 7)

The EvtRpcExportLog (Opnum 7) method instructs the server to create a backup event log at a specified file name.

```
error_status_t EvtRpcExportLog(
  [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control,
  [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in, range(1, MAX_RPC_QUERY_LENGTH), string]
    LPCWSTR query,
  [in, range(1, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR backupPath,
  [in] DWORD flags,
  [out] RpcInfo* error
);
```

control: A handle to an operation control object. This parameter is an RPC context handle, as specified in [C706] Context Handles.

channelPath: A pointer to a string that contains the channel name (for a live event log) or file path (for an existing backup event log) to be used to create a backup event log.

query: A pointer to a string that contains a query that specifies events to be included in the backup event log.

backupPath: A pointer to a string that contains the path of the file for the backup event logs to be created.

flags: The client MUST set the *flags* parameter to one of the following values.

Value	Meaning
EvtQueryChannelPath 0x00000001	*Channel* parameter specifies a channel name.
EvtQueryFilePath 0x00000002	*Channel* parameter specifies a file name.

In addition, the client MAY set the following value in the *flags* parameter:

Value	Meaning
EvtQueryTolerateQueryErrors 0x00001000	The query MUST succeed even if not all the channels or backup event logs that are specified in the query are present, or if the *channelPath* parameter specifies channels that do not exist.

The server MAY ignore unrecognized flag combinations.<25>

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to a nonzero value if the error is related to parsing the query. In addition, the server MAY set the suberror fields to nonzero values for other types of errors. All nonzero values MUST be treated the same. If the method succeeds, the server MUST set all of the values in the structure to 0.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

The server does not validate the control handle passed to **EvtRpcExportLog**, and it SHOULD assume that this parameter is always valid when the method is invoked.

In response to this request from the client, if the *flags* parameter contains the value 0x00000001 (flags & 0x00000001 != 0), the server MUST interpret the *channel* parameter as a channel name. The server then SHOULD search its channel table to find the corresponding entry which has the same

channel name. If the server can't find the entry, the specified channel name is invalid and the server SHOULD return ERROR_EVT_CHANNEL_NOT_FOUND (0x00003A9F). If the *flags* parameter contains the value 0x00000002 (flags & 0x00000002 != 0), the server MUST interpret channel as an existing backup event log file name. The server SHOULD then check if the specified file exists on the server. If the file does not exist, the file path is invalid and the server SHOULD return ERROR_FILE_NOT_FOUND (0x00000002).

The server SHOULD validate that the flags contain one and only one of EvtQueryChannelPath and EvtQueryFilePath; and that no flags which are not defined above are specified. The server SHOULD return error code ERROR_INVALID_PARAMTER (0x00000057) if the flag validation fails.<26>

The server MUST verify that the *query* parameter is a valid XPath expression with correct syntax, based on the grammar definition provided in section 2.2.15 and if it is not, fail the operation with the error code ERROR_INVALID_PARAMETER (0x00000057). For information on XPath filters supported by this protocol, see section 2.2.15.

The server MUST verify that *backupPath* is a valid path (a legal file name for the server's file system), and fail the method if it is not valid or if it specifies a file that already exists. The server SHOULD return ERROR_INVALID_PARAMETER (0x00000057) if the path is invalid or ERROR_FILE_EXISTS (0x00000050) if the specified file already exists on the server.

Next, the server MUST verify that the caller has read access to the channel or the backup event log file and MUST fail the method if the caller does not have read access with the error code ERROR_ACCESS_DENIED (0x00000005). To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

During the exporting log process, the server SHOULD check the **Canceled** field of the operation control object in the *control* parameter periodically, for example, once every 100 milliseconds. If the **Canceled** field becomes TRUE and the exporting operation has not been finished, the server SHOULD abandon the current operation and return to the client immediately with the error code ERROR_CANCELLED (0x000004C7) without updating any state. If the server has created a new backup file and the operation has been canceled, the created file SHOULD be deleted. Failure to delete the file SHOULD NOT trigger the server to take any further actions in response.

If the checks above are successful, the server MUST attempt to create a new backup event log that contains only the records selected by the filter specified by the *query* parameter. The server SHOULD first impersonate the identity of the client. For information on how to impersonate the client's identity for the purpose of performing an authorization or security check, see [MS-RPCE] (section 3.3.3.4.3). Then the server SHOULD call the file system component to create a new backup event log file. Once the server impersonates the client's identity, it can determine whether the client has write access because the file creation will fail with ERROR_ACCESS_DENIED (0x00000005) if the client does not have write access. If the server fails to create the new backup event log file, it MUST return the error (a nonzero value as specified in [MS-ERREF]) reported by the underlying file system component.<27> Otherwise, the server MUST successfully create the file. There is no server state that needs to be updated by this method. However, the server SHOULD ensure the **LogNumberOfRecords**, **LogOldestRecordNumber**, and **LogFull** properties of the created backup log are the correct value. If the *query* parameter is NULL, the created backup event log file SHOULD be the copy of the event log file associated with the live channel so that the **LogNumberOfRecords**, **LogOldestRecordNumber**, and **LogFull** properties are kept in the backup event log file and consequently have the same values as in the event log file associated with the live channel.

If the *query* parameter is not NULL, the server SHOULD then read each event from the log file associated with the live channel and determine whether it meets the criteria specified by the *query* parameter. For every event that passes the filter given in the *query* parameter, the server SHOULD write it to the created backup file. The event record number of the first event that is written into the

created backup file SHOULD be the value of **LogOldestRecordNumber**. The **LogNumberOfRecords** property SHOULD be set to the number of total events the server writes to the backup file. The server SHOULD set the **isLogFull** property to be FALSE and SHOULD set the **curPhysicalRecordNumber** property to the value of (**LogNumberOfRecords** - 1).

The created backup file SHOULD be treated as read-only and never modified subsequently.

The **LogCreationTime**, **LogLastAccessTime**, **LogLastWriteTime**, **LogFileSize**, and **LogAttributes** attributes of the created backup event log file are tracked by the server's file system. The **LogNumberOfRecords**, **LogOldestRecordNumber**, and **LogFull** attributes are tracked by **numberOfRecords**, **oldestRecordNumber**, and **isLogFull** in the backup event log file header.

The server MUST return a value indicating success or failure for this operation.

Note The exported backup event log file does not contain the localized event description strings because the localized strings would consume considerable storage space if included in the exported log file. If the backup log is consumed on the same machine where it is created or on other machines where the publisher is registered, the strings can be retrieved from the publisher on demand. Localized event description strings only need to be added to an exported backup event log file when that file is moved to a different machine where the publisher is not registered. Localized event description strings can be added to an exported backup event log file by calling the EvtRpcLocalizeExportLog (section 3.1.4.18) method.

3.1.4.18 EvtRpcLocalizeExportLog (Opnum 8)

The EvtRpcLocalizeExportLog (Opnum 8) method is used by a client to add localized information to a previously created backup event log, because the backup event log does not contain the localized strings for event descriptions. An example of how this can be useful is if a backup event log needs to be copied to other computers so that support personnel on those other computers can access it; if this method has been called, such support personnel can access or view the localized backup event log, which will then contain events with localized strings. Note that a discussion of tools by which administrators or support personnel can work with localized backup event log files in scenarios such as this is out of scope with respect to this protocol specification.

```
error_status_t EvtRpcLocalizeExportLog(
  [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control,
  [in, range(1, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR logFilePath,
  [in] LCID locale,
  [in] DWORD flags,
  [out] RpcInfo* error
);
```

control: A handle to an operation control object. This parameter MUST be an RPC context handle, as specified in [C706], Context Handles.

logFilePath: A pointer to a string that contains the path of the backup event log to be localized.

locale: Locale, as specified in [MS-GPSI] Appendix A, to be used for localizing the log.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<28>

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to nonzero values if the error is related to loading localization information. All nonzero values MUST be treated the same. If the method succeeds, the server MUST set all of the values in the structure to zero.<29>

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an error value as specified in the processing rules in this section.<30> Callers SHOULD treat all return values other than ERROR_SUCCESS equally and not alter their behavior based on any specific error values.

The server does not validate the control handle passed to **EvtRpcLocalizeExportLog**, and it SHOULD assume that this parameter is always valid when the method is invoked.

In response to this request from the client, the server MUST verify that the *logFilePath* parameter specifies a valid path to a backup event log. A valid path MUST be a legal file name of the server's file system. The server MUST fail the operation if the *logFilePath* parameter is invalid with the error ERROR_INVALID_PARAMETER (0x00000057).<31> The server MUST fail the method if the specified backup event log does not exist in the server with the error ERROR_FILE_NOT_FOUND (0x00000002).

Next the server MUST verify that the caller has read access to the log file (specified by the *logFilePath* parameter) and MUST fail the method if the caller does not have read access with the error ERROR_ACCESS_DENIED (0x00000005). To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then the server SHOULD get the security descriptor string from the file system for the log file. Next the server SHOULD perform the read access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the checks above are successful, the server MUST perform the following operations:

1. The server creates a subdirectory "LocaleMetaData", if the directory does not exist, under the directory where the backup event log file is located (see [PRA-CreateDirectory]). If the directory already exists, the server does nothing. The only expected failures for subdirectory creation are critical system errors, such as file system errors. If the server cannot create the directory, it MUST return the error from the CreateDirectory method that is reporting the error. Otherwise, the server MUST successfully create the subdirectory.

2. The server creates a file with the name *LogFilePath*_<*Locale*>.MTA under the directory LocaleMetaData (see [PRA-CreateFile]). If the file already exists, the server SHOULD always overwrite it. The only expected failures for file creation or overwriting are critical system errors, such as file system errors. If the server can't create the file or overwrite an existing one, it MUST return the error from the CreateFile method that is reporting the error. Otherwise, the server MUST successfully create the file.

3. The server then opens the backup event log file, reads every event and uses the same internal functionality by which it implements the **EvtRpcMessageRender** method (section 3.1.4.31) to obtain the localized strings for event levels, keywords, tasks, opcode, and descriptions. The server then saves those localized strings of each event in the newly created file. Note that the **EvtRpcMessageRender** method needs the PCONTEXT_HANDLE_PUBLISHER_METADATA handle as its first parameter. When the server gets each event, it can get the event publisher name from the event content (see section 2.2.13), thus the server is able to get the context handle by using the internal functionality by which it implements the **EvtRpcGetPublisherMetadata** method (specified in section 3.1.4.25). The internal functionality by which the server implements **EvtRpcGetPublisherMetadata** SHOULD use the value of the *locale* parameter in its processing; the server SHOULD make this value available to that internal functionality by appropriate platform-specific means so that the value can be stored in the publisher metadata object.

   After getting the publisher metadata context handle, the server SHOULD extract the eventId, level, keywords, tasks, and opcode values from the event and fill an **EVENT_DESCRIPTOR** structure, specified in [MS-DTYP] section 2.3.1. With the context handle and the **EVENT_DESCRIPTOR** structure, the server can use the internal functionality by which it implements the **EvtRpcMessageRender** method five times to obtain the localized level, keyword, tasks, opcode, and event description strings. If the server receives an error from the internal functionality by which it implements the **EvtRpcMessageRender** method, it SHOULD ignore the error and continue processing the next event.

During the preceding process, the server SHOULD check the **Canceled** field of the operation control object in the *control* parameter periodically, for example, once every 100 milliseconds. If the **Canceled** field becomes TRUE and the whole operation has not been finished, the server SHOULD abandon the current operation and return to the client immediately with the error code ERROR_CANCELLED (0x000004C7) without updating any state. Any directory or file that has been created SHOULD be deleted. Failure to delete the directory or file SHOULD NOT trigger the server to take any further actions in response.

The server MUST return a value indicating success or failure for this operation.

The server SHOULD create a separate file with the name *LogFilePath*_<*Locale*>.MTA to hold the localized strings for the events in the log file with the name of *LogFilePath*. This method does not change anything in the unlocalized, exported log file. Instead, it generates a separate file that contains the localized strings for the events in the exported log file. The user needs to keep both files together when copying them to another computer in order to consume the events with the localized strings. Any protocol method that can access the exported log file will also be able to access the copied exported file. However, correlating the localized strings with the unlocalized event information in the exported log file in a meaningful way for the user is outside of the scope of this protocol. This protocol provides no methods for integrating the localized strings and the exported event log into a single format for presentation.

3.1.4.19 EvtRpcOpenLogHandle (Opnum 17)

The EvtRpcOpenLogHandle (Opnum 17) method is used by a client to get information about a channel or a backup event log.

```
error_status_t EvtRpcOpenLogHandle(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channel,
  [in] DWORD flags,
  [out, context_handle] PCONTEXT_HANDLE_LOG_HANDLE* handle,
  [out] RpcInfo* error
);
```

binding: An RPC binding handle as specified in section 2.2.21.

channel: A pointer to a string that contains a channel or a file path.

flags: MUST be one of the following two values.

Value	Meaning
0x00000001	*Channel* parameter specifies a channel name.
0x00000002	*Channel* parameter specifies a file name.

handle: A pointer to a log handle. This parameter is an RPC context handle, as specified in [C706], Context Handles.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The server MAY set the suberror fields to supply more comprehensive error information.<32> If the method succeeds, the server MUST set all of the values in the structure to 0.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the *channel* parameter. The server SHOULD search for the given channel name in its channel table. If the server doesn't find the name, the specified channel name is not valid. If the specified channel name is invalid, the server SHOULD return the error code ERROR_EVT_CHANNEL_NOT_FOUND (0x00003A9F). If the *flags* parameter is set to 0x00000001, the server MUST interpret the *channel* parameter as a channel name. If the *flags* parameter is set to 0x00000002, the server MUST interpret channel as the path to an existing event log file. The server SHOULD return ERROR_INVALID_PARAMETER (0x00000057) if the *flags* parameter is not 0x00000001 or 0x00000002.<33> The server checks this by calling the file system to check if the file exists. If the event log file does not exist on the server, the server SHOULD return the error code ERROR_FILE_NOT_FOUND (0x00000002).

Next the server MUST verify that the caller has read access to the channel or the file and MUST fail the method if the caller does not have read access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2). The server MUST fail the method with the error code ERROR_ACCESS_DENIED (0x00000005) if the client does not have read access to the channel or the file.

If the preceding checks succeed, the server MUST attempt to create a CONTEXT_HANDLE_LOG_HANDLE. To perform this operation, the server SHOULD create a log object as specified in section 3.1.1.11. This object is the server-side object for CONTEXT_HANDLE_LOG_HANDLE. The server SHOULD add the newly created handle to its handle table in order to track it.

The server SHOULD set the **LogType** field of the log object to be either a channel or a backup event log based on the client's input flags value. If the type is channel, the server SHOULD try to find the channel in its channel table and SHOULD fail the method with ERROR_CHANNEL_NOT_FOUND (0x00003A9F) if the server cannot find the channel. After the channel is found, the server SHOULD set the **Channel** field of the log object to be the pointer that points to the channel entry in the channel table. If the type is backup event log file, the server SHOULD try to check if the file exists and SHOULD fail the method with the ERROR_FILE_NOT_FOUND (0x00000002) if the backup event log file does not exist. If the backup event log file exists, the server SHOULD try to open the backup event log file (see [PRA-CreateFile]) and set the **Channel** field of the log object to be the file handle if the server successfully opens the file. If the server fails to open the file, it MUST return the error from the **CreateFile** method that is reporting the error.

If any of the preceding checks fail, the server MUST NOT create the context handle.

The server MUST return a value indicating success or failure for this operation.

3.1.4.20 EvtRpcGetChannelList (Opnum 19)

The EvtRpcGetChannelList (Opnum 19) method is used to enumerate the set of available channels.

```
error_status_t EvtRpcGetChannelList(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in] DWORD flags,
  [out] DWORD* numChannelPaths,
  [out, size_is(,*numChannelPaths), range(0, MAX_RPC_CHANNEL_COUNT), string]
    LPWSTR** channelPaths
);
```

binding: An RPC binding handle as specified in section 2.2.21.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<34>

numChannelPaths: A pointer to a 32-bit unsigned integer that contains the number of channel names.

channelPaths: A pointer to an array of strings that contain all the channel names.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST verify that the caller has read access to the channel list and MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the caller does not have read access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the above check succeeds, the server MUST return a list of channel name strings. The server SHOULD enumerate all the channels in its channel table (section 3.1.1.5) and read out the channel name strings as the result for the out parameter *channelPaths*. Meanwhile, the value pointed to by *numChannelPaths* SHOULD be set to the number of channel name strings in the server channel table. The server SHOULD only fail the method due to shortage of memory in which case the server SHOULD return ERROR_OUTOFMEMORY (0x0000000E). The server MUST NOT update its state.

The server MUST return a value indicating success or failure for this operation.

3.1.4.21 EvtRpcGetChannelConfig (Opnum 20)

The EvtRpcGetChannelConfig (opnum 20) method is used by a client to get the configuration for a channel.

```
error_status_t EvtRpcGetChannelConfig(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in] DWORD flags,
  [out] EvtRpcVariantList* props
);
```

binding: An RPC binding handle as specified in section 2.2.21.

channelPath: A pointer to a string that contains the name of a channel for which the information is needed.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<35>

props: A pointer to an EvtRpcVariantList structure to be filled with channel properties, as defined in the following table.

Note The index column in the following table is the array index, not the actual field of the **EvtRpcVariantList** structure. The returned data is an array of **EvtRpcVariantList** for which the index value is used to identify the elements in the array. For example, index 0 means the first element of the returned array.

Index	Type	Meaning
0	EvtRpcVarTypeBoolean	Enabled. If true, the channel can accept new events. If false, any attempts to write events into this channel are automatically dropped.
1	EvtRpcVarTypeUInt32	Channel Isolation. It defines the default access permissions for the channel. Three values are allowed: ▪ 0: Application. ▪ 1: System. ▪ 2: Custom. The default isolation is Application. The default permissions for Application are (shown using SDDL): ▪ L"O:BAG:SYD:" ▪ L"(A;;0xf0007;;;SY)" // local system (read, write, clear) ▪ L"(A;;0x7;;;BA)" // built-in admins (read, write, clear) ▪ L"(A;;0x7;;;SO)" // server operators (read, write, clear) ▪ L"(A;;0x3;;;IU)" // INTERACTIVE LOGON (read, write) ▪ L"(A;;0x3;;;SU)" // SERVICES LOGON (read, write) ▪ L"(A;;0x3;;;S-1-5-3)" // BATCH LOGON (read, write) ▪ L"(A;;0x3;;;S-1-5-33)" // write restricted service (read,write) ▪ L"(A;;0x1;;;S-1-5-32-573)"; // event log readers (read) The default permissions for System are (shown using SDDL): ▪ L"O:BAG:SYD:" ▪ L"(A;;0xf0007;;;SY)" // local system (read, write, clear) ▪ L"(A;;0x7;;;BA)" // built-in admins (read, write, clear) ▪ L"(A;;0x3;;;BO)" // backup operators (read, write) ▪ L"(A;;0x5;;;SO)" // server operators (read, clear) ▪ L"(A;;0x1;;;IU)" // INTERACTIVE LOGON (read) ▪ L"(A;;0x3;;;SU)" // SERVICES LOGON (read, write) ▪ L"(A;;0x1;;;S-1-5-3)" // BATCH LOGON (read) ▪ L"(A;;0x2;;;S-1-5-33)" // write restricted service (write) ▪ L"(A;;0x1;;;S-1-5-32-573)"; // event log readers (read) When the Custom value is used, the Access property will contain the defined SDDL.
2	EvtRpcVarTypeUInt32	Channel type. One of four values:

Index	Type	Meaning
		▪ 0: Admin ▪ 1: Operational ▪ 2: Analytic ▪ 3: Debug For more information, see [MSDN-EVTLGCHWINEVTLG].
3	EvtRpcVarTypeString	OwningPublisher. Name of the publisher that defines and registers the channel with the system. For more information on how the server reacts to changes of this property, see section 3.1.4.22.
4	EvtRpcVarTypeBoolean	ClassicEventlog. If true, the channel represents an event log created according to the EventLog Remoting Protocol, not this protocol (EventLog Remoting Protocol Version 6.0). The server maintains two channel tables: one for the EventLog Remoting Protocol Version 6.0 and one for the legacy EventLog Remoting Protocol. The table for the legacy EventLog Remoting Protocol is called "log table". For more information on the legacy "log table", see [MS-EVEN]. Any channel coming from the new "channel table" gets the value as false, any channel name that is in the legacy "log table" gets the value as true.
5	EvtRpcVarTypeString	Access. A Security Descriptor Description Language (SDDL) string, as specified in [MS-DTYP], that represents access permissions to the channels. If the **isolation** attribute is set to Application or System, the access descriptor controls read access to the file (the write permissions are ignored). If the **isolation** attribute is set to Custom, the access descriptor controls write access to the channel and read access to the file.
6	EvtRpcVarTypeBoolean	Retention. If set to true, events can never be overwritten unless explicitly cleared. If set to false, events are overwritten as needed when the event log is full.
7	EvtRpcVarTypeBoolean	AutoBackup. When set to true, the event log file associated with the channel is closed as soon as it reaches the maximum size specified by the MaxSize property, and a new file is opened to accept new events. If the new file reaches maximum size, another new file will be generated and the previous new file will be backed up. The events in backed up files cannot be queried from this channel in the server unless the client specifies the backup log file names in a separate query.
8	EvtRpcVarTypeUInt64	MaxSize. The value that indicates at which point the size (in bytes) of the event log file stops increasing. When the size is greater than or equal to this value, the file growth stops.
9	EvtRpcVarTypeString	LogFilePath. File path to the event log file for the channel. The path is saved in the channel config and read out by the server when client requests it.
10	EvtRpcVarTypeUInt32	Level. Events with a level property less than or equal to this specified value are logged to the channel.
11	EvtRpcVarTypeUInt64	Keywords. Events with a keyword bit contained in the Keywords bitmask set are logged to the channel.
12	EvtRpcVarTypeGuid	ControlGuid. A GUID value. The client SHOULD ignore this value.

Index	Type	Meaning
13	EvtRpcVarTypeUInt64	BufferSize. Size of the events buffer (in kilobytes) used for asynchronous event delivery. This property is for providing events. Typically the events generated by a publisher are first written to memory buffers on the server. Once the buffer used is full, that buffer is written to a disk file. The BufferSize is used to specify the size of the buffer. The server allocates buffers according to the BufferSize value. The number of buffers the server can allocate is controlled by the MinBuffers and MaxBuffers properties. The server's specific implementation can allocate any number of buffers between MinBuffers and MaxBuffers.
14	EvtRpcVarTypeUInt32	MinBuffers. The minimum number of buffers used for asynchronous event delivery. For more information, see the preceding BufferSize information.
15	EvtRpcVarTypeUInt32	MaxBuffers. The maximum number of buffers used for asynchronous event delivery. For more information, see the preceding BufferSize information.
16	EvtRpcVarTypeUInt32	Latency. The number of seconds of inactivity (if events are delivered asynchronously and no new events are arriving) after which the event buffers MUST be flushed to the server. As specified in the description for BufferSize property, the server keeps a number of buffers when writing events. If the buffers are full, the server writes the buffers to disk file. However, if a certain amount of time elapses and the buffers are still not full, the server SHOULD write the buffers to disk. That certain amount of time is the latency property.
17	EvtRpcVarTypeUInt32	ClockType. One of two values: ▪ 0: SystemTime. Use the system time. When set to this value, the server uses the system time type (which is low-resolution on most platforms) for a time stamp field of any event it writes into this channel. ▪ 1: Query Performance Counter. The server uses a high-resolution time type for the time stamp field of any event it writes into this channel. **Note** The timestamp is simply written into the event without any special handling. Which is to say, the server behavior does not change if a channel's Clock type is SystemTime or Query Performance Counter.
18	EvtRpcVarTypeUInt32	SIDType. One of two values: ▪ 0: The events written by the server to this channel will not include the publisher's SID. ▪ 1: The events written by the server to this channel will include the publisher's SID.
19	EvtRpcVarTypeStringArray	PublisherList. List of publishers that can raise events into the channel. This returns the same list as is returned by the **EvtRpcGetPublisherList** method, as specified in section 3.1.4.24.
20	EvtRpcVarTypeUint32	FileMax. Maximum number of log files associated with an analytic or debug channel. When the number of logs reaches the specified maximum, the system begins to overwrite the logs, beginning with the

Index	Type	Meaning
		oldest. A FileMax value of 0 or 1 indicates that only one file is associated with this channel. A FileMax of 0 is default.<36>

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST verify that the *channelPath* parameter specifies a valid channel name. The server MUST fail the method if the parameter is invalid with the error ERROR_INVALID_PARAMETER (0x00000057). The server checks if a channel name is valid by searching the given name in its channel table.

Next, the server MUST verify that the caller has read access to the information and MUST fail the method if the caller does not have read access with the error ERROR_ACCESS_DENIED (0x00000005). To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the previous checks succeed, the server MUST attempt to return the list of a channel's properties. The server MUST fill the output parameter *props* with all the properties for the channel (which are specified in the preceding *props* properties table) into an EvtRpcVariant list. The server SHOULD only fail the method if the system memory is inadequate with the error ERROR_OUTOFMEMORY (0x0000000E). The client MUST NOT interpret the values in this list. They MUST be passed uninterpreted to the higher-layer protocol or client application. For more information, see [MSDN-EVENTS].

Note that in the interval between a client calling the EvtRpcPutChannelConfig and EvtRpcAssertConfig methods, the server holds two copies of the channel properties. One copy is held in the channel table. The other copy is an encapsulated but inactive set of properties created by EvtRpcPutChannelConfig that is held in temporary storage. The server MUST return the properties held in the channel table. It MUST NOT return the encapsulated but inactive set of properties created by EvtRpcPutChannelConfig. See EvtRpcAssertConfig (section 3.1.4.29) for more information.

The server MUST NOT update its state as a result of this method, nor SHOULD the server apply any channel properties. The server SHOULD always return ERROR_SUCCESS (0x00000000) if the inputs are valid, because reading the channel properties SHOULD never fail.

The server MUST return a value indicating success or failure for this operation.

3.1.4.22 EvtRpcPutChannelConfig (Opnum 21)

The EvtRpcPutChannelConfig (Opnum 21) method is used by a client to update the configuration for a channel.

```
error_status_t EvtRpcPutChannelConfig(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in] DWORD flags,
  [in] EvtRpcVariantList* props,
  [out] RpcInfo* error
);
```

binding: An RPC binding handle as specified in section 2.2.21.

channelPath: A pointer to a string that contains a channel name (this is not a file path as the parameter name might suggest).

flags: A 32-bit unsigned integer that indicates what to do depending on the existence of the channel. This MUST be set to one of the following, and the server SHOULD return ERROR_INVALID_PARAMETER (0x00000057) if the flag is not one of the following values.<37>

Value	Meaning
0x00000000	The server MUST open the existing channel entry in its channel table or create a new entry if the specified channel is not in the table.
0x00000001	The server MUST open the existing channel entry in its channel table.
0x00000002	Always create a new channel entry in the server's channel table and delete the existing entry.
0x00000003	Only create a new channel entry in the server's channel table.

props: A pointer to an EvtRpcVariantList (section 2.2.9) structure containing channel properties, as defined in the following table.

Index	Type	Meaning
0	EvtRpcVarTypeBoolean	Enabled. If true, the channel can accept new events. If false, any attempts to write events into this channel are automatically dropped.
1	EvtRpcVarTypeUInt32	Channel Isolation. It defines the default access permissions for the channel. Three values are allowed: ▪ 0: Application. ▪ 1: System. ▪ 2: Custom. The default isolation is Application. The default permissions for Application are (shown using SDDL): ▪ L"O:BAG:SYD:" ▪ L"(A;;0xf0007;;;SY)" // local system (read, write, clear) ▪ L"(A;;0x7;;;BA)" // built-in admins (read, write, clear) ▪ L"(A;;0x7;;;SO)" // server operators (read, write, clear) ▪ L"(A;;0x3;;;IU)" // INTERACTIVE LOGON (read, write) ▪ L"(A;;0x3;;;SU)" // SERVICES LOGON (read, write) ▪ L"(A;;0x3;;;S-1-5-3)" // BATCH LOGON (read, write) ▪ L"(A;;0x3;;;S-1-5-33)" // write restricted service (read,write) ▪ L"(A;;0x1;;;S-1-5-32-573)"; // event log readers (read) The default permissions for System are (shown using SDDL): ▪ L"O:BAG:SYD:" ▪ L"(A;;0xf0007;;;SY)" // local system (read, write, clear)

Index	Type	Meaning
		▪ L"(A;;0x7;;;BA)" // built-in admins (read, write, clear) ▪ L"(A;;0x3;;;BO)" // backup operators (read, write) ▪ L"(A;;0x5;;;SO)" // server operators (read, clear) ▪ L"(A;;0x1;;;IU)" // INTERACTIVE LOGON (read) ▪ L"(A;;0x3;;;SU)" // SERVICES LOGON (read, write) ▪ L"(A;;0x1;;;S-1-5-3)" // BATCH LOGON (read) ▪ L"(A;;0x2;;;S-1-5-33)" // write restricted service (write) ▪ L"(A;;0x1;;;S-1-5-32-573)"; // event log readers (read) When the Custom value is used, the Access property will contain the defined SDDL.
2	EvtRpcVarTypeUInt32	Channel Type. One of four values: ▪ 0: Admin ▪ 1: Operational. ▪ 2: Analytic ▪ 3: Debug For more information, see [MSDN-EVTLGCHWINEVTLG].
3	EvtRpcVarTypeString	OwningPublisher. The name of the publisher that defines and registers the channel with the system.
4	EvtRpcVarTypeBoolean	ClassicEventlog. If true, the channel represents an event log created according to the EventLog Remoting Protocol, not this protocol (EventLog Remoting Protocol Version 6.0).
5	EvtRpcVarTypeString	Access. A Security Descriptor Description Language (SDDL) string, as specified in [MS-DTYP], that represents access permissions to the channels. If the **isolation** attribute is set to Application or System, the access descriptor controls read access to the file (the write permissions are ignored). If the **isolation** attribute is set to Custom, the access descriptor controls write access to the channel and read access to the file.
6	EvtRpcVarTypeBoolean	Retention. If set to true, events can never be overwritten unless explicitly cleared. This is the way to configure the logs to be circular. If set to false, events are overwritten as needed when the event log is full.
7	EvtRpcVarTypeBoolean	AutoBackup. When set to true, the event log file associated with the channel is closed as soon as it reaches the maximum size specified by the MaxSize property, and a new file is opened to accept new events. If the new file reaches maximum size, another new file will be generated and the previous new file will be backed up. The events in backed up files cannot be queried from this channel in the server unless the client specifies the backup log file names in a separate query.
8	EvtRpcVarTypeUInt64	MaxSize. The value that indicates at which point the size (in bytes) of

Index	Type	Meaning
		the event log file stops increasing. When the size is greater than or equal to this value, the file growth stops.
9	EvtRpcVarTypeString	LogFilePath. The server changes the file path to the event log file for the channel.
10	EvtRpcVarTypeUInt32	Level. Events with a level property less than or equal to this specified value are logged to the channel.
11	EvtRpcVarTypeUInt64	Keywords. Events with a keyword bit contained in the keywords bitmask set are logged to the channel.
12	EvtRpcVarTypeGuid	ControlGuid. A GUID value. The server SHOULD ignore this value.<38>
19	EvtRpcVarTypeStringArray	PublisherList. A list of publishers that can raise events into the channel.
20	EvtRpcVarTypeUInt32	FileMax. The maximum number of log files associated with an analytic or debug channel. When the number of logs reaches the specified maximum, the system begins to overwrite the logs, beginning with the oldest. A FileMax value of 0 or 1 indicates that only one file is associated with this channel. A MaxFile of 0 is the default.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to nonzero values if the error is related to a particular property. All nonzero values MUST be treated the same. If the method succeeds, the server MUST set all of the values in the structure to 0.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].<39>

In response to this request from the client, the server MUST verify that the *channelPath* parameter specifies a valid channel name. The server MUST fail the method if the parameter is invalid with the error ERROR_INVALID_PARAMETER (0x00000057). The server checks if a channel name is valid by searching the given name in its channel table.

Next, the server MUST verify that the caller has write access to the information and MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the caller does not have write access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the client specifies 0x00000000 for the *flags* value, the server MUST try to find the channel entry specified by the *channelPath* parameter in its channel table. If the server does not find the channel entry in the table, it creates a new entry with the parameter *channelPath* as the new channel name. The creation of a new channel table entry SHOULD only fail when there is inadequate memory. The server SHOULD return ERROR_OUTOFMEMORY (0x0000000E) in that case. When a new channel is created, the server SHOULD assign the default property values to the channel as in the following table.

Property	Default Value
Enabled	true
Isolation	0

Property	Default Value
ChannelType	0
OwningPublisher	null
Classic	false
Access	"O:BAG:SYD:(A;;0xf0007;;;SY)(A;;0x7;;;BA)(A;;0x7;;;SO)"(A;;0x3;;;IU)(A;;0x3;;;SU)(A;;0x3;;;S-1-5-3)(A;;0x3;;;S-1-5-33)(A;;0x1;;;S-1-5-32-573)"
Retention	false
Autobackup	false
maxSize	20 * 1024 * 1024
LogFilePath	%systemfolder%\winevt\<channelname>.evtx
Level	0
Keywords	0xFFFFFFFFFFFFFFFF
ControlGuid	{00000000-0000-0000-0000-000000000000}
BufferSize	64K
MinBuffers	2 * Number of the CPUs.
MaxBuffers	22 + MinBuffers
Latency	1
ClockType	0
SidType	1
PublisherList	null
FileMax	0

If the client specifies 0x00000001 for the *flags* value and the specified channel entry is not found in the channel table, the server MUST return ERROR_NOT_FOUND (0x00000490).

If the client specifies 0x00000002 for the *flags* value, the processing rule is similar to processing rule 1 except that when the server finds the exiting channel entry, it SHOULD delete it first before creating a new one. Delete an existing entry SHOULD never fail.

If the client specifies the 0x00000003 for the *flags* value, the server MUST fail the method if the specified channel is already in the channel table with the error code ERROR_ALREADY_EXISTS (0x000000B7).

The server SHOULD check if the publisher specified has already registered in its publisher table when the client tries to update the owning publisher property. If not, the server SHOULD fail the method with ERROR_INVALID_PARAMETER (0x00000057).<40>

Note The configuration properties for *BufferSize*, *MinBuffers*, *MaxBuffers*, *Latency*, *ClockType*, and *SIDType* can't be updated by the client. These properties are maintained by the server administrator on the physical machine only and cannot be updated through the remote protocol methods. The server administrator can specify these properties with any allowable values.<41> The server SHOULD make sure the client does not update these properties. The server SHOULD fail the method with the error ERROR_INVALID_OPERATION (0x000010DD) in this case.

If the previous checks succeed, the server MUST attempt to update the channel's properties using the value specified in the *props* parameter. The server SHOULD proceed in the following manner to update the data for each channel property:

The server SHOULD allocate a memory block with the same size as the EvtRpcVariantList (section 2.2.9) pointed to by the *props* parameter. If the memory allocation fails, the server SHOULD return ERROR_OUTOFMEMORY (0x0000000E). The server then copies all the data in the *props* parameter into the new allocated memory block. Before copying the data, the server SHOULD validate the data as follows:

- The Isolation property SHOULD be either 0, 1, or 2, if the client has specified that property.
- The Channel type property SHOULD be either 0, 1, 2, or 3, if the client has specified that property.
- The Access property string SHOULD be a valid security descriptor as specified in section [MS-DTYP], if the client specifies that property. Note that the only access permissions defined for channels are read, write, and clear; if a client attempts to specify any other access permissions in the security descriptor, the server SHOULD ignore them.
- The LogFilePath property MUST be a valid file path string for the server's file system, if the client specifies that property.
- The server SHOULD verify that the publishers specified in the PublisherList property exist in the server's publisher table. If so, the server SHOULD add the current channel to the publisher entries in the server's publisher table for every publisher specified by the PublisherList property so that as soon as the new settings are applied, the server is prepared for those publishers to publish events to this channel.

If any of the validation checks fail, the server SHOULD return ERROR_INVALID_DATA. After copying the data, the server SHOULD return to the client with ERROR_SUCCESS (0x00000000), but SHOULD NOT apply the new channel properties until EvtRpcAssertConfig is called or the server restarts. **EvtRpcAssertConfig** causes the server to apply an in-memory copy of the channel configuration, whereas when the server restarts, it loads channel configuration data from persistent storage as specified in section 3.1.1.5. Before applying the properties, all the server's behaviors are still the same as they were originally, even after the method has successfully returned to the client. For information on the server saving the configuration and then applying the changes with the **EvtRpcAssertConfig** method, see the processing rules in EvtRpcAssertConfig (section 3.1.4.29). The server SHOULD check if the value passed by the client is within the allowed range. If not, the server SHOULD return ERROR_INVALID_PARAMETER (0x00000057). The server will not fail the method if all the previous checks are passed.

The server MUST return a value indicating success or failure for this operation.

3.1.4.23 EvtRpcGetPublisherList(Opnum 22)

The EvtRpcGetPublisherList (Opnum 22) method is used by a client to get the list of publishers.

```
error_status_t EvtRpcGetPublisherList(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in] DWORD flags,
  [out] DWORD* numPublisherIds,
  [out, size_is(,*numPublisherIds), range(0, MAX_RPC_PUBLISHER_COUNT), string]
    LPWSTR** publisherIds
);
```

binding: An RPC binding handle as specified in section 2.2.21.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<42>

numPublisherIds: A pointer to a 32-bit unsigned integer that contains the number of publisher names.

publisherIds: A pointer to an array of strings that contain publisher names.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST verify that the caller has read access to the publisher table and MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the caller does not have read access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then the server MAY get the security descriptor of the publisher table. The server MAY assign a security descriptor when the publisher table is created or if the publisher table is built on the server's file system, it can get its security descriptor from the file system.<43> Next, the server SHOULD be able to perform the read access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the above check succeeds, the server MUST go to its publisher table and read all the publisher names and use the results to fill the *publisherIds* parameter. The server MUST also set the *numberPublisherIds* parameter value to be the number of the publisher names it returns. The server SHOULD only fail if it has inadequate memory to allocate for the *publisherIds* parameter to copy all the publisher names from its publisher table into the buffer that is pointed to by *publisherIds*. In that case, the server SHOULD return ERROR_OUTOFMEMORY (0x0000000E).

The server MUST return a value indicating success or failure for this operation.

3.1.4.24 EvtRpcGetPublisherListForChannel (Opnum 23)

The EvtRpcGetPublisherListForChannel (Opnum 23) method is used by a client to get the list of publishers that write events to a particular channel.

```
error_status_t EvtRpcGetPublisherListForChannel(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in] LPCWSTR channelName,
  [in] DWORD flags,
  [out] DWORD* numPublisherIds,
  [out, size_is(,*numPublisherIds), range(0, MAX_RPC_PUBLISHER_COUNT), string]
    LPWSTR** publisherIds
);
```

binding: An RPC binding handle as specified in section 2.2.21.

channelName: A pointer to a string that contains the name of the channel for which the publisher list is needed.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<44>

numPublisherIds: A pointer to a 32-bit unsigned integer that contains the number of publishers that are registered and that can write to the log.

publisherIds: A pointer to an array of strings that contain publisher names.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST verify that the *channelName* parameter specifies a correct channel name. The server MUST fail the method if the *channelName* parameter is

invalid with the error ERROR_INVALID_PARAMETER (0x00000057). The server checks if a channel name is valid by searching the given name in its channel table.

Next, the server MUST verify that the caller has read access to the channel and MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the caller does not have read access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the read access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the previous checks succeed, the server MUST attempt to return a list of publishers for the channel specified by the *channelName* parameter. In order to do this, the server searches all the publisher entries in its publisher table. For each publisher, the server checks if the publisher declares that it will generate events to the given channel. If that is true, the server adds this publisher to the result parameter *publisherIds* and the *numPublisherIds* (initialized as 0) is increased by 1. The server SHOULD only fail when not enough memory space can be allocated to copy the matched publisher names into the *publisherIds* parameter. In that case, the server SHOULD return ERROR_OUTOFMEMORY (0x0000000E). The server MUST NOT update its state.

The server MUST return a value indicating success or failure for this operation.

3.1.4.25 EvtRpcGetPublisherMetadata (Opnum 24)

The EvtRpcGetPublisherMetadata (Opnum 24) method is used by a client to open a handle to publisher metadata. It also gets some initial information from the metadata.

```
error_status_t EvtRpcGetPublisherMetadata(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, unique, range(0, MAX_RPC_PUBLISHER_ID_LENGTH), string]
    LPCWSTR publisherId,
  [in, unique, range(0, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR logFilePath,
  [in] LCID locale,
  [in] DWORD flags,
  [out] EvtRpcVariantList* pubMetadataProps,
  [out, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA* pubMetadata
);
```

binding: An RPC binding handle as specified in section 2.2.21.

publisherId: A pointer to a string that contains the publisher for which information is needed.

logFilePath: A pointer to a null string that MUST be ignored on receipt.

locale: A Locale value, as specified in [MS-GPSI]. This is used later if the *pubMetadata* handle is used for rendering.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<45>

pubMetadataProps: A pointer to an EvtRpcVariantList (section 2.2.9) structure containing publisher properties.

pubMetadata: A pointer to a publisher handle. This parameter is an RPC context handle, as specified in [C706], Context Handles. For information on handle security and authentication considerations, see sections 2.2.20 and 5.1.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST verify that the *publisherID* parameter specifies either a publisher name or NULL. The server MUST fail the method if the *publisherID* is non-NULL and is not the name of a publisher with the error code ERROR_INVALID_PARAMETER (0x00000057). The server SHOULD check whether the non-NULL *publisherID* is in the publisher table to verify whether the *publisherID* is a publisher name. If the *publisherID* parameter is NULL, the server MUST use the default publisher (as specified in section 3.1.1.14).

Next, the server MUST verify that the caller has read access to the information and MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the caller does not have read access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then the server MAY get the security descriptor of the publisher table. The server MAY assign a security descriptor when the publisher is registered in its publisher table or if the publisher entry is built on the server's file system, it can get its security descriptor from the file system.<46> Next, the server SHOULD be able to perform the read access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2).

If the previous checks succeed, the server MUST attempt to create a CONTEXT_HANDLE_PUBLISHER_METADATA for the publisher. As specified in section 3.1.1.11, the server SHOULD create a publisher metadata object and then cast it to the context and assign the handle to the *pubMetadata* parameter. The server SHOULD add the newly created handle to its handle table in order to track it. The server SHOULD only fail the creation of handles in the case of inadequate memory and return ERROR_OUTOFMEMORY (0x0000000E). The server MUST store the *locale* value in the publisher metadata object.

The server MUST fill an EvtRpcVariantList (for more information, see section 2.2.9) that contains 29 EvtRpcVariants and save them in the *pubMetadataProps* parameter. As noted in the *pubMetadataProps* description, not all of the EvtRpcVariant entries are actually used, and all unused ones MUST be set to type EvtRpcVarTypeNULL. The following table lists those entries that are used.

Index	Type	Description
0	EvtVarTypeGuid	PublisherGuid: This is the identifier of the publisher which is mentioned in section 3.1.1.2.
1	EvtVarTypeString	ResourceFilePath: This is the publisher resource file path which is specified in section 3.1.1.14.
2	EvtVarTypeString	ParameterFilePath: This is the publisher parameter file which is specified in section 3.1.1.14.
3	EvtVarTypeString	MessageFilePath: This is the publisher message file which is specified in section 3.1.1.14.
7	EvtVarTypeStringArray	ChannelReferencePath: This is the array of the channel paths into which the publisher generates events.
8	EvtVarTypeUInt32Array	ChannelReferenceIndex: The channel start index values, as specified in section 3.1.1.2.
9	EvtVarTypeUInt32Array	ChannelReferenceID: The channel reference ID values, as specified in section 3.1.1.2.
10	EvtVarTypeUInt32Array	ChannelReferenceFlags: The channel reference flags, as specified in section 3.1.1.2.
11	EvtVarTypeUInt32Array	ChannelReferenceMessageID: This is the message Ids for the channels.

As specified earlier in this section, the server SHOULD find the publisher entry in its publisher table based on the specified publisherId parameter from the client. Once the server locates the publisher entry, the server SHOULD get the publisherGUID, ResourceFilePath, ParameterFilePath, MessageFilePath, ChannelReferenceIndex, ChannelReferenceID and ChannelReferenceFlags directly from the publisher table entry (as specified in sections 3.1.1.3 for publisher tables and 3.1.1.2 for publishers). Then the server SHOULD open the publisher resource file and locate the channel information in the resource file (as specified in section 3.1.1.14). Next, the server reads the channel name strings and channel message IDs, returning them as the result for the ChannelReferencePath and ChannelReferenceMessageID. See section 4.12 for an example EvtRpcVariantList containing all 29 EvtRpcVariants.

If the server can't allocate enough memory for the returning parameter *pubMetadataProps*, it SHOULD return the error ERROR_OUTOFMEMORY (0x0000000E).

The server MUST return a value indicating success or failure for this operation.

If the server can't find the corresponding information from the publisher resource, the server SHOULD set the entry to *EvtRpcVarTypeNULL* to indicate some of the values are not retrieved, but SHOULD still proceed with the other eleven data fields without returning any error. Even if all of the eleven fields are all not found, the server SHOULD still return ERROR_SUCCESS (0x00000000).

3.1.4.26 EvtRpcGetPublisherResourceMetadata (Opnum 25)

The EvtRpcGetPublisherResourceMetadata (Opnum 25) method obtains information from the publisher metadata.

```
error_status_t EvtRpcGetPublisherResourceMetadata(
  [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA handle,
  [in] DWORD propertyId,
  [in] DWORD flags,
  [out] EvtRpcVariantList* pubMetadataProps
);
```

handle: A handle to an event log. This handle is returned by the EvtRpcGetPublisherMetadata (Opnum 24) method. This parameter is an RPC context handle, as specified in [C706], Context Handles.

propertyId: Type of information as specified in the following table.

Value	Meaning
0x00000004	Publisher help link.
0x00000005	Publisher friendly name.
0x0000000C	Level information.
0x00000010	Task information.
0x00000015	Opcode information.
0x00000019	Keyword information.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<47>

pubMetadataProps: Pointer to an EvtRpcVariantList (section 2.2.9) structure. This list MUST contain multiple entries.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the context handle value that it creates in the **EvtRpcGetPublisherMetadata** method (as specified in section 3.1.4.25) in its handle table and compare it with the handle passed here to perform that check.<48> The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid.

The server MUST return an error if *propertyID* is anything other than 0x00000004, 0x00000005, 0x0000000C, 0x00000010, 0x00000015, or 0x00000019.

If all the above checks succeed, the server MUST attempt to return a list of properties for the publisher specified by the handle. If the publisher does not have metadata, this method SHOULD fail with the error ERROR_INVALID_DATA (0x0000000D).<49> Note that any one publisher that does have metadata can optionally specify only a subset of the metadata described herein. For example, not all publishers with metadata specify help links or keywords. For those cases, the server MUST return ERROR_SUCCESS (0x00000000) along with a complete EvtRpcVariantList having the corresponding entries set to EvtVarTypeNull.<50>

The EvtRpcVariantList (for more information, see section 2.2.9) MUST contain 29 EvtRpcVariants whenever this function returns success. As indicated below, not all of those EvtRpcVariant entries are used, and all unused entries MUST be set to EvtVarTypeNull.

The set of entries used depends on the value specified by the *propertyID* parameter. For the sake of brevity, the unused entries are not shown.

Note The indexes referenced below are 0-based; for example, index 4 refers to the fifth variant that is returned in the EvtRpcVariantList.

When *propertyID* = 0x00000004, the following entries MUST be set in *pubMetadataProps*.

To do this, the server SHOULD get the helperlink string from the publisher resource file (as specified in section 3.1.1.14).

Index	Type	Description
4	EvtVarTypeString	HelpLink

When *propertyID* = 0x00000005, the following entries MUST be set in *pubMetadataProps*.

To do this, the server gets the *messageId* of the publisher name from the resource file (as specified in the section 3.1.1.14).

Index	Type	Description
5	EvtVarTypeUInt32	PublisherMessageID

When *propertyID* = 0x0000000C, the following entries MUST be set in *pubMetadataProps*.

To do this, the server SHOULD get all the levels' names, values, and *messageId* and pack them into the array from the publisher resource file (as specified in the section 3.1.1.14).

Index	Type	Description
13	EvtVarTypeStringArray	LevelName
14	EvtVarTypeUInt32Array	LevelValue

Index	Type	Description
15	EvtVarTypeUInt32Array	LevelMessageID

When *propertyID* = 0x00000010, the following entries MUST be set in *pubMetadataProps*.

To do this, the server SHOULD get all the tasks' names, values, and *messageId*s and pack them into the array from the publisher resource file (as specified in the section 3.1.1.14).

Index	Type	Description
17	EvtVarTypeStringArray	TaskName
18	EvtVarTypeGuidArray	TaskEventGuid
19	EvtVarTypeUInt32Array	TaskValue
20	EvtVarTypeUInt32Array	TaskMessageID

When *propertyID* = 0x00000015, the following entries MUST be set in *pubMetadataProps*.

To do this, the server SHOULD get all the Opcodes' names, values, and *messageId*s and pack them into the array from the publisher resource file (as specified in the section 3.1.1.14).

Index	Type	Description
22	EvtVarTypeStringArray	OpcodeName
23	EvtVarTypeUInt32Array	OpcodeValue
24	EvtVarTypeUInt32Array	OpcodeMessageID

When *propertyID* = 0x00000019, the following entries MUST be set in *pubMetadataProps*.

To do this, the server SHOULD get all the Keywords' names, values, and *messageId*s and pack them into the array from the publisher resource file (as specified in the section 3.1.1.14).

Index	Type	Description
26	EvtVarTypeStringArray	KeywordName
27	EvtVarTypeUInt64Array	KeywordValue
28	EvtVarTypeUInt32Array	KeywordMessageID

The server MUST NOT update its state.

The server MUST return a value indicating success or failure for this operation.

3.1.4.27 EvtRpcGetEventMetadataEnum (Opnum 26)

The EvtRpcGetEventMetadataEnum (Opnum 26) method obtains a handle for enumerating a publisher's event metadata.

```
error_status_t EvtRpcGetEventMetadataEnum(
  [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA pubMetadata,
  [in] DWORD flags,
  [in, unique, range(0, MAX_RPC_FILTER_LENGTH), string]
    LPCWSTR reservedForFilter,
```

```
  [out, context_handle] PCONTEXT_HANDLE_EVENT_METADATA_ENUM* eventMetaDataEnum
);
```

pubMetadata: This parameter is an RPC context handle, as specified in [C706], Context Handles. For information on handle security and authentication considerations, see sections 2.2.20 and 5.1.

flags: A 32-bit unsigned integer that MUST be set to zero when sent and MAY be ignored on receipt.<51>

reservedForFilter: A pointer to a null string that MUST be ignored on receipt.

eventMetaDataEnum: A pointer to an event numeration handle. This parameter is an RPC context handle, as specified in [C706], Context Handles.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the context handle value it creates in the **EvtRpcGetPublisherMetadata** method (as specified in section 3.1.4.25) in its handle table and compare it with the handle passed here to perform the check.<52> The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid.

If the previous check succeeds, the server MUST attempt to create an event metadata object and assign it to the *eventMetadataEnum* parameter after casting to a CONTEXT_HANDLE_EVENT_METADATA_ENUM handle (see section 3.1.1.11 for the content of an event metadata object). The server SHOULD add the newly created handle to its handle table in order to track it. If the previous check fails, the server MUST NOT create the context handle or add it to the handle table. Creating the context handle SHOULD only fail due to a shortage of memory, in which case the server SHOULD return ERROR_OUTOFMEMORY (0x0000000E).

After the server creates the event metadata object, it SHOULD preload the **EventsMetaData** field for the metadata object. First, the server SHOULD cast the pubMetadata context handle into the publisher metadata object and then read out the *ResourceFile* value. Next, the server SHOULD open the resource file and find the events information section (as specified in section 3.1.1.14). The server SHOULD read all the events information into memory and assign the start address to the **EventsMetaData** field and then set the **Enumerator** field to 0.

The server MUST return a value indicating success or failure for this operation.

3.1.4.28 EvtRpcGetNextEventMetadata (Opnum 27)

The EvtRpcGetNextEventMetadata (Opnum 27) method gets details about a possible event and also returns the next event metadata in the enumeration. It is used to enumerate through the event definitions for the publisher associated with the handle. The enumeration is in the forward direction only, and there is no reset functionality.

```
error_status_t EvtRpcGetNextEventMetadata(
  [in, context_handle] PCONTEXT_HANDLE_EVENT_METADATA_ENUM eventMetaDataEnum,
  [in] DWORD flags,
  [in] DWORD numRequested,
  [out] DWORD* numReturned,
  [out, size_is(,*numReturned), range(0, MAX_RPC_EVENT_METADATA_COUNT)]
    EvtRpcVariantList** eventMetadataInstances
);
```

eventMetaDataEnum: A handle to an event metadata enumerator. This parameter is an RPC context handle, as specified in [C706], Context Handles. For information on handle security and

authentication considerations, see sections 2.2.20 and 5.1. This is the value which comes from the return parameter *eventMetaDataEnum* of function **EvtRpcGetEventMetadataEnum** (as specified in 3.1.4.27).

flags: A 32-bit unsigned integer that MUST be set to 0x00000000 when sent and MAY be ignored on receipt.<53>

numRequested: A 32-bit unsigned integer that contains the number of events for which the properties are needed.

numReturned: A pointer to a 32-bit unsigned integer that contains the number of events for which the properties are retrieved.

eventMetadataInstances: A pointer to an array of EvtRpcVariantList (section 2.2.9) structures.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the context handle value that it creates in the **EvtRpcGetPublisherMetadata** method (as specified in section 3.1.4.25) in its handle table and compare it with the handle passed here to perform the check.<54> The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid.

The server MUST verify that event metadata entries remain in the publisher metadata that have not yet been enumerated. As specified in section 3.1.1.11, the event metadata context handle (*eventsMetaDataEnum* parameter) is an event metadata object in the server. The server SHOULD cast the context handle into the event metadata object. In the object, the **Enumerator** field tracks the delivered entries and remaining entries. If the enumeration has already returned the metadata for every event, the method SHOULD fail with the error ERROR_NO_DATA (0x000000E8).<55> Note that it is acceptable for a publisher to have no event metadata entries. In this case, the server MUST respond to the first call to EvtRpcGetNextEventMetadata with a return code of ERROR_SUCCESS (0x00000000) with numReturned set to zero. In particular, Windows event publishers that use the legacy protocol documented in [MS-EVEN] will not have event metadata associated with them. These include, but are not limited to, the events reported in the Application, System, and Security logs.

If the preceding checks succeed, the server MUST attempt to return the metadata for as many events as are specified in the *numRequested*, or until all the event metadata has been returned.

The server MUST fill an array of EvtRpcVariantList (section 2.2.9) objects, with an EvtRpcVariantList for each event's metadata, and assign the array to the *eventMetadataInstances* parameter. The server SHOULD only fail in creation of the array EvtRpcVariantList due to shortness of memory. In that case, the server SHOULD return ERROR_OUTOFMEMORY (0x000000E). Each EvtRpcVariantList MUST contain the following nine EvtVariant entries.

Index	Type	Description
0	EvtVarTypeUInt32	Event identifier
1	EvtVarTypeUInt32	Version
2	EvtVarTypeUInt32	Channel identifier
3	EvtVarTypeUInt32	Level value of event
4	EvtVarTypeUInt32	Opcode value of event
5	EvtVarTypeUInt32	Task value of event
6	EvtVarTypeUInt32	Keyword value of event

Index	Type	Description
7	EvtVarTypeUInt64	MessageID for event description string
8	EvtVarTypeString	Event definition template

The preceding nine entries SHOULD be retrieved from the event information section in the publisher resource file (as specified in section 3.1.1.14).

If the preceding checks succeed and the server successfully creates the array of **EvtRpcVariantList** objects, the server MUST update the cursor value in the event metadata object to keep track of the event metadata that has already been enumerated. If the checks fail, or if the server is unable to create the array, the server MUST NOT update anything.

The server MUST return a value indicating success or failure for this operation.

3.1.4.29 EvtRpcAssertConfig (Opnum 15)

The EvtRpcAssertConfig (Opnum 15) method indicates to the server that the publisher or channel configuration has been updated.

```
error_status_t EvtRpcAssertConfig(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR path,
  [in] DWORD flags
);
```

binding: An RPC binding handle as specified in section 2.2.21.

path: A pointer to a string that contains a channel or publisher name to be updated.

flags: The client MUST specify exactly one of the following.

Value	Meaning
EvtRpcChannelPath 0x00000000	*Path* specifies a channel name.
EvtRpcPublisherName 0x00000001	*Path* specifies a publisher name.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server SHOULD first validate the *path* parameter. The server MUST interpret the *path* parameter as a channel name if the *flags* parameter is equal to 0x00000000. The server SHOULD try to determine if the specified channel name has been already registered in its channel table (as specified in section 3.1.1.5). If the flags value is 0x00000001, the server MUST interpret *path* as a publisher name. The server SHOULD then check if the publisher has been registered in its publisher table (as specified in section 3.1.1.3). The server SHOULD fail the operation if the validation of *path* fails. The server MAY<56> return the error ERROR_INVALID_PARAMETER (0x00000057) to indicate such failure.

Next, the server MUST verify that the caller has write access to the information and MUST fail the method if the caller does not have write access with the error ERROR_ACCESS_DENIED (0x00000005). To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check

is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2). If the access property is not present for the channel, the channel gets a default SDDL which is "O:BAG:SYD:(A;;0xf0005;;;SY)(A;;0x5;;;BA)(A;;0x1;;;S-1-5-32-573)".

The server assumes that something has changed some properties of a publisher or channel before processing this call. Such changes can be made by the **EvtRpcPutChannelConfig** method (as specified in section 3.1.4.22). The changes made by **EvtRpcPutChannelConfig** only create an in-memory copy of the new properties of the publisher or channel, but do not impact the server's immediate behavior. **EvtRpcAssertConfig** puts the in-memory copy created by **EvtRpcPutChannelConfig** into effect. When applying the changes from the in-memory copy, the server SHOULD check the potential conflicts of the new settings with existing settings. The following is a list of checks the server SHOULD make before accepting the changes:

1. If the channel type is set, the server SHOULD check whether the new value is one of the four allowed values which are specified in section 3.1.4.22.

2. If the owning publisher is set, the server SHOULD check whether the publisher exists and also check whether the channel is owned by another publisher. The server SHOULD go through its channel table and make sure the **OwningPublisher** field points to a different publisher's name and no two channels have the same publisher name.

After that, the server activates the new properties of the channel or the publisher based on the latest settings and frees the memory associated with the in-memory copy of the new channel or publisher properties allocated during the call to **EvtRpcPutChannelConfig**.

In the interval between the calls to **EvtRpcPutChannelConfig** and **EvtRpcAssertConfig**, there are effectively two copies of the channel or publisher properties. One copy is the encapsulated but inactive set of properties created by **EvtRpcPutChannelConfig**, waiting to be made active. The other copy is the unencapsulated set of active property values that are stored in individual run-time variables within the implementation. It is an implementation-specific detail whether these variables exist in a form that resembles the encapsulated representation in terms of memory layout or other matters not germane to the semantics of the properties. The encapsulated property set is by definition ephemeral, serving only as a temporary holding vessel for the updated channel or publisher properties until the **EvtRpcAssertConfig** method is called.

When this method is called, the server activates the new properties of the channel or publisher in a two-stage process. First, the server stores the values in the encapsulated property set into whatever persistent storage the server uses to store channel and publisher properties between invocations of the server itself. Second, the server loads the new property values from persistent storage, using them to update the unencapsulated, active variables that implement the properties at run time. In this way, the server simultaneously puts the new values into effect and ensures that those values will remain in effect if the server is restarted.

When this two-stage process is complete, the encapsulated property set serves no further purpose and the server frees the memory associated with it.

Note that this protocol does not include a method for changing publisher configuration data. The client SHOULD provide this functionality if it wants to call this function specifying the publisher. For more information, see section 3.2.7.

The configuration properties of the channels and publishers SHOULD include the following:

- Physical location of the publisher's event definition binaries
- Channel security settings
- Channel and publisher names

- Enabled/disabled state or any other implementation-dependent configurable settings

The server MUST NOT change those states until this method is called.

The server MUST return a value indicating success or failure for this operation.

3.1.4.30 EvtRpcRetractConfig (Opnum 16)

The EvtRpcRetractConfig (Opnum 16) method indicates to the server that the publisher or channel is to be removed.

```
error_status_t EvtRpcRetractConfig(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR path,
  [in] DWORD flags
);
```

binding: An RPC binding handle as specified in section 2.2.21.

path: A pointer to a string that contains a channel or publisher name to be removed.

flags: A 32-bit unsigned integer that indicates how the path parameter is to be interpreted. This MUST be set as follows.

Value	Meaning
EvtRpcChannelPath 0x00000000	Path specifies a channel name.
EvtRpcPublisherName 0x00000001	Path specifies a publisher name.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server SHOULD first validate the *path* parameter.<57> The server MUST interpret the *path* parameter as a channel name if the flags parameter is equal to 0x00000000. The server SHOULD try to find if the specified channel name has been already registered in its channel table (as specified in section 3.1.1.5). If the flags value is 0x00000001, the server MUST interpret *path* as a publisher name. The server SHOULD then check if the publisher has been registered in its publisher table (as specified in section 3.1.1.3). The server SHOULD fail the operation if the validation of *path* fails. The server MAY return the error ERROR_INVALID_PARAMETER (0x00000057) to indicate such failure.<58>

Next, the server MUST verify that the caller has delete access to the information and MUST fail the method with the error ERROR_ACCESS_DENIED (0x00000005) if the caller does not have delete access. To perform the access check, the server SHOULD first determine the identity of the caller. Information determining the identity of the caller for the purpose of performing an access check is specified in [MS-RPCE] section 3.2.3.4.2. Then, if the client specifies a channel, the server SHOULD read the channel's access property (as specified in section 3.1.4.21) as the security descriptor string. Next, the server SHOULD be able to perform the write and clear access check using the Access Check algorithm (as specified in [MS-DTYP] section 2.5.3.2). If the access property is not present for the channel, the channel gets a default SDDL, which is "O:BAG:SYD:(A;;0xf0005;;;SY)(A;;0x5;;;BA)(A;;0x1;;;S-1-5-32-573)".

If the above checks succeed, the server MUST delete the publisher entry from its publisher table or delete the channel from the channel table. Operations like deleting entries from the table SHOULD always be successful.

Any information in the channel table and publisher table MUST not be removed until this method is called.

The server MUST return a value indicating success or failure for this operation.

3.1.4.31 EvtRpcMessageRender (Opnum 9)

The EvtRpcMessageRender (Opnum 9) method is used by a client to get localized descriptive strings for an event.

```
error_status_t EvtRpcMessageRender(
  [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA pubCfgObj,
  [in, range(1, MAX_RPC_EVENT_ID_SIZE)]
    DWORD sizeEventId,
  [in, size_is(sizeEventId)] BYTE* eventId,
  [in] DWORD messageId,
  [in] EvtRpcVariantList* values,
  [in] DWORD flags,
  [in] DWORD maxSizeString,
  [out] DWORD* actualSizeString,
  [out] DWORD* neededSizeString,
  [out, size_is(,*actualSizeString), range(0, MAX_RPC_RENDERED_STRING_SIZE)]
    BYTE** string,
  [out] RpcInfo* error
);
```

pubCfgObj: A handle to a publisher object. This parameter is an RPC context handle, as specified in [C706], Context Handles. This value comes from the return parameter *pubMetadata* of the function EvtRpcGetPublisherMetadata (section 3.1.4.25).

sizeEventId: A 32-bit unsigned integer that contains the size, in bytes, of the data in the *eventId* field. The server MUST ignore this value if EvtFormatMessageId is specified as the *flags* parameter. If EvtFormatMessageId is not specified in the *flags* parameter, the server MUST use the *sizeEventId* parameter and ignore the *messageId* parameter.

eventId: A pointer to an **EVENT_DESCRIPTOR** structure, as specified in [MS-DTYP] section 2.3.1.

messageId: A 32-bit unsigned integer that specifies the required message. This is an alternative to using the *eventID* parameter used by a client application that has obtained the value through some method outside those documented by this protocol. The server MUST ignore this value unless the *flags* value is set to EvtFormatMessageId; in which case, the server MUST use this value to determine the required message and ignore the *eventID* parameter.

values: An array of strings used as substitution values for event description strings. The number of strings submitted is determined by the number of description strings contained in the event message specified by the *eventID* or *messageId* parameter.<59>

flags: For all options except EvtFormatMessageId, the *eventId* parameter is used to specify an event descriptor. For the EvtFormatMessageId option, the *messageId* is used for locating the message. This MUST be set to one of the values in the following table, which indicates the action a server is requested to perform.

Value	Meaning
EvtFormatMessageEvent	Locate the message for the event that corresponds to *eventID*, and then insert the values specified by the values parameter.

Value	Meaning
0x00000001	
EvtFormatMessageLevel 0x00000002	Extract the **level** field from *eventID*, and then return the localized name for that level.
EvtFormatMessageTask 0x00000003	Extract the **task** field from *eventID*, and then return the localized name for that task.
EvtFormatMessageOpcode 0x00000004	Extract the **opcode** field from *eventID*, and then return the localized name for that opcode.
EvtFormatMessageKeyword 0x00000005	Extract the **keyword** field from *eventID*, and then return the localized name for that keyword.
EvtFormatMessageChannel 0x00000006	Extract the **channel** field from *eventID*, and then return the localized name for that channel.
EvtFormatMessageProvider 0x00000007	Return the localized name of the publisher.
EvtFormatMessageId 0x00000008	Locate the message for the event corresponding to the *messageId* parameter, and then insert the values specified by the values parameter.

maxSizeString: A 32-bit unsigned integer that contains the size, in bytes, of the string that is provided by the caller.

actualSizeString: A pointer to a 32-bit unsigned integer that, on return, contains the actual size, in bytes, of the resulting description (including null termination).

neededSizeString: A pointer to a 32-bit unsigned integer that, on return, contains the needed size, in bytes (including null termination).

string: A pointer to a bytearray that, on return, contains a localized string containing the message requested. This can contain a simple string, such as the localized name of a keyword, or a fully rendered message that contains multiple inserts.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to nonzero values if the error is related to loading the necessary resource. All nonzero values MUST be treated the same. If the method succeeds, the server MUST set all the fields in the structure to 0.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success. The method MUST return ERROR_INSUFFICIENT_BUFFER (0x0000007A) if *maxSizeString* is too small to hold the result string. In that case, *neededSizeString* MUST be set to the necessary size. Otherwise, the method MUST return a different implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the context handle value it creates in the **EvtRpcGetPublisherList** method (as specified in section 3.1.4.25) in its handle table (as specified in 3.1.1.12) and compare it with the handle passed here to perform the handle validation work.<60> The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid or there is no such handle on the server.

The server MUST then check the *flags* parameter. If *flags* is not one of the 8 values in the flags table in this section, the server MUST fail the method with the error ERROR_INAVLID_PARAMETER(0x00000057).

The server MUST ignore the *eventId* parameter if **EvtFormatMessageId** is specified as the flags parameter. If **EvtFormatMessageId** is not specified in the *flags* parameter, the server MUST use *eventId* parameter and ignore the *messageId* parameter.

If validation is successful, the server SHOULD cast the pubCfgObj context handle into the publisher metadata object. The publisher metadata object contains the ResourceFileHandle which the server SHOULD use to read the publisher resource information (as specified in section 3.1.1.14) to serve this method. The server MUST attempt to return a localized string. If the string being requested is for the level, task, opcode, and keyword choices, the server MUST use its own localized string table if the value is within the range of the server (the value is retrieved from the *eventId* parameter). The server MUST define range values as the following:

- Levels 0 through 15
- Task 0
- Opcodes 0 through 9, and 240
- The following keywords for levels 0 through 15.
  - 0
  - 0x1000000000000
  - 0x2000000000000
  - 0x4000000000000
  - 0x8000000000000
  - 0x10000000000000
  - 0x20000000000000
  - 0x40000000000000
  - 0x80000000000000.

For example, if the level requested is 2, the server's list of strings for levels MUST be used since 2 is in the range [0,15]. The server MUST NOT change any state.

The server MUST prepare the description strings for these defined values itself. There are no formal recommendations about what strings to assign for these reserved values. The server can assign any strings for the keywords for the levels 0 through 15 values or it can assign part of them for its own development purpose and leave the remaining as dummy strings. The assigned strings MUST be kept by the server in dedicated files. The dedicated files which keep the predefined strings are the server's own localized string tables. Thus the server can be a default publisher. It maintains these predefined strings for every language. The server can then have multiple language-specific resource files and each language-specific resource file contains all the predefined strings for one language. For more information on language-specific resource files, see [MSDN-MUIResrcMgmt].

When the EvtFormatMessageId is specified in the *flags* parameter, the server SHOULD use the *messageId* parameter and search through the language-specific resource file (as specified in section 3.1.1.13) to find the *messageId* that the client specified. Once the server locates the *messageId* in the language-specific resource file, it gets the localized string associated with that *messageId* and returns the result to the client.<61>

When the EvtFormatMessageId is not specified in the *flags* parameter, the server SHOULD first use the *eventId* parameter to find the *messageId* in the publisher resource file. Depending on the *flags* value, processing is as follows:

- If EvtFormatMessageEvent is specified in the *flags* parameter, the server SHOULD search the events information (as specified in section 3.1.1.14) in the publisher resource file to get the *messageId* for that event and then get the event description string from the language-specific resource file using the *messageId*.

- If EvtFormatMessageLevel, EvtFormatMessageTask, EvtFormatMessageOpcode, or EvtFormatMessageKeyword is specified in the *flags* parameter, the server SHOULD first get the event information based on the *eventId* and then locate the level *messageId*, task *messageId*, opcode *messageId*, or the keyword *messageId* for that event based on the *flags* value. Next, it uses the *messageId* to get the description string from the language-specific resource file.

- If EvtFormatMessageProvider is specified in the *flags* parameter, the server SHOULD first get the events information (as specified in section 3.1.1.14) based on the *eventId*. Next, it SHOULD search the publisher information (as specified in the section 3.1.1.14) in the publisher resource file to get the *messageId* for that publisher name based on the publisher identifier it gets from the first step and then get the publisher name string from the language-specific resource file using the *messageId*.

- If EvtFormatMessageChannel is specified in the *flags* parameter, the server SHOULD first get the events information (as specified in section 3.1.1.14) based on the eventId. Next, it SHOULD search the channel information (as specified in section 3.1.1.14) in the publisher resource file to get the *messageId* parameter for that channel name based on the publisher identifier it gets from the first step and then get the localized channel name string from the language-specific resource file using the *messageId* parameter.

The message string that the server gets is from the publisher localized string table on the server. Because the publisher object contains the locale value that the client requires when opening the publisher through the EvtRpcGetPublisherMetadata function, the server determines which localized string table (as specified in section 3.1.1.13) is needed to fetch the localized string.

If the server can't find the localized string either because it can't find the corresponding *messageId* or the localized string is missing for the *messageId*, it SHOULD fail the method with the error code ERROR_EVT_MESSAGE_ID_NOT_FOUND (0x00003AB4) or ERROR_EVT_MESSAGE_NOT_FOUND (0x00003AB3).

The message strings that the server gets from the language-specific resource file can contain some "%" symbols, which are symbol indicators of substitutions. If the client specifies the values parameter, which is an array of string values, those values will replace the "%" symbols in the message string. For example, the following could be a raw message string:

- "The file system has failed to locate the file %1 with the error %2." And if the values contain 2 elements, one is "sample.evtx", the other is "access denied". Then the string will be expanded into "The file system has failed to locate the file sample.evtx with the error access denied". If the values array contains more elements than the required substitution, the server SHOULD discard the extra ones. If the values array contains less elements than the required substitution, the server SHOULD replace with as many as possible and leave the left one as %number for the final result string. The server SHOULD NOT fail the method regardless of what is specified for the values parameter.

By checking the *flags* parameter, the server knows which information (level, task, opcode, keywords, and so on) the client requests. The server MUST fail the method with the error ERROR_INVALID_PARAMETER(0x00000057) if the *flags* parameter is not one of the values specified in this section.

When EvtFormatMessageId is specified in the *flags* parameter, the server SHOULD use the *messageId* parameter and search through the publisher resource file to find the *messageId* the client specified. Once the server locates the *messageId* in the publisher resource file, it will get the localized string associated with that *messageId* and return the result to the client.<62>

The server MUST return a value indicating success or failure for this operation.

3.1.4.32 EvtRpcMessageRenderDefault (Opnum 10)

The EvtRpcMessageRenderDefault (Opnum 10) method is used by a client to get localized strings for common values of opcodes, tasks, or keywords, as specified in section 3.1.4.31.

```
error_status_t EvtRpcMessageRenderDefault(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_EVENT_ID_SIZE)]
    DWORD sizeEventId,
  [in, size_is(sizeEventId)] BYTE *eventId,
  [in] DWORD messageId,
  [in] EvtRpcVariantList *values,
  [in] DWORD flags,
  [in] DWORD maxSizeString,
  [out] DWORD *actualSizeString,
  [out] DWORD *neededSizeString,
  [out, size_is(,*actualSizeString), range(0, MAX_RPC_RENDERED_STRING_SIZE)]
    BYTE** string,
  [out] RpcInfo *error
);
```

binding: An RPC binding handle as specified in section 2.2.21.

sizeEventId: A 32-bit unsigned integer that contains the size in bytes of the *eventId* field.

eventId: A pointer to an EVENT_DESCRIPTOR structure, as specified in [MS-DTYP] section 2.3.1.

messageId: A 32-bit unsigned integer that specifies the required message. This is an alternative to using the *eventID* parameter that can be used by a client application that has obtained the value through some method outside those documented by this protocol. The server MUST ignore this value unless the *flags* value is set to EvtFormatMessageId, in which case the server MUST use this value to determine the required message and ignore the *eventID* parameter.

values: An array of strings to be used as substitution values for event description strings. Substitution values MUST be ignored by the server except when the *flags* are set to either EvtFormatMessageEvent or EvtFormatMessageId.

flags: This field MUST be set to a value from the following table, which indicates the action that the server is requested to perform.

Value	Meaning
EvtFormatMessageEvent 0x00000001	Locate the message for the event corresponding to *eventId*, and then insert the values specified by the **values** parameter.
EvtFormatMessageLevel 0x00000002	Extract the **level** field from *eventId*, and then return the localized name for that level.
EvtFormatMessageTask 0x00000003	Extract the **task** field from *eventId*, and then return the localized name for that task.
EvtFormatMessageOpcode 0x00000004	Extract the **opcode** field from *eventId*, and then return the localized name for that opcode.
EvtFormatMessageKeyword 0x00000005	Extract the **keyword** field from *eventId*, and then return the localized name for that keyword.
EvtFormatMessageId 0x00000008	Locate the message for the event corresponding to the *messageId* parameter, and then insert the values specified by the *values* parameter.

maxSizeString: A 32-bit unsigned integer that contains the maximum size in bytes allowed for the *string* field.

actualSizeString: A pointer to a 32-bit unsigned integer that contains the actual size of the resulting description string returned in the string. It MUST be set to the size in bytes of the string returned in the *string* parameter, including the NULL ('\0') terminating character. If the description string cannot be retrieved, actualSizeString MUST be set to zero.

neededSizeString: A pointer to a 32-bit unsigned integer that contains the size in bytes of the fully instantiated description string, even if the length of the description string is greater than *maxSizeString*. The returned value MUST be zero when the description string cannot be computed by the server.

string: A buffer in which to return either a null-terminated string or multiple null-terminated strings, terminated by a double NULL in the case of keywords. In the case of failure, the client MUST ignore this value.

error: A pointer to an RpcInfo (section 2.2.1) structure in which to place error information in the case of a failure. The RpcInfo (section 2.2.1) structure fields MUST be set to nonzero values if the error is related to loading the necessary resource. All nonzero values MUST be treated the same. If the method succeeds, the server MUST set all of the values in the structure to 0.

Return Values: The method MUST return the following value on success.

ERROR_SUCCESS (0x00000000)

The method MUST return ERROR_INSUFFICIENT_BUFFER (0x0000007A) if *maxSizeString* is too small to hold the result string. In that case, *neededSizeString* MUST be set to the necessary size.

Otherwise, the method MUST return a different implementation-specific nonzero value as specified in [MS-ERREF].

This method is the same as the EvtRpcMessageRender (section 3.1.4.31) method, except that this method always uses the server's default strings (default strings come from the server's default publisher, so a publisher handle is not required), whereas the EvtRpcMessageRender (section 3.1.4.31) method uses only the default strings in the case of level, task, opcode, and keyword values that fall in certain ranges. Therefore it takes only 6 possible format flags. The server MUST fail the method with ERROR_INVALID_PARAMETER (0x00000057) for any other flags than the 6 values given in the flags table.

3.1.4.33 EvtRpcClose (Opnum 13)

The EvtRpcClose (Opnum 13) method is used by a client to close context handles that are opened by other methods in this protocol.

```
error_status_t EvtRpcClose(
  [in, out, context_handle] void** handle
);
```

handle: This parameter is an RPC context handle, as specified in [C706], Context Handles.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the handle value in its handle table (as specified in section 3.1.1.12) when the handle is created so that it can look up the handle in its table to determine if it is valid.<63> The server MUST fail the operation with the error ERROR_INVALID_PARAMETER (0x00000057) if the handle is not in its

handle table. For more information on handle security and authentication considerations, see sections 2.2.20 and 5.1.

If the above check succeeds, the server MUST remove the handle from its handle table. The server SHOULD NOT fail the operation of removing the handle.

The server MUST return a value indicating success or failure for this operation.

3.1.4.34 EvtRpcCancel (Opnum 14)

The EvtRpcCancel (Opnum 14) method is used by a client to cancel another method. This can be used to terminate long-running methods gracefully. Methods that can be canceled include the subscription and query functions, and other functions that take a CONTEXT_HANDLE_OPERATION_CONTROL argument.

```
error_status_t EvtRpcCancel(
  [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL handle
);
```

handle: A handle obtained by any of the other methods in this interface. This parameter is an RPC context handle, as specified in [C706], Context Handles.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, the server MUST first validate the handle. The server SHOULD save the handle value it created in the handle table (as specified in section 3.1.1.12) and compare it with the handle passed here to perform the validation check.<64>

The server MUST return ERROR_INVALID_PARAMETER (0x00000057) if the handle is invalid. For information on handle security and authentication considerations, see sections 2.2.20 and 5.1.

If the above check succeeds, the server MUST attempt to cancel the outstanding call associated with this handle. As specified in section 3.1.1.10, the context handle SHOULD be a control object on the server. The control object contains the detail operation object pointers such as query object pointer, subscription object pointer, and so forth, plus the Boolean flag. The server SHOULD check if the Boolean flag is true. If the flag is true, the server does nothing and returns success. If this flag is not true, the server SHOULD get the operation object pointer and cancel the operation by stopping operation object processing. That would include stopping processing of the query or subscription tasks and then setting its cancelation Boolean flag to true. For information, see section 3.1.4.

In response to this call, the server MUST NOT remove the associated handle from its handle table.

If the server is too busy to process the outstanding operation, it might not be able to cancel the call. The server SHOULD then return ERROR_CANCELLED (0x00004C7) or other implementation-dependent error codes. If there is no outstanding call or operation, or if the outstanding call or operation has already been canceled, the server SHOULD return ERROR_SUCCESS (0x00000000).

The server MUST return a value indicating success or failure for this operation.

3.1.4.35 EvtRpcRegisterControllableOperation (Opnum 4)

The EvtRpcRegisterControllableOperation (Opnum 4) method obtains a CONTEXT_HANDLE_OPERATION_CONTROL handle that can be used to cancel other operations.

```
error_status_t EvtRpcRegisterControllableOperation(
  [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* handle
```

```
);
```

handle: A context handle for a control object. This parameter MUST be an RPC context handle, as specified in [C706], Context Handles. For information on handle security and authentication considerations, see sections 2.2.20 and 5.1.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, for a successful operation, the server MUST attempt to create a CONTEXT_HANDLE_OPERATION_CONTROL handle. The server SHOULD create a control object. The control object SHOULD initialize its operation pointer to be NULL and Canceled flag to be FALSE. Then the server SHOULD save the control object pointer in its handle table and return the pointer as the context handle to the client. If it cannot create the handle, the server MUST fail the operation with the error ERROR_OUTOFMEMORY (0x0000000E).

The control handle created with this method can be used by the client when it issues the EvtRpcClearLog (section 3.1.4.16), EvtRpcExportLog (section 3.1.4.17), and EvtRpcLocalizeExportLog (section 3.1.4.18) methods so that the client can cancel those operations if the server takes too long to return. Those methods take the control object as their first parameter, and periodically check the object's **Canceled** field during the processes. The server SHOULD NOT change the control object's operational pointer data field in its implementation of the **EvtRpcClearLog**, **EvtRpcExportLog**, and **EvtRpcLocalizeExportLog** methods. If the client sets the **Canceled** field to be TRUE by using the method (section EvtRpcCancel (section 3.1.4.34), the server SHOULD respond to the change by halting the process of clearing, exporting, or localizing the log file.

Note The control object created by this method SHOULD only be used in the **EvtRpcClearLog**, **EvtRpcExportLog**, and **EvtRpcLocalizeExportLog** methods; it is not updated or consumed by any other methods. As specified in sections 3.1.4.8 and 3.1.4.12, operation control objects are created by the server when processing calls to the **EvtRpcRegisterRemoteSubscription** and **EvtRpcRegisterLogQuery** methods; those are the only situations in which the server sets the operational pointer data field for the control object. For more information, see sections 3.1.4.8 and 3.1.4.12.

The server MUST return a value indicating success or failure for this operation.

3.1.4.36 EvtRpcGetClassicLogDisplayName (Opnum 28)

The EvtRpcGetClassicLogDisplayName (Opnum 28) method obtains a descriptive name for a channel.

```
error_status_t EvtRpcGetClassicLogDisplayName(
  /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR logName,
  [in] LCID locale,
  [in] DWORD flags,
  [out] LPWSTR* displayName
);
```

binding: An RPC binding handle as specified in section 2.2.21.

logName: The channel name for which the descriptive name is needed.

locale: The locale, as specified in [MS-GPSI] Appendix A, to be used for localizing the log.

flags: A 32-bit unsigned integer that MUST be set to one of the following values:

Value	Meaning
0x0	If a locale is specified, that locale will be used and no fallback locale will be attempted if the locale is not present.
0x100	If set, instructs the server to pick the best locale, if the locale specified by the *locale* parameter is not present. Please see the following processing rules for more information on how the server picks the best locale.

displayName: Returned display name.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value as specified in [MS-ERREF].

In response to this request from the client, for a successful operation, the server MUST attempt to retrieve a display name for a channel. In [MS-EVEN] section 3.1.1.2, there are two configuration entries for a classic event log: DisplayNameFile and DisplayNameID. The server tries to use the log name passed here to find the log entry in the registry and then locate the DisplayNameID and DisplayNameFile. The DisplayNameID is the message ID for the display name. The file which is pointed at by DisplayNameFile contains the localized string for the display name. The server uses the messageId and locale as the combination key to look for the string inside the file and then retrieve the classic event log display name. The server verifies that the channel name, as specified by the *logName* parameter, is a known classic event log. If the *logName* parameter does not specify a registered event log (the log name can't be located in the registry described in [MS-EVEN]), the server MUST fail the method with the error ERROR_NOT_FOUND (0x00000490). If the server can't find the DisplayNameID or the DisplayNameFile, the server SHOULD fail the method with the error ERROR_INVALID_DATA (0x0000000D).

If the locale is not specified (that is, the caller passes zero for the locale parameter) and the *flags* parameter is 0x0, the server MUST substitute the locale of its own execution thread as the value of the locale parameter.

If the display name is not present in the specified *locale* as specified above and the *flags* parameter is set to 0x100, the server makes a best effort attempt to find a localized display name string by following these steps:

1. The server SHOULD attempt to find a fallback locale with the same base language which does contain the localized display name.<65> For example, if the caller requested the U.S. English "en-US" locale but the display name was only available in another "en" prefixed locale such as the British English "en-GB" locale, the server would return the British English version of the display name.

2. Using the fallback locale, the server SHOULD attempt to find a localized display name string by searching the appropriate DisplayNameFile in the same manner as specified above.

3. If the server cannot determine an appropriate fallback locale or cannot find a localized display name string using the fallback locale, the server MUST return ERROR_NOT_FOUND (0x00000490).

The server SHOULD validate the flags to ensure that no flags are present other than 0x0 and 0x100.<66>

Otherwise, the server MUST fail the method with the error ERROR_INVALID_PARAMETER (0x00000057).

This API only succeeds if called for a log name that is exposed by the obsolete Eventlog Remote protocol. If called on logs that are not exposed by the obsolete Eventlog Remote protocol, the method will fail with the error ERROR_NOT_FOUND (0x00000490). For more information, see section 3.1.1.6.

The server MUST return a value indicating success or failure for this operation.

3.1.5 Timer Events

None.

3.1.6 Other Local Events

None.

3.2 Client Details

The client side of this protocol is simply a pass-through.

3.2.1 Abstract Data Model

The client does not maintain state as part of this protocol.

3.2.2 Timers

None.

3.2.3 Initialization

None.

3.2.4 Message Processing Events and Sequencing Rules

Calls made by the higher-layer protocol or application MUST be passed directly to the transport. All return values from method invocations MUST be returned uninterpreted to the higher-layer protocol or application.

3.2.5 Timer Events

None.

3.2.6 Other Local Events

None.

3.2.7 Changing Publisher Configuration Data

The configuration property for a publisher contains only the publisher resource file location. For the client to change a publisher configuration property, the client and the server MUST be on the same machine. The client MAY access the server's memory and locate the resource file in the server's memory for a specified publisher and change its value. The memory used by the server to save the resource file location of a publisher can be a shared memory so that the client can also read that memory and access the data. After the client changes the resource file location, it can call the **EvtRpcAssertConfig** method (as specified in section 3.1.4.29) to apply the change to the server's disk.<67>

4 Protocol Examples

4.1 Query Example

In this example, the client wants to obtain events from a channel log file and render the resultant events as XML text.

This involves the following steps:

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the EvtRpcRegisterLogQuery (section 3.1.4.12) method to establish a query over the log file and to obtain a query result and operation control handles.

```
error_status_t
EvtRpcRegisterLogQuery(
   [in] RPC_BINDING_HANDLE binding = {binding handle from step 1.},
   [in, unique, range(0, MAX_RPC_CHANNEL_PATH_LENGTH)]
      LPCWSTR path = "Application",
   [in, range(1, MAX_RPC_QUERY_LENGTH)] LPCWSTR query = "*",
   [in] DWORD flags = 0x00000101
   [out, context_handle] PCONTEXT_HANDLE_LOG_QUERY* handle,
   [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL*
     opControl,
   [out] DWORD* queryChannelInfoSize,
   [out, size_is(,*queryChannelInfoSize),
     range(0,MAX_RPC_QUERY_CHANNEL_SIZE)]
     EvtRpcQueryChannelInfo** queryChannelInfo,
   [out] RpcInfo *error
);
```

3. When the server processes this call, it opens the Application channel, saves the * as the query expression, and creates two handles.

   Note This example uses * as the query; see sections 4.5 and 4.6 for examples of structured queries.

   The call returns successfully, and the client is given two handles: a query result handle and an operation control handle. The former is used to enumerate the results, and the latter is used to cancel a currently executing control handle.

   As noted in section 3.1.4.12, the query result handle is a query object from the server side. It ~~SHOULD contain~~contains the channel path, the query filter XPath expression, and the result cursor value. For this example, the channel path is "Application", the XPath filter expression is "*" and the result cursor value is 0. The operation control handle is a control object that contains only one Boolean field which indicates whether the query operation has been canceled or not. In this example, the value is currently false which means the operation has not yet been canceled by the client.

4. The client enumerates events in the resultant list by calling the EvtRpcQueryNext (section 3.1.4.13) method by using the query handle obtained in the previous step.

```
error_status_t
EvtRpcQueryNext(
   [in, context_handle] PCONTEXT_HANDLE_LOG_QUERY logQuery
```

```
      = { handle obtained by the call to EvtRpcRegisterLogQuery },
   [in] DWORD numRequestedRecords = 5,
   [in] DWORD timeOutEnd = 3000,
   [in] DWORD flags = 0,
   [out] DWORD* numActualRecords,
   [out, size_is(,*numActualRecords),
     range(0, MAX_RPC_RECORD_COUNT)] DWORD** eventDataIndices,
   [out, size_is(,*numActualRecords),
     range(0, MAX_RPC_RECORD_COUNT)] DWORD** eventDataSizes,
   [out] DWORD* resultBufferSize,
   [out, size_is(,*resultBufferSize),
     range(0, MAX_RPC_BATCH_SIZE)] BYTE** resultBuffer
);
```

5. The server implements this call by returning the requested number of events (or as many events as it has) in BinXml form.

   See section 4.8 for an example in BinXml form.

   The client enumerates through the events, using multiple calls to the EvtRpcQueryNext (section 3.1.4.13) method, until it no longer responds to events, or it reaches the end of the log file.

   If the client's query expression selects sparse events, and the log file contains a huge number of events, the EvtRpcQueryNext can take a long time to complete. In this case, the client has the option to cancel the EvtRpcQueryNext call by passing the query result handle to the EvtRpcCancel (section 3.1.4.34) method.

6. For each event, it is translated from BinXml encoding to the XML representation.

   This is done according to the BinXml ABNF, as specified in section 3.1.4.7.

   The server is not involved in this step.

   If the event XML representation conforms to event.xsd (for more information, see section 2.2.13), standard attributes can be retrieved either directly from the BinXml representation or after translating to text XML.

   The client optionally translates the event into text XML. See section 4.8 for an example of how to translate from BinXml form into XML event format.

7. When the client is done enumerating, it closes both the query and operation control handles using EvtRpcClose. In this call, the server frees all resources related to the query result.

```
error_status_t EvtRpcClose(
   [in, out, context_handle] void** handle = {query handle}
);

error_status_t EvtRpcClose(
   [in, out, context_handle] void** handle
     = {operation control handle}
);
```

4.2 Get Log Information Example

In this example, the client wants to get information about a channel or log file.

This involves the following steps:

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the **EvtRpcOpenLogHandle** (as specified in section 3.1.4.19) method to open the log handle from which it wants to get information.

```
error_status_t
EvtRpcOpenLogHandle(
  [in, range(1, MAX_RPC_CHANNEL_PATH_LENGTH), string]
    LPCWSTR channel = "Application",
  [in] DWORD flags = 1,
  [out, context_handle] PCONTEXT_HANDLE_LOG_HANDLE* handle,
  [out] RpcInfo* error
);
```

   After this function returns successfully, the client receives the log context handle. As mentioned in section 3.1.1.11, the context handle is a log information object. For this example, its content is:

   - LogType = {A value which means it is a channel}
   - Channel = {Pointer to the "application" entry in the channel table}

3. The client then calls the **EvtRpcGetLogFileInfo** (as specified in section 3.1.4.12) method to get the necessary information. For the following example, assume the client wants to know the number of events in the channel.

```
error_status_t
EvtRpcGetLogFileInfo(
  [in, context_handle] PCONTEXT_HANDLE_LOG_HANDLE logHandle = {The handle received above},
  [in] DWORD propertyId = 0x00000005(EvtLogNumberOfLogRecords),
  [in, range(0, MAX_RPC_PROPERTY_BUFFER_SIZE)]
    DWORD propertyValueBufferSize = sizeof(BinXmlVariant),
  [out, size_is(propertyValueBufferSize)]
    BYTE* propertyValueBuffer = {The pointer which points to the result buffer},
  [out] DWORD* propertyValueBufferLength
);
```

   After the method returns successfully, the propertyValueBuffer contains the required value and is packed in the following data format:

```
0x0000000000000020 -------- There are 32 records totally.
0x0001                      -------- There is only one result in the returned data
0x000A                      -------- The data type is BinXmlVarUInt64.
```

4.3 Bookmark Example

The following is an example of Bookmark use.

```
<?xml version="1.0" encoding="UTF-8"?>
<BookmarkList>
<Bookmark Channel=" Microsoft-Windows-PrintSpooler/Operational"
```

```
     RecordId="9"/>
<Bookmark Channel="c:/dir1/dir2/file.evtx" RecordId="1"/>
<Bookmark Channel="System" RecordId="26" IsCurrent="true"/>
</BookmarkList>
```

4.4 Simple BinXml Example

The following is an example of a simple BinXml fragment (without use of templates):

```
<Event>
<Element1>abc</Element1>
<Element2> def &amp;&#60; ghi </Element2>
<Element3 AttrA='abc' AttrB='def&amp;&#60;ghi'/>
</Event>
00 : 0f 01 01 00 01 f2 00 00-00 ba 0c 05 00 45 00 76   <Event>
10 : 00 65 00 6e 00 74 00 00-00 02 01 22 00 00 00 b5
20 : 79 08 00 45 00 6c 00 65-00 6d 00 65 00 6e 00 74
30 : 00 31 00 00 00 02 05 01-03 00 61 00 62 00 63 00
40 : 04 01 44 00 00 00 b6 79-08 00 45 00 6c 00 65 00
                                                 </Element1> <Element2>
50 : 6d 00 65 00 6e 00 74 00-32 00 00 00 02 45 01 05
60 : 00 20 00 64 00 65 00 66-00 20 00 49 24 fb 03 00
70 : 61 00 6d 00 70 00 00 00-48 3c 00 05 01 05 00 20
80 : 00 67 00 68 00 69 00 20-00 04 41 6b 00 00 00 b7
                                                 </Element2> <Element3>
90 : 79 08 00 45 00 6c 00 65-00 6d 00 65 00 6e 00 74
A0 : 00 33 00 00 00 50 00 00-00 46 90 d8 05 00 41 00
B0 : 74 00 74 00 72 00 41 00-00 00 05 01 03 00 61 00
C0 : 62 00 63 00 06 91 d8 05-00 41 00 74 00 74 00 72
D0 : 00 42 00 00 00 45 01 03-00 64 00 65 00 66 00 49
E0 : 24 fb 03 00 61 00 6d 00-70 00 00 00 48 3c 00 05
F0 : 01 03 00 67 00 68 00 69-00 03 04 00
                                                   <Element3/> </Event>
```

Token offset	Token type	Comments on encoding
00	0F - FragmentHeaderToken	Version 1.1, flags=0
04	01 - OpenStartElementToken	<Event> start tag. There is no Dependency ID because this is not a template definition. The length of the data for the entire element is 0xF2, and this data starts at offset 0x09. The data consists of three parts: name hash (2 bytes), string length (2 bytes), and name itself (the rest of the data). At offset 0x09, the name begins for the start tag. The length of the name is five Unicode characters (does not include a null-terminator). Note, the more bit is not set on the OpenStartElementToken, so attributes do not follow.
19	02 - CloseStartElementToken	Close <Event> start tag.
1A	01 - OpenStartElementToken	<Element1> start tag, no attributes to follow.
35	02 - CloseStartElementToken	Close <Element1> start tag.
36	05 - ValueTextToken	The character data abc. It has a length of three Unicode characters (character data strings are not null- terminated).
40	04 - EndElementToken	End </Element1>.

Token offset	Token type	Comments on encoding
41	01 - OpenStartElementToken	<Element2>.
5C	02 - CloseStartElementToken	Close <Element2> start tag.
5D	45 - ValueTextToken (MoreBit)	The character data "def ". Spaces surround def so its length is five Unicode characters. The more bit is set, so there is more character data that follows.
6B	49 - EntityRefToken (MoreBit)	An entity reference with Name amp. More character data follows.
78	48 - CharRefToken (MoreBit)	The character reference for "&". More character data follows.
7B	05 - ValueTextToken	The character data "ghi " (again with spaces). No more character data appears before the next markup token.
89	04 - EndElementToken	End </Element2>.
8A	41 - OpenStartElementToken (MoreBit - AttrList)	<Element3>. The more bit is set, so an attribute list follows. The first attribute starts at offset 0xA9.
A9	46 - AttributeToken (More Bit)	This is AttrA, and more attributes follow.
BA	05 - ValueTextToken	This is abc.
C4	06 - AttributeToken (No More Bit)	This is AttrB, and no more attributes follow.
D5	45 - ValueTextToken (More Bit)	The character data def, with more character data to follow.
DF	49 - EntityRefToken (MoreBit)	The entity ref with Name amp with more character data to follow.
EC	48 - CharRefToken (MoreBit)	The character reference for ampersand (&) with more character data to follow.
EF	05 - ValueTextToken	The character data ghi, with no more character data before the next markup token.
F9	03 - CloseEmptyElementToken	Close empty <Element3/>.
FA	04 - EndElementToken	End </Event>.
FB	00 - EOFToken	End of fragment / document.

4.5 Structured Query Example

The following is an example of a structured XML query. It contains two subqueries with the IDs of 1 and 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<QueryList>
<Query Id="1" Path="System">
<Select Path="Microsoft-Windows-PrintSpooler/Operational">
*[System/Level=1]
</Select>
<Select>*[System/Level=2]</Select>
 <Suppress>*[UserData/*/PrinterName="MyPrinter"]</Suppress>
</Query>
<Query Id="2" Path="file://c:/dir1/dir2/file.evtx">
<Select>*[System/Level=2]</Select>
```

```
</Query>
</QueryList>
```

4.6 Push Subscription Example

In this example, the client asks to get all future events from the "Application" and "Microsoft-Windows-Backup/Operational" channel through push mode. This involves the following steps:

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the **EvtRpcRegisterRemoteSubscription** method (section 3.1.4.8) to establish a subscription connection and to obtain a subscription context and operation control handles.

```
error_status_t EvtRpcRegisterRemoteSubscription(
  [in] RPC_BINDING_HANDLE binding = {binding handle from step 1.},
  [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH), string] LPCWSTR channelPath = NULL,
  [in, range(1, MAX_RPC_QUERY_LENGTH), string]
    LPCWSTR query = {pointer to a structure query which describes the two channels.},
  [in, unique, range(0, MAX_RPC_BOOKMARK_NG,] LPCWSTR bookmarkXml = NULL,
  [in] DWORD flags = 0x00000001,
  [out, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION* handle,
  [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* control,
  [out] DWORD* queryChannelInfoSize,
  [out, size_is(, *queryChannelInfoSize), range(0, MAX_RPC_QUERY_CHANNEL_SIZE)]
    EvtRpcQueryChannelInfo** queryChannelInfo,
  [out] RpcInfo*error
);
```

The structure query content for parameter query in this example would be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<QueryList>
<Query Id="1" Path="Application">
<Select Path="Application">*</Select>
</Query>
<Query Id="2" Path="Microsoft-Windows-Backup/Operational">
<Select Path="Microsoft-Windows-Backup/Operational">*</Select>
</Query>
</QueryList>
```

3. The server handles the registration request from the client and the server creates a subscription object and a control object and casts them to remote subscription context handle and operation control context handle. As described in section 3.1.4.8, the subscription object contains the list of client subscribed channels. In this example, the subscription object contains "Application" and "Microsoft-Windows-Backup/Operational". Because these two channels are registered with the server already, the server finds and opens both of them to read how many events there are for each channel. Suppose the Application channel contains 200 existing events and the other one contains 100 existing events. The subscription object then has two longlong type of numeric cursor values for each channel, one is 201 and the other is 101. Because the client asks to get the future events, the subscription object sets the two cursors to the end of the channel to indicate that only future events will be delivered to the client. Then the subscription object sees that the flag does not specify the pull mode, and it sets the push mode flag to be true. For information on the operation control handle value, see section 4.1.

After the server has completed these steps, it passes the subscription handle and control handle to the client.

4. Once the client gets the subscription handle, it calls the **EvtRpcRemoteSubscriptionNextAsync** method (section 3.1.4.9) to fetch its subscribed events in the asynchronized way.

```
error_status_t EvtRpcRemoteSubscriptionNextAsync(
  [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle = {handle from step 2},
  [in] DWORD numRequestedRecords = 5,
  [in] DWORD flags = 0,
  [out] DWORD* numActualRecords,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataIndices,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
    DWORD** eventDataSizes,
  [out] DWORD* resultBufferSize,
  [out, size_is(, *resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
    BYTE** resultBuffer
);
```

5. On the server, it first checks if there are any new events in either of the channels. If there are no new events, the server returns the call immediately but does not complete the call. In this way the client does not need to wait when there are no new events coming into either of the channels. But the server keeps monitoring the channel. The server implements that by either checking the latest event in the channel periodically or registering a callback function from a system component that accepts the events from a provider. Note that for a Windows server, the server registers its callback to the system component so that it can receive notification when a provider generates events. Later, if a provider generates a new event into the Application channel, with that information the server fills the event data in the resultBuffer and notifies the client that the events are coming now.

6. The client then gets notified by RPC that the result is ready in its supplies buffer, as described in step 4. The client can then access the event data in its buffer. For more information on how the client interprets the data in the result buffer, see section 4.1.

This example shows the benefits of the push subscription. The client is not blocked by the server if there are no events that match the criteria of the client's subscriber requirement. Instead, the client can perform its own tasks while the server is waiting for the new events, and get notified when the server has new events ready.

4.7 Pull Subscription Example

In this example, the client asks to get all the events from the "Application" channel after its supplied bookmark comes through pull mode. This involves the following steps:

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the **EvtRpcRegisterRemoteSubscription** method (section 3.1.4.8) to establish a subscription connection and to obtain a subscription context and operation control handles.

```
error_status_t EvtRpcRegisterRemoteSubscription(
  [in] RPC_BINDING_HANDLE binding = {binding handle from step 1.},
  [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH), string] LPCWSTR channelPath =
L"Application",
  [in, range(1, MAX_RPC_QUERY_LENGTH), string]
    LPCWSTR query = NULL,
  [in, unique, range(0, MAX_RPC_BOOKMARK_LENGTH), string]
```

```
    LPCWSTR bookmarkXml = {pointer to the bookmark XML text},
  [in] DWORD flags = 0x10000003,{Pull mode and after bookmark}
  [out, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION* handle,
  [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* control,
  [out] DWORD* queryChannelInfoSize,
  [out, size_is(, *queryChannelInfoSize), range(0, MAX_RPC_QUERY_CHANNEL_SIZE)]
    EvtRpcQueryChannelInfo** queryChannelInfo,
  [out] RpcInfo* error
);
```

The bookmark XML for the example could be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<BookmarkList>
<Bookmark Channel="Application" RecordId="10"/>
</BookmarkList>
```

3. On the server, the **EvtRpcRegisterRemoteSubscription** method creates the subscription object and control object. The subscription object contains one channel called "Application". Because the client requires the events after the bookmark, the server parses the bookmark XML and finds that the client requests the events whose record ID is larger than 10. Thus it sets its cursor value for the Application channel to 11. Then the server notes that the flag contains a pull mode so it sets its push mode flag to be false. For information on the control object content, see section 4.1.

   After **EvtRpcRegisterRemoteSubscription** creates the two objects, the server casts them to the subscription context handle and the operation control handle.

4. After the client gets the subscription context handle, it calls the **EvtRpcRemoteSubscriptionNext** method (section 3.1.4.10) to fetch the events in a synchronized way.

```
error_status_t EvtRpcRemoteSubscriptionNext(
  [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle = {the handle from step 2},
  [in] DWORD numRequestedRecords = 5,
  [in] DWORD timeOut = 1000,
  [in] DWORD flags = 0,
  [out] DWORD* numActualRecords,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)] DWORD**
eventDataIndices,
  [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)] DWORD** eventDataSizes,
  [out] DWORD* resultBufferSize,
  [out, size_is(,*resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
    BYTE** resultBuffer
);
```

   Unlike the **EvtRpcRemoteSubscriptionNextAsync** method, this method will block the client if there are no events that match the criteria of the client's subscriber requirement. In this example, the timeOut value is 1000 (which means one second), so the client waits for one second if there are no events after the 10th record in the Application channel. Next, suppose there are events after the 10th record, the server then fills the result buffer with the available events (but not exceeding 5 because the client only demands 5 events), and then completes the call.

5. For information on how the events in the result buffer are interpreted by the client, see section 4.1.

4.8 BinXml Example Using Templates

This example demonstrates the use of BinXml templates. There is one outer template <Event> and one inner template <MyEvent>. The outer template has substitutions (shown in bold) under the <System> element. However, it also has a BinXml substitution within the <UserData> element. In other words, the BinXml that describes <MyEvent> is contained as a value for the outer <Event> template instance. The BinXml for <MyEvent> happens to also be another template instance (although it could have been a normal fragment). The MyEvent template substitutions are also shown in bold.

Also, the outer template substitutions are all optional, and some values of that template are NULL; therefore, some of the BinXml elements or attributes are not present in the following XML text.

```
<Event xmlns=
  "'http: //schemas.microsoft.com/win/2004/08/events/event'">
<System>
  <Provider Name="'Microsoft-Windows-Wevttest'"
            Guid="'{03f41308-fa7b-4fb3-98b8-c2ed0a40d1ef}'"/>
  <EventID>100</EventID>
  <Version>0</Version>
  <Level>1</Level>
  <Task>100</Task>
  <Opcode>1</Opcode>
  <Keywords>0x4000000000e00000</Keywords>
  <TimeCreated SystemTime="'2006-0614T21:40:16.312Z'"/>
  <EventRecordID>5</EventRecordID>
  <Correlation/>
  <Execution ProcessID="'2088'" ThreadID="'2464'"/>
  <Channel>Microsoft-Windows-Wevttest/Operational/Wevttest</Channel>
  <Computer>michaelm4-lh.ntdev.corp.microsoft.com</Computer>
  <Security
UserID="'S-1-5-21-397955417-626881126-188441444-2967838'"/>
</System>
<UserData>
  <MyEvent xmlns:autons2=
"'http: //schemas.microsoft.com/win/2004/08/events'"
  xmlns='myNs'><Property>1</Property>
  <Property2>2</Property2>
  </MyEvent>
</UserData>
</Event>
```

Start of <Event> TemplateInstance ...

```
00 : 0f 01 01 00 0c 00 4a 46-4c cc 16 dc 46 8e 80 a2
10 : dc 45 ea 94 9c bd ef 04-00 00 0f 01 01 00 41 ff  <Event>
20 : ff e3 04 00 00 ba 0c 05-00 45 00 76 00 65 00 6e
30 : 00 74 00 00 00 7f 00 00-00 06 bc 0f 05 00 78 00
40 : 6d 00 6c 00 6e 00 73 00-00 00 05 01 35 00 68 00
50 : 74 00 74 00 70 00 3a 00-2f 00 2f 00 73 00 63 00
60 : 68 00 65 00 6d 00 61 00-73 00 2e 00 6d 00 69 00
70 : 63 00 72 00 6f 00 73 00-6f 00 66 00 74 00 2e 00
80 : 63 00 6f 00 6d 00 2f 00-77 00 69 00 6e 00 2f 00
90 : 32 00 30 00 30 00 34 00-2f 00 30 00 38 00 2f 00
A0 : 65 00 76 00 65 00 6e 00-74 00 73 00 2f 00 65 00
B0 : 76 00 65 00 6e 00 74 00-02 01 ff ff 24 04 00 00  <System>
C0 : 6f 54 06 00 53 00 79 00-73 00 74 00 65 00 6d 00
D0 : 00 00 02 41 ff ff c1 00-00 00 f1 7b 08 00 50 00  <Provider>
E0 : 72 00 6f 00 76 00 69 00-64 00 65 00 72 00 00 00
F0 : a6 00 00 00 46 4b 95 04-00 4e 00 61 00 6d 00 65
100: 00 00 00 05 01 1a 00 4d-00 69 00 63 00 72 00 6f
110: 00 73 00 6f 00 66 00 74-00 2d 00 57 00 69 00 6e
120: 00 64 00 6f 00 77 00 73-00 2d 00 57 00 65 00 76
130: 00 74 00 74 00 65 00 73-00 74 00 06 29 15 04 00
```

```
140: 47 00 75 00 69 00 64 00-00 00 05 01 26 00 7b 00
150: 30 00 33 00 66 00 34 00-31 00 33 00 30 00 38 00
160: 2d 00 66 00 61 00 37 00-62 00 2d 00 34 00 66 00
170: 62 00 33 00 2d 00 39 00-38 00 62 00 38 00 2d 00
180: 63 00 32 00 65 00 64 00-30 00 61 00 34 00 30 00
190: 64 00 31 00 65 00 66 00-7d 00 03 41 03 00 3d 00
<Provider/> <EventID>
1A0: 00 00 f5 61 07 00 45 00-76 00 65 00 6e 00 74 00
1B0: 49 00 44 00 00 00 1f 00-00 00 06 29 da 0a 00 51
1C0: 00 75 00 61 00 6c 00 69-00 66 00 69 00 65 00 72
1D0: 00 73 00 00 00 0e 04 00-06 02 0e 03 00 06 04 01 </EventID>
1E0: 0b 00 1a 00 00 00 18 09-07 00 56 00 65 00 72 00
1F0: 73 00 69 00 6f 00 6e 00-00 00 02 0e 0b 00 04 04
200: 01 00 00 16 00 00 00 64-ce 05 00 4c 00 65 00 76
210: 00 65 00 6c 00 00 00 02-0e 00 00 04 04 01 02 00
220: 14 00 00 00 45 7b 04 00-54 00 61 00 73 00 6b 00
230: 00 00 02 0e 02 00 06 04-01 01 00 18 00 00 00 ae
240: 1e 06 00 4f 00 70 00 63-00 6f 00 64 00 65 00 00
250: 00 02 0e 01 00 04 04 01-05 00 1c 00 00 00 6a cf
260: 08 00 4b 00 65 00 79 00-77 00 6f 00 72 00 64 00
270: 73 00 00 00 02 0e 05 00-15 04 41 ff ff 40 00 00
280: 00 3b 8e 0b 00 54 00 69-00 6d 00 65 00 43 00 72
290: 00 65 00 61 00 74 00 65-00 64 00 00 00 1f 00 00
2A0: 00 06 3c 7b 0a 00 53 00-79 00 73 00 74 00 65 00
2B0: 6d 00 54 00 69 00 6d 00-65 00 00 00 00 0e 06 00 11
2C0: 03 01 0a 00 26 00 00 00-46 03 0d 00 45 00 76 00
2D0: 65 00 6e 00 74 00 52 00-65 00 63 00 6f 00 72 00
2E0: 64 00 49 00 44 00 00 00-02 0e 0a 00 0a 04 41 ff
2F0: ff 6d 00 00 00 a2 f2 0b-00 43 00 6f 00 72 00 72
300: 00 65 00 6c 00 61 00 74-00 69 00 6f 00 6e 00 00
310: 00 4c 00 00 00 46 0a f1-0a 00 41 00 63 00 74 00
320: 69 00 76 00 69 00 74 00-79 00 49 00 44 00 00 00
330: 0e 07 00 0f 06 35 c5 11-00 52 00 65 00 6c 00 61
340: 00 74 00 65 00 64 00 41-00 63 00 74 00 69 00 76
350: 00 69 00 74 00 79 00 49-00 44 00 00 00 0e 12 00
360: 0f 03 41 ff ff 55 00 00-00 b8 b5 09 00 45 00 78
370: 00 65 00 63 00 75 00 74-00 69 00 6f 00 6e 00 00
380: 00 38 00 00 00 46 0a d7-09 00 50 00 72 00 6f 00
390: 63 00 65 00 73 00 73 00-49 00 44 00 00 00 0e 08
3A0: 00 08 06 85 39 08 00 54-00 68 00 72 00 65 00 61
3B0: 00 64 00 49 00 44 00 00-00 0e 09 00 08 03 01 ff
3C0: ff 78 00 00 00 83 61 07-00 43 00 68 00 61 00 6e
3D0: 00 6e 00 65 00 6c 00 00-00 02 05 01 2f 00 4d 00
3E0: 69 00 63 00 72 00 6f 00-73 00 6f 00 66 00 74 00
3F0: 2d 00 57 00 69 00 6e 00-64 00 6f 00 77 00 73 00
400: 2d 00 57 00 65 00 76 00-74 00 74 00 65 00 73 00
410: 74 00 2f 00 4f 00 70 00-65 00 72 00 61 00 74 00
420: 69 00 6f 00 6e 00 61 00-6c 00 2f 00 57 00 65 00
430: 76 00 74 00 74 00 65 00-73 00 74 00 04 01 ff ff
440: 66 00 00 00 3b 6e 08 00-43 00 6f 00 6d 00 70 00
450: 75 00 74 00 65 00 72 00-00 00 02 05 01 25 00 6d
460: 00 69 00 63 00 68 00 61-00 65 00 6c 00 6d 00 34
470: 00 2d 00 6c 00 68 00 2e-00 6e 00 74 00 64 00 65
480: 00 76 00 2e 00 63 00 6f-00 72 00 70 00 2e 00 6d
490: 00 69 00 63 00 72 00 6f-00 73 00 6f 00 66 00 74
4A0: 00 2e 00 63 00 6f 00 6d-00 04 41 ff ff 32 00 00
4B0: 00 a0 2e 08 00 53 00 65-00 63 00 75 00 72 00 69
4C0: 00 74 00 79 00 00 00 17-00 00 00 06 66 4c 06 00
4D0: 55 00 73 00 65 00 72 00-49 00 44 00 00 00 0e 0c      </System>
4E0: 00 13 03 04 01 13 00 1c-00 00 00 35 44 08 00 55      <UserData>
4F0: 00 73 00 65 00 72 00 44-00 61 00 74 00 61 00 00
500: 00 02 0e 13 00 21 04 04-00 </UserData> </Event> EOF
```

Start of <Event> TemplateInstanceData ValueSpec ...

```
                                  14 00 00 00 01 00 04
510: 00 01 00 04 00 02 00 06-00 02 00 06 00 00 00 00
```

```
520: 00 08 00 15 00 08 00 11-00 00 00 00 00 04 00 08
530: 00 04 00 08 00 08 00 0a-00 01 00 04 00 1c 00 13
540: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
550: 00 00 00 00 00 00 00 00-00 83 01 21 00
```

Start of <Event> TemplateInstanceData Values ...

```
                                                  01 01 64
560: 00 64 00 00 00 e0 00 00-00 00 40 9c f4 d6 36 fb
570: 8f c6 01 28 08 00 00 a0-09 00 00 06 00 00 00 00
580: 00 00 00 00 01 05 00 00-00 00 00 05 15 00 00 00
590: 59 51 b8 17 66 72 5d 25-64 63 3b 0b 1e 49 2d 00
```

Start of <MyEvent> inner TemplateInstance ...

```
5A0: 0f 01 01 00 0c 00 a7 65-05 7a 02 84 f0 a1 67 ab
5B0: 96 df 09 0d 39 a7 54 01-00 00 41 ff ff 04 01 00  <MyEvent>
5C0: 00 4e c0 07 00 4d 00 79-00 45 00 76 00 65 00 6e
5D0: 00 74 00 00 00 a2 00 00-00 46 4d 77 0e 00 78 00
5E0: 6d 00 6c 00 6e 00 73 00-3a 00 61 00 75 00 74 00
5F0: 6f 00 2d 00 6e 00 73 00-32 00 00 00 05 01 2f 00
600: 68 00 74 00 74 00 70 00-3a 00 2f 00 2f 00 73 00
610: 63 00 68 00 65 00 6d 00-61 00 73 00 2e 00 6d 00
620: 69 00 63 00 72 00 6f 00-73 00 6f 00 66 00 74 00
630: 2e 00 63 00 6f 00 6d 00-2f 00 77 00 69 00 6e 00
640: 2f 00 32 00 30 00 30 00-34 00 2f 00 30 00 38 00
650: 2f 00 65 00 76 00 65 00-6e 00 74 00 73 00 06 bc
660: 0f 05 00 78 00 6d 00 6c-00 6e 00 73 00 00 00 05
670: 01 04 00 6d 00 79 00 4e-00 73 00 02 01 ff ff 1c  <Property>
680: 00 00 00 b5 db 08 00 50-00 72 00 6f 00 70 00 65
690: 00 72 00 74 00 79 00 00-00 02 0d 00 00 08 04 01
   </Property> <Property2>
6A0: ff ff 1e 00 00 00 bd 11-09 00 50 00 72 00 6f 00
6B0: 70 00 65 00 72 00 74 00-79 00 32 00 00 00 02 0d
6C0: 01 00 08 04 04 00           </Property2> </MyEvent> EOF
```

Waste bytes that could occur after template definition EOF but included in TemplateDefLength ...

```
                       00 00-00 00 08 08 00 00 00 00
6D0: 00 00 00 00 00 00 08 07-00 00 00 00 00 00 08 08
6E0: 00 00 00 00 00 00 00 00-00 00 18 07 00 00 10 00
6F0: 00 00 50 00 72 00 6f 00-70 00 31 00 00 00 10 00
700: 00 00 50 00 72 00 6f 00-70 00 32 00 00 00
```

Start of <MyEvent> inner TemplateInstanceData ...

```
                                                    02 00
710: 00 00 04 00 08 00 04 00-08 00 01 00 00 00 02 00
720: 00 00 00 00
```

Token offset	Token type	Comments on encoding
0x00	0x0F -	Version1.1, Flags = 0. This is at the "document" level, and it is likely

Token offset	Token type	Comments on encoding
	FragmentHeaderToken	that an EOFToken will occur at the end.
0x04	0x0C - TemplateInstanceToken	Outer template instance <Event>. The TempleDefByteLength is 0x4EF and the template definition starts at 0x1A. This means that the end of the template definition will be at 0x1A + 0x4EF = 0x509 (which is the start of the TemplateInstanceData). The ValueSpec of the TemplateInstanceData specifies that there are 0x14 values with a total length of 0x1C6 bytes. This length is calculated by adding up all the lengths of the values specified in the value spec entries. The actual raw values of the template instance data start just after the value spec entries (at offset 0x55D). Offset 0x55D + 0x1C6 bytes leave us at the EOF token for the outer fragment containing the TemplateInstance.
0x1A	0x0F - FragmentHeaderToken	Version for template definition BinXml. This could be different from the template instance version.
0x1E	0x41 - OpenStartElementToken (more Bit)	<Event>. Note that because this is a template definition, the dependency ID is included, but 0xFFFF indicates no dependency. This value actually consists of two parts. The 0x01 indicates that it is an OpenStartElementToken, and the 0x40 is the "more" bit, which indicates that there are additional attributes.
0xB9	0x1 - OpenStartElementToken	<System>. This has a dependency of 0xFFFF.
0x19B	0x41 - OpenStartElementToken (more Bit)	<EventID>. This does have a dependency (of 0x03). This means that if the template instance value at index 3 (the fourth value), in the ValueSpec, is of NULL type, then this element is to be omitted from the XML text. In this case, the type is non-NULL and so the element is included in the XML text representation. This value actually consists of two parts. The 0x01 indicates that it is an OpenStartElementToken. The 0x40 is the "more" bit, which indicates that there are additional attributes.
0x1BA	0x06 - AttributeToken	Attribute called EventIDQualifiers. Note that it does not appear in the XML text due to the OptionalSubstitutionToken specified next.
0x1D5	0x0E - OptionalSubstitutionToken	Optional substitution of the value specified at index 4 in the value spec. Looking forward into the TemplateInstanceData shows that this value is of NULL type, and so the enclosing attribute is not included in the XML text representation.
0x1D9	0x02 - CloseStartElementToken	Close <EventID> start tag.
0x1DA	0x0E - OptionalSubstitutionToken	OptionalSubstitution of the value specified at index 3 in the value spec. The value is 100 (in decimal).
0x4E4	0x01 - OpenStartElementToken	<UserData> start tag. It specifies that it is dependent on the value at index 0x13 in the value spec. This value is the BinXml for the inner template <MyEvent>. Because it is present, <UserData> is included in the XML representation.
0x502	0x0E - OptionalSubstitutionToken	This is the substitution for the BinXml, and its expected type is BinXmlType. The index into the value spec is 0x13.
0x506	0x04 - EndElementToken	End <UserData>.
0x507	0x04 - EndElementToken	End <Event>.

Token offset	Token type	Comments on encoding
0x508	0x00 - EOFToken	EOF for the outer template definition.
0x5A0	0x0F - FragmentHeaderToken	This is actually the last value that is specified in the outer TemplateInstance; however, because this value is itself BinXml, it starts with an (optional) header token and ends with an EOFToken.
0x5A4	0x0C - TemplateInstanceToken	For the inner template instance <MyEvent>, the TempleDefByteLength is 0x154 and the template definition itself starts at 0x5BA. This means that end of template definition will be at offset 0x5BA + 0x154 = 0x70E (which is the offset of the start of the TemplateInstanceData). The ValueSpec of the TemplateInstanceData specifies that there are 2 values with a total length of 8 bytes. This length is calculated by adding up all the lengths of the values specified in the value spec entries. The actual raw values of the template instance data start just after the value spec entries (at offset 0x71A). Adding the offset 0x71A to 0x8 bytes leaves us at the EOFToken for the inner fragment containing the TemplateInstance.
0x722	0x00 - EOFToken	EOF for the inner TemplateInstance.
0x723	0x00 - EOFToken	EOF for the outer TemplateInstance.

4.9 Render Localized Event Message Example

In this example, the client asks to get the event description from a known publisher. This involves the following steps:

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the **EvtRpcGetPublisherMetadata** method (section 3.1.4.25) to open the publisher metadata context handle.

```
error_status_t EvtRpcGetPublisherMetadata(
  [in] RPC_BINDING_HANDLE binding = {binding handle from step 1.},
  [in, unique, range(0, MAX_RPC_PUBLISHER_ID_LENGTH), string]
    LPCWSTR publisherId = "Microsoft-Windows-TestProvider",
  [in, unique, range(0, MAX_RPC_FILE_PATH_LENGTH), string] LPCWSTR logFilePath = NULL,
  [in] LCID locale = 1033,
  [in] DWORD flags = 0,
  [out] EvtRpcVariantList* pubMetadataProps,
  [out, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA* pubMetadata
);
```

3. In the response to the client call, the server finds the registered publisher "Microsoft-Windows-TestProvider" and opens its resource file. The server then creates a publisher metadata object, which contains the publisher name "Microsoft-Windows-TestProvider", the resource file location such as "c:\windows\system32\TestProvider.dll", the opened file handle, and the locale value 1033. The server then casts the object into the publisher metadata context handle.

   At the same time, the server reads the publisher resource file and extracts some of the publisher metadata and saves them in the *pubMetadataProps* parameter. Suppose this test publisher

declares two channels: "Microsoft-Windows-TestProvider/Operational" and "Microsoft-Windows-TestProvider/Admin". The publisher message file and parameter file are the same file as the resource file (a publisher usually uses the same file for all the resource, message, and parameter files). Then the data in *pubMetadataProps* will look as follows:

```
EvtCarTypeGuid   {836e133c-493c-4885-a780-4f0c61430fb9}
EvtVarTypeString  c:\windows\system32\TestProvider.dll
EvtVarTypeString  c:\windows\system32\TestProvider.dll
EvtVarTypeString  c:\windows\system32\Testrovider.dll

EvtTypeStringArray
    2 (array count)
    Microsoft-Windows-TestProvider/Operational
    Microsoft-Windows-TestProvider/Admin

EvtVarTypeUInt32Array
    2 (array count)
    0
    1

EvtVarTypeUInt32Array
    2 (array count)
    1
    2

EvtVarTypeUInt32Array
    2 (array count)
    0
    0

EvtVarTypeUInt32Array
    2 (array count)
    1001 (message Id for the channel)
    1002 (message Id for the channel)
```

4. After the client gets the publisher metadata context handle, it calls the **EvtRpcMessageRender** method (section 3.1.4.31) to render the desired event description.

```
error_status_t EvtRpcMessageRender(
  [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA pubCfgObj = {handlefrom step 2},
  [in, range(1, MAX_RPC_EVENT_ID_SIZE)] DWORD sizeEventId = sizeof(EVENT_DESCRIPTOR),
  [in, size_is(sizeEventId)] BYTE* eventId = {pointer to the event descriptor for an event},
  [in] DWORD messageId = 0,
  [in] EvtRpcVariantList* values = {pointer to values which will be used for substituion},
  [in] DWORD flags = 0x00000001 ({Format the event),
  [in] DWORD maxSizeString = 1024,
  [out] DWORD* actualSizeString,
  [out] DWORD* neededSizeString,
  [out, size_is(,*actualSizeString), range(0, MAX_RPC_RENDERED_STRING_SIZE)] BYTE** string,
  [out] RpcInfo* error
);
```

For the *eventId* parameter in this example, the values can look as follows:

```
0x0010   --- EventId
 0x02       --- Level
 0x00       --- Channel
 0x20       --- OpCode
 0x1000     --- Task
 0x8000000000000000 --- Keyword
```

5. In response to the client call, the server finds the event according to the passing event descriptor and reads out the raw event description strings from the provider publisher resource file. Because in step 2, the client requests the locale value as 1033, the server opens the English publisher resource file. Suppose the raw event description is "The system has been restarted after applying the updates of %1". The server then reads the data from the values provided by the client (assume it is "Adobe Flash") and replaces the %1 with the value it reads out. Thus, the returned string is:

   "The system has been restarted after applying the updates of Adobe Flash".

6. Later, if the client needs to get the localized message for the event level, it calls the same **EvtRpcMessageRender** method (section 3.1.4.31) with the same parameters except the flags value is 0x00000002.

7. In response to the client call, the server finds the event according to the passing event descriptor and reads out the level value. The level is 2, which means it falls into the system defined category. Suppose the system defined string for a level with the value 2 is "Error" for English. Thus, the resulting string is "Error".

8. When the client is done, it closes the publisher metadata handle by calling **EvtRpcClose** (section 3.1.4.33). In this call, the server frees all resources related to the publisher and closes the resource file.

```
error_status_t EvtRpcClose(
   [in, out, context_handle] void** handle = {publisher metadata handle}
);
```

4.10 Get Publisher List Example

In this example, the client obtains a list of registered publishers on the server. This involves the following steps.

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the EvtRpcGetPublisherList (section 3.1.4.23) method to receive the results.

```
error_status_t EvtRpcGetPublisherList(
  [in] RPC_BINDING_HANDLE binding = {handle from step 1},
  [in] DWORD flags = 0,
  [out] DWORD* numPublisherIds,
  [out, size_is(,*numPublisherIds), range(0, MAX_RPC_PUBLISHER_COUNT), string]
       LPWSTR** publisherIds
);
```

3. The server then goes to the publisher table and enumerates all the publisher names from the table to fill the *publisherIds* parameter as the result. At the same time, the *numPublisherIds* parameter is also set to the number of publishers in the server. Assuming the server has four publishers, a sample result for *publisherIds* looks as follows:

   "TestPublisher""Microsoft-Windows-EventLog""NTFSProvider""Microsoft-Windows-Firewall".

   The *numPublisherIds* is set to 4.

4.11 Get Channel List Example

In this example, the client tries to obtain a list of registered channels on the server.

This involves the following steps:

1. The client registers with RPC to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information on how to get the RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the **EvtRpcGetChannelList** method (section 3.1.4.20) to receive the results.

```
error_status_t EvtRpcGetChannelList(
  [in] RPC_BINDING_HANDLE binding = {handle from step 1},
  [in] DWORD flags = 0,
  [out] DWORD* numChannelPaths,
  [out, size_is(,*numChannelPaths), range(0, MAX_RPC_CHANNEL_COUNT), string]
     LPWSTR** channelPaths
);
```

3. The server then goes to the channel table and enumerates all the channel names from the table to fill the *channelPaths* parameter as the result. At the same time, the *numPublisherIds* parameter is also set to the number of publishers in the server. Assuming that the server has 5 channels, a sample resulting value in *channelPaths* might looks like the following:

```
"Application""System""Microsoft-Windows-EventLog/Admin""Microsoft-Windows-
NTFS/operational""Setup".
```

   In this case, the *numChannelPaths* value is 5.

4.12 Get Event Metadata Example

In this example, the client retrieves the event metadata information from a known publisher on the server.

This involves the following steps:

1. The client registers with RPS to obtain an RPC binding handle to the service based on the endpoint information specified in section 2.1. For information about how to obtain an RPC binding handle, see [MSDN-BNDHNDLS].

2. The client calls the **EvtRpcGetPublisherMetadata** method (section 3.1.4.25) to obtain a publisher metadata context handler.

```
error_status_t EvtRpcGetPublisherMetadata(
  [in] RPC_BINDING_HANDLE binding = {handle from step 1},
  [in, unique, range(0, MAX_RPC_PUBLISHER_ID_LENGTH), string]
    LPCWSTR publisherId = L"Microsoft-Windows-SamplePublisher",
  [in, unique, range(0, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR logFilePath = NULL,
  [in] LCID locale = 1033,
  [in] DWORD flags = 0,
  [out] EvtRpcVariantList* pubMetadataProps,
  [out, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA* pubMetadata
);
```

3. The server then opens the publisher resource file for the publisher whose name is "Microsoft-Windows-SamplePublisher" and creates a publisher metadata object. In this example, the publisher's resource file is %SystemDrive%\windows\SamplePublisher.dll; thus, the server sets the values for the data fields in the publisher metadata object as follows:

   HandleType: 4. Indicates the publisher metadata type.

   ResourceFile: %SystemDrive%\windows\SamplePublisher.dll.

   MessageFile: %SystemDrive%\windows\SamplePublisher.dll.

   ParameterFile: %SystemDrive\windows\SamplePublisher.dll.

   Locale: 1033.

   ResourceFileHandle: 0x00000AF0. Indicates the handle for the resource file.

   The server also fills the *pubMetadataProps* parameter with an EvtRpcVariantList containing 29 EvtRpcVariants as specified in section 3.1.4.25. The server obtains the data for these EvtRpcVariants from two sources, its publisher table and the publisher resource file. The server locates the publisher entry in its publisher table based on the specified publisherId parameter from the client. The server reads the publisherGUID, ResourceFilePath, ParameterFilePath, MessageFilePath, ChannelReferenceIndex, ChannelReferenceID, and ChannelReferenceFlags values directly from the publisher entry in the publisher table. The server locates the channel information in the publisher resource file in order to obtain the channel name strings and channel message IDs, which correspond to the ChannelReferencePath and ChannelReferenceMessageID entries of the *pubMetadataProps* list. In this example, the data is as follows:

```
[0]
15 ---- EvtVarTypeGuid
{59206ea5-6655-4ffa-8426-a2ce213b26f5}   ---- The publisher GUID
[1]
1  ---- EvtVarTypeString
"%SystemDrive%\windows\SamplePublisher.dll"
[2]
1  ---- EvtVarTypeString
"%SystemDrive%\windows\SamplePublisher.dll"
[3]
1  ---- EvtVarTypeString
"%SystemDrive%\windows\SamplePublisher.dll"
[4]
0  ---- EvtVarTypeNull
[5]
0  ---- EvtVarTypeNull
[6]
0  ---- EvtVarTypeNull
[7]
5  ---- ArrayCount
1  ---- EvtVarTypeString
"Application"
1  ---- EvtVarTypeString
"System"
1  ---- EvtVarTypeString
"Microsoft-Windows-EventLog/Admin"
1  ---- EvtVarTypeString
"Microsoft-Windows-NTFS/operational"
1  ---- EvtVarTypeString
"Setup"
[8]
5  ---- ArrayCount
8  ---- EvtVarTypeUint32
10 ---- ChannelReferenceIndex
8  ---- EvtVarTypeUint32
10 ---- ChannelReferenceIndex
8  ---- EvtVarTypeUint32
```

```
10 ---- ChannelReferenceIndex
8  ---- EvtVarTypeUint32
10 ---- ChannelReferenceIndex
8  ---- EvtVarTypeUint32
10 ---- ChannelReferenceIndex
[9]
5  ---- ArrayCount
8  ---- EvtVarTypeUint32
0 ---- ChannelreferenceID
8  ---- EvtVarTypeUint32
1 ---- ChannelreferenceID
8  ---- EvtVarTypeUint32
2 ---- ChannelreferenceID
8  ---- EvtVarTypeUint32
3 ---- ChannelreferenceID
8  ---- EvtVarTypeUint32
4 ---- ChannelreferenceID
[10]
5  ---- ArrayCount
8  ---- EvtVarTypeUint32
0 ---- ChannelreferenceFlags
8  ---- EvtVarTypeUint32
0 ---- ChannelreferenceFlags
8  ---- EvtVarTypeUint32
0 ---- ChannelreferenceFlags
8  ---- EvtVarTypeUint32
0 ---- ChannelreferenceFlags
8  ---- EvtVarTypeUint32
0 ---- ChannelreferenceFlags
[11]
5  ---- ArrayCount
8  ---- EvtVarTypeUint32
10000 ---- ChannelreferenceMessageID
8  ---- EvtVarTypeUint32
10001 ---- ChannelreferenceMessageID
8  ---- EvtVarTypeUint32
10002 ---- ChannelreferenceMessageID
8  ---- EvtVarTypeUint32
10003 ---- ChannelreferenceMessageID
8  ---- EvtVarTypeUint32
10004 ---- ChannelreferenceMessageID
[12]
0  ---- EvtVarTypeNull
[13]
0  ---- EvtVarTypeNull
[14]
0  ---- EvtVarTypeNull
[...]  ---- Entries 15 through 27, which are also EvtVarTypeNull, have been omitted
[28]
0  ---- EvtVarTypeNull
```

4. The server assigns the pointer of the publisher metadata object to the output parameter *pubMetadata* as the publisher metadata context handle.

5. After obtaining the publisher metadata context handle, the client calls the **EvtRpcGetEventMetadataEnum** method (section 3.1.4.27) to open the enumeration for the publisher's event metadata.

```
error_status_t EvtRpcGetEventMetadataEnum(
  [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA pubMetadata = {handle from step
2},
  [in] DWORD flags = 0,
  [in, unique, range(0, MAX_RPC_FILTER_LENGTH), string]
    LPCWSTR reservedForFilter = NULL,
  [out, context_handle] PCONTEXT_HANDLE_EVENT_METADATA_ENUM* eventMetaDataEnum
);
```

6. The server reads the publisher resource file and locates the event metadata section (as specified in section 3.1.1.14). The server then creates the event metadata object. After the server creates the object, it sets the values of the data fields in the object as follows:

   HandleType: 5. Indicates the event metadata type.

   EventsMetaData: 0x001ABEC8. A pointer to the event metadata section.

   Enumerator: 0

7. The server assigns the pointer of the event metadata object to the output parameter *eventMetadataEnum* as the event metadata enumeration context handle.

8. After obtaining the eventMetaDataEnum context handle, the client calls the **EvtRpcGetNextEventMetadata** method (section 3.1.4.28) to get the event metadata information.

```
error_status_t EvtRpcGetNextEventMetadata(
  [in, context_handle] PCONTEXT_HANDLE_EVENT_METADATA_ENUM eventMetaDataEnum = {handle from
step 3},
  [in] DWORD flags = 0,
  [in] DWORD numRequested = 2,
  [out] DWORD* numReturned,
  [out, size_is(,*numReturned), range(0, MAX_RPC_EVENT_METADATA_COUNT)]
    EvtRpcVariantList** eventMetadataInstances
);
```

   In this example method call, the client requests metadata for the first two events.

9. The server reads the metadata for the first two events from the publisher's resource file and fills the data into the *eventMetadataInstances* output parameter as follows:

```
8          ---- The count of the EvtRpcVariant of the first EvtRpcVariant list.
1001        ---- The EventID value.
1           ---- The Event version.
0           ---- The ChannelIndex value.
16          ---- The Level value.
10          ---- The Opcode value.
5           ---- The Task value.
0xFFFFFFFF  ---- The keywords mask value.
10001       ---- The MessageID value for the event description string.
<template tid="T22">
<data inType="win:UInt32" name="ErrorCode"/>
<data inType="win:UnicodeString" name="Path"/>
<UserData>
<Error Code="%1"/>
<ChannelPath>%2</ChannelPath>
</UserData>
</template> ---- The EvtVarTypeString value.

8           ---- The count of the EvtRpcVariant of the second EvtRpcVariant list.
1002        ---- The EventID value.
1           ---- The Event version.
0           ---- The ChannelIndex value.
16          ---- The Level value.
11          ---- The Opcode value.
6           ---- The Task value.
0xFFFFFFFF  ---- The keywords mask value.
10002       ---- The MessageID value for the event description string.
```

```
<template tid="T22">
<data inType="win:UInt32" name="ErrorCode"/>
<data inType="win:UnicodeString" name="Path"/>
<data inType="win:UnicodeString" name="NewLogFilePath"/>
<UserData>
<Error Code="%1"/>
<ChannelPath>%2</ChannelPath>
<NewLogFilePath>%3</NewLogFilePath>
</template> ---- The EvtVarTypeString value.
```

10. The client can call the **EvtRpcGetNextEventMetadata** method repeatedly to obtain metadata for additional events.

11. When the client finishes, it calls the **EvtRpcClose** method (section 3.1.4.33) to close both the event metadata enumeration context handle and the publisher metadata context handle.

```
error_status_t EvtRpcClose(
  [in, out, context_handle] void** handle = eventMetaDataEnum
);

error_status_t EvtRpcClose(
  [in, out, context_handle] void** handle = pubMetaData
);
```

4.13 Publisher Table and Channel Table Example

A publisher table is a list of publishers. The following example shows a publisher table with two entries.

```
{0063715b-eeda-4007-9429-ad526f62696e}    -------------    Publisher ID
"Microsoft-Windows-Services"                              ------  Publisher Name
    "%SystemRoot%\system32\services.exe"                  ------  Resource File
    "%SystemRoot%\system32\services.exe"                  ------  Message File
    "w                                                    ------ Parameter File(empty)
Channels
    1   ----------  channel count
         0x10  --------- channel ID for the channel 1
         0     --------- channel flags for the channel 1
         0     --------- channel start index for the channel 1
         "Microsoft-Windows-Services/Operational"  --------  channel name for channel 1

{134ea407-755d-4a93-b8a6-f290cd155023}        -------------    Publisher ID
"Microsoft-Windows-HomeGroup-ControlPanel"    ------  Publisher Name
    "%SystemRoot%\system32\hgcpl.dll"         ------  Resource File
    "%SystemRoot%\system32\hgcpl.dll"         ------  Message File
    ""                                        ------ Parameter File(empty)
Channels
    2   ----------  channel count
         0x10  --------- channel ID for the channel 1
         0     --------- channel flags for the channel 1
         0     --------- channel start index for the channel 1
         "Microsoft-Windows-HomeGroup-ControlPanel/operational" ---- channel name for channel
1
    0x11  --------- channel ID for the channel 2
         0     --------- channel flags for the channel 2
         0     --------- channel start index for the channel 2
         "Microsoft-Windows-HomeGroup-ControlPanel/admin"   ---- channel name for channel 2
```

A channel table is a list of registered channels on the server. The following example shows a channel table with one channel entry:

```
ForwardedEvents             ----  Name of the channel
Enabled: 0
Isolation: 2
Type: 1
OwningPublisher: {b977cf02-76f6-df84-cc1a-6a4b232322b6}
Classic: 0
Access:    O:BAG:SYD:(A;;0x2;;;S-1-15-2-
1)(A;;0xf0007;;;SY)(A;;0x7;;;BA)(A;;0x7;;;SO)(A;;0x3;;;IU)(A;;0x3;;;SU)(A;;0x3;;;S-1-5-
3)(A;;0x3;;;S-1-5-33)(A;;0x1;;;S-1-5-32-573)
Retention: 0
Autobackup: 0
MaxSize: 0x01400000
FilePath: "%SystemRoot%\system32\winevt\logs\forwardedevents.evtx"
Level:    0x0000FFFF
Keywords: 0xFFFFFFFFFFFFFFFF
ControlGuid: {00000000-0000-0000-0000-000000000000}
BufferSize: 0x000000000000FFFF
MinBuffers: 4
MaxBuffers: 10
Latency: 1
ClockType: 0
SIDType: 1
FileMax: 16
```

Note The list of the publishers is not in the channel table entry because the channel table entry is built at runtime using the publisher table and the channel name.

4.14 Backup and Archive the Event Log Example

In this example, the client wants to export all the events in the application channel into a backup event log file and then bring the backup file to another computer to view the events with no publisher registered on the destination computer. This involves the following steps:

1. The client calls the **EvtRpcRegisterControllableOperation** method (section 3.1.4.35) to get an operation control handle.

```
error_status_t EvtRpcRegisterControllableOperation(
  [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* handle
);
```

2. The client calls the **EvtRpcExportLog** method (section 3.1.4.17) to export the events into a backup log file.

```
error_status_t EvtRpcExportLog(
  [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control = {handle from step 1},
  [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath = L"Application",
  [in, range(1, MAX_RPC_QUERY_LENGTH), string]
    LPCWSTR query = L"*",
  [in, range(1, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR backupPath = L"c:\\backup\\application.evtx",
  [in] DWORD flags = 0x00000001 (EvtExportLogChannelPath),
  [out] RpcInfo* error
);
```

3. In the implementation of the server, it opens the application channel and reads every event and copies the events from the channel into the file "c:\backup\application.evtx". Now the backup event log file contains all the events from the application channel except the localized strings for each event's level, task, opcode, keyword, and description.

4. To get those localized strings, the client calls the **EvtRpcLocalizeExportLog** method (section 3.1.4.18) to save the localized strings in a separate file in a subdirectory of the directory where the backup file is located.

```
error_status_t EvtRpcLocalizeExportLog(
  [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control = {handle from step 1},
  [in, range(1, MAX_RPC_FILE_PATH_LENGTH), string]
    LPCWSTR logFilePath = L"c:\\backup\\application.evtx",
  [in] LCID locale = 1033,
  [in] DWORD flags = 0,
  [out] RpcInfo* error
);
```

After the server returns, there file is created on the server under with the name "c:\backup\LocalMedadata\Application_1033.MTA". The file contains all the localized English strings for all events.

5 Security

5.1 Security Considerations for Implementers

Implementers ~~MUST take care~~are required to enforce the read/write permissions, as specified in section 3.1.4.21, to prevent unauthorized access to event logs.

Servers ~~SHOULD~~ authenticate the caller and verify that the caller has proper access before returning a handle. When the handle is subsequently used, the server ~~SHOULD verify~~verifies that the client created the handle, that it was created by a method of this interface, and that the handle is appropriate for the operation.

5.2 Index of Security Parameters

Security parameter	Section
Authentication service	Transport (section 2.1)

6 Appendix A: Full IDL

For ease of implementation, the full IDL is provided as follows, where "ms-dtyp.idl" is the IDL found in [MS-DTYP] Appendix A. Please note that the binding handle is commented out for each method with the binding handle as the first parameter because that parameter is automatically added by the IDL compiler to the method signature in the resulting header (.h) file.

```
import "ms-dtyp.idl";

const int MAX_PAYLOAD = 2 * 1024 * 1024;
const int MAX_RPC_QUERY_LENGTH = MAX_PAYLOAD / sizeof(WCHAR);
const int MAX_RPC_CHANNEL_NAME_LENGTH = 512;
const int MAX_RPC_QUERY_CHANNEL_SIZE = 512;
const int MAX_RPC_EVENT_ID_SIZE = 256;
const int MAX_RPC_FILE_PATH_LENGTH = 32768;
const int MAX_RPC_CHANNEL_PATH_LENGTH = 32768;
const int MAX_RPC_BOOKMARK_LENGTH = MAX_PAYLOAD / sizeof(WCHAR);
const int MAX_RPC_PUBLISHER_ID_LENGTH = 2048;
const int MAX_RPC_PROPERTY_BUFFER_SIZE = MAX_PAYLOAD;
const int MAX_RPC_FILTER_LENGTH = MAX_RPC_QUERY_LENGTH;
const int MAX_RPC_RECORD_COUNT = 1024;
const int MAX_RPC_EVENT_SIZE = MAX_PAYLOAD;
const int MAX_RPC_BATCH_SIZE = MAX_PAYLOAD;
const int MAX_RPC_RENDERED_STRING_SIZE = MAX_PAYLOAD;
const int MAX_RPC_CHANNEL_COUNT = 8192;
const int MAX_RPC_PUBLISHER_COUNT = 8192;
const int MAX_RPC_EVENT_METADATA_COUNT = 256;
const int MAX_RPC_VARIANT_LIST_COUNT = 256;
const int MAX_RPC_BOOL_ARRAY_COUNT = MAX_PAYLOAD / sizeof(BOOL);
const int MAX_RPC_UINT32_ARRAY_COUNT = MAX_PAYLOAD / sizeof(UINT32);
const int MAX_RPC_UINT64_ARRAY_COUNT = MAX_PAYLOAD / sizeof(UINT64);
const int MAX_RPC_STRING_ARRAY_COUNT = MAX_PAYLOAD / 512;
const int MAX_RPC_GUID_ARRAY_COUNT = MAX_PAYLOAD / sizeof(GUID);
const int MAX_RPC_STRING_LENGTH = MAX_PAYLOAD / sizeof(WCHAR);

[
    uuid (f6beaff7-1e19-4fbb-9f8f-b89e2018337c),
    version(1.0),
    pointer_default(unique)
]
interface IEventService
{
    typedef [context_handle] void* PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION;
    typedef [context_handle] void* PCONTEXT_HANDLE_LOG_QUERY;
    typedef [context_handle] void* PCONTEXT_HANDLE_LOG_HANDLE;
    typedef [context_handle] void* PCONTEXT_HANDLE_OPERATION_CONTROL;
    typedef [context_handle] void* PCONTEXT_HANDLE_PUBLISHER_METADATA;
    typedef [context_handle] void* PCONTEXT_HANDLE_EVENT_METADATA_ENUM;




    typedef struct tag_RpcInfo
    {
      DWORD m_error,
            m_subErr,
            m_subErrParam;
    } RpcInfo;

    typedef struct _BooleanArray
    {
        [range(0, MAX_RPC_BOOL_ARRAY_COUNT)] DWORD count;
        [size_is(count)] boolean* ptr;
    } BooleanArray;

    typedef struct _UInt32Array
```

```
{
    [range(0, MAX_RPC_UINT32_ARRAY_COUNT)] DWORD count;
    [size_is(count)] DWORD* ptr;
} UInt32Array;

typedef struct _UInt64Array
{
    [range(0, MAX_RPC_UINT64_ARRAY_COUNT)] DWORD count;
    [size_is(count)] DWORD64* ptr;
} UInt64Array;

typedef struct _StringArray
{
    [range(0, MAX_RPC_STRING_ARRAY_COUNT)] DWORD count;
    [size_is(count),string] LPWSTR *ptr;
} StringArray;

typedef struct _GuidArray
{
    [range(0, MAX_RPC_GUID_ARRAY_COUNT)] DWORD count;
    [size_is(count)] GUID* ptr;
} GuidArray;

typedef [v1_enum] enum tag_EvtRpcVariantType
{
    EvtRpcVarTypeNull = 0,
    EvtRpcVarTypeBoolean,
    EvtRpcVarTypeUInt32,
    EvtRpcVarTypeUInt64,
    EvtRpcVarTypeString,
    EvtRpcVarTypeGuid,
    EvtRpcVarTypeBooleanArray,
    EvtRpcVarTypeUInt32Array,
    EvtRpcVarTypeUInt64Array,
    EvtRpcVarTypeStringArray,
    EvtRpcVarTypeGuidArray

} EvtRpcVariantType;

typedef [v1_enum] enum tag_EvtRpcAssertConfigFlags
{
    EvtRpcChannelPath = 0,
    EvtRpcPublisherName = 1

} EvtRpcAssertConfigFlags;


cpp_quote("#define EvtRpcSubscribePull 0x10000000")

cpp_quote("#define EvtRpcVarFlagsModified 0x0001")

typedef struct tag_EvtRpcVariant
{
    EvtRpcVariantType type;
    DWORD flags;
    [switch_is(type)] union
    {
        [case(EvtRpcVarTypeNull)] int nullVal;
        [case(EvtRpcVarTypeBoolean)] boolean booleanVal;
        [case(EvtRpcVarTypeUInt32)] DWORD uint32Val;
        [case(EvtRpcVarTypeUInt64)] DWORD64 uint64Val;
        [case(EvtRpcVarTypeString)] [string]LPWSTR stringVal;
        [case(EvtRpcVarTypeGuid)] GUID* guidVal;
        [case(EvtRpcVarTypeBooleanArray)] BooleanArray booleanArray;
        [case(EvtRpcVarTypeUInt32Array)] UInt32Array uint32Array;
        [case(EvtRpcVarTypeUInt64Array)] UInt64Array uint64Array;
        [case(EvtRpcVarTypeStringArray)] StringArray stringArray;
        [case(EvtRpcVarTypeGuidArray)] GuidArray guidArray;
    };
} EvtRpcVariant;
```

```
typedef struct tag_EvtRpcVariantList
{
    [range(0, MAX_RPC_VARIANT_LIST_COUNT)] DWORD count;
    [size_is(count)] EvtRpcVariant* props;
} EvtRpcVariantList;

typedef struct tag_EvtRpcQueryChannelInfo
{
    LPWSTR name;
    DWORD status;
} EvtRpcQueryChannelInfo;

error_status_t EvtRpcRegisterRemoteSubscription(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR channelPath,
    [in, range(1, MAX_RPC_QUERY_LENGTH),string] LPCWSTR query,
    [in, unique, range(0, MAX_RPC_BOOKMARK_LENGTH),string] LPCWSTR bookmarkXml,
    [in] DWORD flags,
    [out, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION* handle,
    [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* control,
    [out] DWORD* queryChannelInfoSize,
    [out, size_is(,*queryChannelInfoSize),
          range(0, MAX_RPC_QUERY_CHANNEL_SIZE)]
              EvtRpcQueryChannelInfo** queryChannelInfo,
    [out] RpcInfo *error);

error_status_t EvtRpcRemoteSubscriptionNextAsync(
    [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle,
    [in] DWORD numRequestedRecords,
    [in] DWORD flags,
    [out] DWORD* numActualRecords,
    [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
        DWORD** eventDataIndices,
    [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
        DWORD** eventDataSizes,
    [out] DWORD* resultBufferSize,
    [out, size_is(,*resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
        BYTE** resultBuffer );

error_status_t EvtRpcRemoteSubscriptionNext(
    [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle,
    [in] DWORD numRequestedRecords,
    [in] DWORD timeOut,
    [in] DWORD flags,
    [out] DWORD* numActualRecords,
    [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
        DWORD** eventDataIndices,
    [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
        DWORD** eventDataSizes,
    [out] DWORD* resultBufferSize,
    [out, size_is(,*resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
        BYTE** resultBuffer );

error_status_t EvtRpcRemoteSubscriptionWaitAsync(
    [in, context_handle] PCONTEXT_HANDLE_REMOTE_SUBSCRIPTION handle );

error_status_t EvtRpcRegisterControllableOperation(
    [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* handle );

error_status_t EvtRpcRegisterLogQuery(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, unique, range(0, MAX_RPC_CHANNEL_PATH_LENGTH),string] LPCWSTR path,
    [in, range(1, MAX_RPC_QUERY_LENGTH),string] LPCWSTR query,
    [in] DWORD flags,
    [out, context_handle] PCONTEXT_HANDLE_LOG_QUERY* handle,
    [out, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL* opControl,
    [out] DWORD* queryChannelInfoSize,
    [out, size_is(,*queryChannelInfoSize),
          range(0, MAX_RPC_QUERY_CHANNEL_SIZE)]
```

```
            EvtRpcQueryChannelInfo** queryChannelInfo,
        [out] RpcInfo *error );

    error_status_t EvtRpcClearLog(
        [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control,
        [in, range(0, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR channelPath,
        [in, unique, range(0, MAX_RPC_FILE_PATH_LENGTH),string] LPCWSTR backupPath,
        [in] DWORD flags,
        [out] RpcInfo *error );

    error_status_t EvtRpcExportLog(
        [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control,
        [in, unique, range(0, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR channelPath,
        [in, range(1, MAX_RPC_QUERY_LENGTH),string] LPCWSTR query,
        [in, range(1, MAX_RPC_FILE_PATH_LENGTH),string] LPCWSTR backupPath,
        [in] DWORD flags,
        [out] RpcInfo *error );

    error_status_t EvtRpcLocalizeExportLog(
        [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL control,
        [in, range(1, MAX_RPC_FILE_PATH_LENGTH),string] LPCWSTR logFilePath,
        [in] LCID locale,
        [in] DWORD flags,
        [out] RpcInfo *error );

    error_status_t EvtRpcMessageRender(
        [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA pubCfgObj,
        [in, range(1, MAX_RPC_EVENT_ID_SIZE)] DWORD sizeEventId,
        [in, size_is(sizeEventId)] BYTE *eventId,
        [in] DWORD messageId,
        [in] EvtRpcVariantList *values,
        [in] DWORD flags,
        [in] DWORD maxSizeString,
        [out] DWORD *actualSizeString,
        [out] DWORD *neededSizeString,
        [out, size_is(,*actualSizeString), range(0, MAX_RPC_RENDERED_STRING_SIZE)]
            BYTE** string,
        [out] RpcInfo *error );

    error_status_t EvtRpcMessageRenderDefault(
        /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
        [in, range(1, MAX_RPC_EVENT_ID_SIZE)] DWORD sizeEventId,
        [in, size_is(sizeEventId)] BYTE *eventId,
        [in] DWORD messageId,
        [in] EvtRpcVariantList *values,
        [in] DWORD flags,
        [in] DWORD maxSizeString,
        [out] DWORD *actualSizeString,
        [out] DWORD *neededSizeString,
        [out, size_is(,*actualSizeString), range(0, MAX_RPC_RENDERED_STRING_SIZE)]
            BYTE** string,
        [out] RpcInfo *error );

    error_status_t EvtRpcQueryNext(
        [in, context_handle] PCONTEXT_HANDLE_LOG_QUERY logQuery,
        [in] DWORD numRequestedRecords,
        [in] DWORD timeOutEnd,
        [in] DWORD flags,
        [out] DWORD* numActualRecords,
        [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
            DWORD** eventDataIndices,
        [out, size_is(,*numActualRecords), range(0, MAX_RPC_RECORD_COUNT)]
            DWORD** eventDataSizes,
        [out] DWORD* resultBufferSize,
        [out, size_is(,*resultBufferSize), range(0, MAX_RPC_BATCH_SIZE)]
            BYTE** resultBuffer );

    error_status_t EvtRpcQuerySeek(
        [in, context_handle] PCONTEXT_HANDLE_LOG_QUERY logQuery,
        [in] __int64 pos,
```

```
    [in, unique, range(0, MAX_RPC_BOOKMARK_LENGTH),string] LPCWSTR bookmarkXml,
    [in] DWORD timeOut,
    [in] DWORD flags,
    [out] RpcInfo *error );

error_status_t EvtRpcClose(
    [in, out, context_handle] void** handle );

error_status_t EvtRpcCancel(
    [in, context_handle] PCONTEXT_HANDLE_OPERATION_CONTROL handle );

error_status_t EvtRpcAssertConfig(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR path,
    [in] DWORD flags );

error_status_t EvtRpcRetractConfig(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR path,
    [in] DWORD flags );

error_status_t EvtRpcOpenLogHandle(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR channel,
    [in] DWORD flags,
    [out, context_handle] PCONTEXT_HANDLE_LOG_HANDLE* handle,
    [out] RpcInfo *error );

error_status_t EvtRpcGetLogFileInfo(
    [in, context_handle] PCONTEXT_HANDLE_LOG_HANDLE logHandle,
    [in] DWORD propertyId,
    [in, range(0, MAX_RPC_PROPERTY_BUFFER_SIZE)]
        DWORD propertyValueBufferSize,
    [out, size_is(propertyValueBufferSize)] BYTE * propertyValueBuffer,
    [out] DWORD* propertyValueBufferLength );

error_status_t EvtRpcGetChannelList(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in] DWORD flags,
    [out] DWORD* numChannelPaths,
    [out, size_is(,*numChannelPaths), range(0, MAX_RPC_CHANNEL_COUNT),string]
        LPWSTR** channelPaths );

error_status_t EvtRpcGetChannelConfig(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR channelPath,
    [in] DWORD flags,
    [out] EvtRpcVariantList* props );

error_status_t EvtRpcPutChannelConfig(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR channelPath,
    [in] DWORD flags,
    [in] EvtRpcVariantList* props,
    [out] RpcInfo *error );

error_status_t EvtRpcGetPublisherList(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in] DWORD flags,
    [out] DWORD* numPublisherIds,
    [out, size_is(,*numPublisherIds), range(0, MAX_RPC_PUBLISHER_COUNT),string]
        LPWSTR** publisherIds );

error_status_t EvtRpcGetPublisherListForChannel(
    /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
    [in] LPCWSTR channelName,
    [in] DWORD flags,
    [out] DWORD* numPublisherIds,
    [out, size_is(,*numPublisherIds), range(0, MAX_RPC_PUBLISHER_COUNT),string]
        LPWSTR** publisherIds );
```

```
    error_status_t EvtRpcGetPublisherMetadata(
        /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
        [in, unique, range(0, MAX_RPC_PUBLISHER_ID_LENGTH),string] LPCWSTR publisherId,
        [in, unique, range(0, MAX_RPC_FILE_PATH_LENGTH),string] LPCWSTR logFilePath,
        [in] LCID locale,
        [in] DWORD flags,
        [out] EvtRpcVariantList* pubMetadataProps,
        [out, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA* pubMetadata );

    error_status_t EvtRpcGetPublisherResourceMetadata(
        [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA handle,
        [in] DWORD propertyId,
        [in] DWORD flags,
        [out] EvtRpcVariantList* pubMetadataProps);

    error_status_t EvtRpcGetEventMetadataEnum(
        [in, context_handle] PCONTEXT_HANDLE_PUBLISHER_METADATA pubMetadata,
        [in] DWORD flags,
        [in, unique, range(0, MAX_RPC_FILTER_LENGTH),string] LPCWSTR reservedForFilter,
        [out, context_handle] PCONTEXT_HANDLE_EVENT_METADATA_ENUM* eventMetaDataEnum );

    error_status_t EvtRpcGetNextEventMetadata(
        [in, context_handle] PCONTEXT_HANDLE_EVENT_METADATA_ENUM eventMetaDataEnum,
        [in] DWORD flags,
        [in] DWORD numRequested,
        [out] DWORD* numReturned,
        [out, size_is(,*numReturned), range(0, MAX_RPC_EVENT_METADATA_COUNT)]
            EvtRpcVariantList** eventMetadataInstances );

    error_status_t EvtRpcGetClassicLogDisplayName(
        /* [in] RPC_BINDING_HANDLE binding, {the binding handle will be generated by MIDL} */
        [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH),string] LPCWSTR logName,
        [in] LCID locale,
        [in] DWORD flags,
        [out] LPWSTR* displayName );
}
```

7 Appendix B: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs.

~~Note: Some of the information in this section is subject to change because it applies to a preliminary product version, and thus may differ from the final version of the software when released. All behavior notes that pertain to the preliminary product version contain specific references to it as an aid to the reader.~~

- Windows Vista operating system
- Windows Server 2008 operating system
- Windows 7 operating system
- Windows Server 2008 R2 operating system
- Windows 8 operating system
- Windows Server 2012 operating system
- Windows 8.1 operating system
- Windows Server 2012 R2 operating system
- Windows 10 operating system
- Windows Server 2016 ~~Technical Preview~~ operating system

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

<1> Section 1.8.1: Windows prefixes the names of some of the channels it creates with the string Microsoft-Windows-. For more information, see [MSDN-EVENT].

<2> Section 1.8.2: Windows prefixes the names of some of the publishers it creates with the string Microsoft-Windows-. For more information, see [MSDN-EVENTS].

<3> Section 1.8.4: Windows uses only the values specified in [MS-ERREF] section 2.3.

<4> Section 3.1.1.12: In Windows based server implementation, the server leverages the context handle table provided by RPC. For more information about RPC context handles, see [MSDN-CH].

<5> Section 3.1.4: All errors are as specified in [MS-ERREF] section 2.3.

<6> Section 3.1.4.7.2: In a Windows implementation, the event definition is part of a compiled binary image, and as such is external to this protocol.

<7> Section 3.1.4.8: In a Windows server-based implementation, the server returns ERROR_NOT_FOUND (0x00000490) when the *bookmark* is invalid and the EvtSubscribeStrictrestrict flag is set. The server does not return an error if the bookmark is invalid and the EvtSubscribeStrict flag is not set. The server ignores the bookmark parameter.

<8> Section 3.1.4.8: In Windows server implementations, the server returns ERROR_EVT_INVALID_QUERY (0x00003A99).

<9> Section 3.1.4.8: In Windows, if query parameter is not null the server will attempt to determine if the channel is valid. If the channel string contains one or more invalid characters (any character whose ASCII value is less than 32 or character '<', '>', '|', '\', '"', ':', '', '*', '?'), the server will return ERROR_EVT_INVALID_CHANNEL_PATH (0x00003A98). If the channel does not exist, the server will return ERROR_EVT_CHANNEL_NOT_FOUND (0x00003A9F). If the channel is valid, and the non-null query parameter is invalid, the server will return ERROR_EVT_INVALID_QUERY (0x00003A99).

<10> Section 3.1.4.9: Windows Vista and Windows Server 2008 ignore this **flags** field.

<11> Section 3.1.4.9: In Windows, the server does not do a thorough validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space, and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. As such, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and could potentially cause system issues.

<12> Section 3.1.4.10: Windows Vista and Windows Server 2008 ignore this **flags** field.

<13> Section 3.1.4.11: In Windows, the server does not do a thorough validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space, and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. As such, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and could potentially cause system issues.

<14> Section 3.1.4.12: In Windows Vista and Windows Server 2008, the server does not validate the flags. It ignores any unrecognized flags, assumes that the path is a file if not specified, and iterates from oldest to newest if a direction flag is unspecified.

In Windows 7, Windows Server 2008 R2 operating system, Windows 8, Windows Server 2012, Windows 8.1, Windows Server 2012 R2, Windows 10, and Windows Server 2016~~ Technical Preview~~, the server does not completely validate the flags. It does not return an error when neither the EvtQueryChannelPath nor EvtQueryFilePath bits are set, and does not return an error when neither the 0x00000100 nor 0x00000200 bits are set. The server assumes that the path is a file if not specified, and iterates from oldest to newest if a direction flag is not specified.

<15> Section 3.1.4.12: In Windows, the server can omit the invalid channels.

<16> Section 3.1.4.13: Windows limits the *numRequestedRecords* to 1024. If *numRequestedRecords* is greater than 1024, ERROR_INVALID_PARAPMETER is returned.

<17> Section 3.1.4.13: Windows Vista and Windows Server 2008 ignore this flag.

<18> Section 3.1.4.13: In Windows, the server does not do a thorough validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space and that the pointer points to a buffer that states the correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. As such, it is possible to circumvent the server, especially if the handle has been obtained from a different method in the EventLog Remoting Protocol Version 6. In that case, the server's behavior is undefined and can potentially cause system issues.

<19> Section 3.1.4.14: In the Windows implementation, the sign of the *pos* parameter is validated against the seek direction by the server. Windows Vista and Windows Server 2008~~ will~~ return ERROR_NOT_FOUND (0x00000490) in the above cases if the EvtSeekStrict flag is set. If the EvtSeekStrict flag is not set, Windows Vista and Windows Server 2008 will not return an error in the

two cases above. In Windows Vista and Windows Server 2008, if the EvtSeekRelativeToFirst flag is set and the *pos* parameter has a negative value, the cursor of the result set remains at the first record. If the EvtSeekRelativeToLast flag is set and the *pos* parameter has a positive value, the cursor remains at the last record.

Except for Windows Vista and Windows Server 2008, Windows always returns ERROR_INVALID_PARAMETER (0x00000057) when errors are encountered validating the *pos* parameter and will not change the cursor position.

<20> Section 3.1.4.15: For more information on attributes, see [MSDN-FILEATT].

<21> Section 3.1.4.15: In Windows, the server does not do a thorough validation of the handle. It verifies that the handle can be transformed into a pointer in the appropriate address space, and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. As such, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<22> Section 3.1.4.16: Windows Vista and Windows Server 2008 ignore this **flags** field.

<23> Section 3.1.4.16: In the case of the failure of an internal function about which Windows doesn't receive detailed error information, it will fill the sub-error fields with 0xFFFFFFFF, which is often used as a generic error return code.

<24> Section 3.1.4.16: In Windows server implementation, the server uses the CreateFile function [MSDN-CreateFile] to create the backup file and return any error code the CreateFile function can possibly set to the last error code when it fails.

<25> Section 3.1.4.17: In the Windows implementation, the client API layer typically validates the flags, and the server does not. Therefore, the onus is on the RPC client either to validate flags or to restrict support to valid flag combinations.

<26> Section 3.1.4.17: In Windows Vista and Windows Server 2008, the server does not validate the flags. It will ignore any unrecognized flags; and will assume that the path is a file if not specified.

<27> Section 3.1.4.17: In a Windows server implementation, the server returns any possible error code from the last errors set by CreateFile function [MSDN-CreateFile] if the method fails.

<28> Section 3.1.4.18: Windows Vista and Windows Server 2008 ignore this **flags** field.

<29> Section 3.1.4.18: Windows can erroneously return ERROR_SUCCESS. In such cases, the fields of the RpcInfo structure "error" will be set to nonzero values to specify the detail error. For example, the function **EvtRpcLocalizeExportLog** can return ERROR_SUCCESS with the RpcInfo structure containing the error ERROR_EVT_MESSAGE_NOT_FOUND (0x00003AB3).

<30> Section 3.1.4.18: Windows can erroneously return ERROR_SUCCESS. In such cases, the fields of the RpcInfo structure "error" will be set to nonzero values to specify the detail error. For example, the function **EvtRpcLocalizeExportLog** can return ERROR_SUCCESS with the RpcInfo structure containing the error ERROR_EVT_MESSAGE_NOT_FOUND (0x00003AB3).

<31> Section 3.1.4.18: Server implementations on Windows return ERROR_INVALID_NAME when the logFilePath parameter is not valid for the underlying file system.

<32> Section 3.1.4.19: In the case of the failure of an internal function about which Windows does not receive detailed error information, it will fill the sub-error fields with 0xFFFFFFFF, which is often used as a generic error return code.

<33> Section 3.1.4.19: Windows Vista and Windows Server 2008 ignore this **flags** field.

<34> Section 3.1.4.20: Windows Vista and Windows Server 2008 ignore this **flags** field.

<35> Section 3.1.4.21: Windows Vista and Windows Server 2008 ignore this **flags** field.

<36> Section 3.1.4.21: The FileMax property is not supported by Windows Vista and Windows Server 2008. Windows Vista and Windows Server 2008 call EvtRpcGetChannelConfig to receive the EvtRpcVariantList structure. FileMax is ignored by Windows Vista and Windows Server 2008. The value of FileMax received in the EvtRpcVariantList structure is passed back unmodified by calls to EvtRpcPutChannelConfig on Windows 7, Windows Server 2008 R2, Windows Server 2012, Windows 8, Windows 8.1, Windows Server 2012 R2, Windows 10, and Windows Server 2016~~ Technical Preview~~.

<37> Section 3.1.4.22: In Windows Vista and Windows Server 2008, the server does not validate the flag.

<38> Section 3.1.4.22: Windows based implementations of this protocol use the ControlGuid property to identify the WPP provider, a special publisher that is used to log debugging events. The WPP provider is not intended for general use. See [MSDN-WPPST] for more information about this publisher.

<39> Section 3.1.4.22: Windows will erroneously return ERROR_SUCCESS. In such cases the fields of the RpcInfo structure "error" will be set to nonzero values.

<40> Section 3.1.4.22: Server implementations on Windows do not check whether the publisher is already registered in the publisher table and will return ERROR_SUCCESS for unregistered publishers.

<41> Section 3.1.4.22: In a Windows server implementation, the initial value for BufferSize is 64k. Initial MinBuffers value is twice the number of processors of the system. Initial MaxBuffers value is the MinBuffers value plus 22. The initial Latency value is 1 second. The initial clocktype value is 0 and the initial value for SIDType is 1.

<42> Section 3.1.4.23: Windows Vista and Windows Server 2008 ignore this **flags** field.

<43> Section 3.1.4.23: In Windows, the server uses registry to implement the publisher table. The security descriptor for the publisher table is provided by the Windows registry system.

<44> Section 3.1.4.24: Windows Vista and Windows Server 2008 ignore this **flags** field.

<45> Section 3.1.4.25: Windows Vista and Windows Server 2008 ignore this **flags** field.

<46> Section 3.1.4.25: In Windows, the server uses a registry entry to save the publisher in the publisher table. The security descriptor for the publisher is the security descriptor for the registry entry.

<47> Section 3.1.4.26: Windows Vista and Windows Server 2008 ignore this **flags** field.

<48> Section 3.1.4.26: In Windows, the server does not do a complete validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. Therefore, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<49> Section 3.1.4.26: In a Windows server implementation, the server does not return an error in this case and sets nothing in the *pubMetadataProps* parameter.

<50> Section 3.1.4.26: A Windows implementation wraps the RPC calls with an API layer that provides default values for metadata that are not supplied by the publisher. For example, Windows provides a helplink based on the executable name for a particular provider if that provider does not supply a helplink.

<51> Section 3.1.4.27: Windows Vista and Windows Server 2008 ignore this **flags** field.

<52> Section 3.1.4.27: In Windows, the server does not do a thorough validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space, and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. Therefore, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<53> Section 3.1.4.28: Windows Server 2008 does not validate this flag.

<54> Section 3.1.4.28: In Windows, the server does not do a thorough validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space and that the pointer points to a buffer that states a correct handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. Therefore, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<55> Section 3.1.4.28: In Windows, the method does not fail when there is no metadata to return.

<56> Section 3.1.4.29: In Windows, the server does not validate the path parameter, and will start a new, partially configured channel or publisher registration if supplied with an invalid name.

<57> Section 3.1.4.30: In Windows, the server only validates that the path parameter is syntactically correct; it does not validate that the channel exists. The server returns ERROR_SUCCESS (0x00000000) if it is passed a channel name which is syntactically correct but nonexistent.

<58> Section 3.1.4.30: In a Windows server implementation, the server returns ERROR_INVALID_PARAMETER (0x00000057). When the path is NULL, the server returns any possible error codes a RegOpenKeyEx function could return.

<59> Section 3.1.4.31: In the Windows implementation, substitution parameters are denoted by "%1", "%2", and so on. An event message can be "Error number %1 occurred on disk drive %2." To format this message, the client could specify the eventId that denotes this event in the eventId parameter and supply the strings "5" and "C:" in the values parameter. If the client supplied a buffer that is large enough in the strings parameter, the server would set that buffer to "Error number 5 occurred on disk drive C".

<60> Section 3.1.4.31: In Windows, the server does not do a complete validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space and that the pointer points to a buffer that states a correct handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. Therefore, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<61> Section 3.1.4.31: In a Windows server implementation, the server uses the FormatMessage function (see [MSDN-FMT]) to perform this task.

<62> Section 3.1.4.31: In a Windows server implementation, the server uses the FormatMessage function (see [MSDN-FMT]) to perform this task.

<63> Section 3.1.4.33: In Windows, the server does not do a complete validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. Therefore, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<64> Section 3.1.4.34: In Windows, the server does not do a complete validation of the handle. It verifies that the handle can be transformed into a pointer in the proper address space and that the pointer points to a buffer that states a correct context handle type, and fails with ERROR_INVALID_PARAMETER (0x00000057) if the handle type does not match. Therefore, it is possible to circumvent the server, especially if the handle has been obtained from a different method in this specification. In that case, the server's behavior is undefined and can potentially cause system issues.

<65> Section 3.1.4.36: Windows implementations of this protocol use the GetThreadPreferredUILanguages function [MSDN-GETTHDPREUILANG] to determine the fallback locale.

<66> Section 3.1.4.36: In Windows Vista and Windows Server 2008, the server does not validate the *flags* parameter. The server responds to flags 0x0 and 0x100, and ignores all others.

<67> Section 3.2.7: In a Windows client implementation, because the Windows Server operating system keeps its publisher table in the registry, the client accesses the registry directly and finds the resource file location for a given publisher and changes the registry value directly. The resource file location is saved in the registry: "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\<Publisher Identifier>\ResourceFileName". When the **EvtRpcAssertConfig** method (section 3.1.4.29) is issued, the server locates the same key, reads the *ResourceFileName*, and tries to open it. If the file can be opened, the Windows Server accepts the change and saves it. Otherwise, it reverts the registry change and discards the client's change.

8 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

9 Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V