[MS-EVEN6]:
EventLog Remoting Protocol
Version 6.0 Specification

Revision Summary

Date	Revision History	Revision Class	Comments
10/22/2006	0.01		MCPP Milestone 1 Initial Availability
01/19/2007	1.0		MCPP Milestone 1
03/02/2007	1.1		Monthly release
04/03/2007	1.2		Monthly release

Date	Revision History	Revision Class	Comments
05/11/2007	1.3		Monthly release
06/01/2007	2.0	Major	Updated and revised the technical content.
07/03/2007	2.0.1	Editorial	Revised and edited the technical content.
07/20/2007	2.0.2	Editorial	Revised and edited the technical content.
08/10/2007	2.1	Minor	Updated the technical content.
09/28/2007	2.2	Minor	Updated 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.
01/25/2008	5.0	Major	Updated and revised the technical content.

Table of Contents

1 Introduction

This document specifies the behavior of the EventLog Remoting Protocol Version 6.0. This version is not available for versions of Windows before Windows Vista.

The EventLog Remoting Protocol Version 6.0 is a Microsoft- proprietary **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**.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Endpoint
Globally Unique Identifier (GUID)
Interface Definition Language (IDL)
Opnum
Remote Procedure Call (RPC)
RPC Protocol Sequence
Universally Unique Identifier (UUID)

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**.

Event: A discrete piece of historical information that may be of interest to administrators of a computer system. 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**.

Live Event Log: An **event log** that can be written to and read from.

Publisher: An application or component that writes to one or more **event logs**.

Query: A context-dependent term commonly overloaded with three meanings, defined as follows:

- The act of requesting **records** from a set of **records**.
- The request itself.
- The particular string defining the criteria for which **records** are to be returned. This string may either be an XPath, as specified in [XPATH], (for more information, see section 2.2.15)

or a **structured XML query**, as specified in [XML10], (for more information, see section 2.2.16).

Record: The physical data structure that contains an **event** that is currently in an **event log**.

Result Set: **Records** selected by a **query**.

Structured XML Query: An XML document that specifies a **query** that may 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.

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

1.2 References

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. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[GSS] Piper, D. and Swander, B., "A GSS-API Authentication Method for IKE", Internet Draft, July 2001, http://www3.ietf.org/proceedings/02mar/I-D/draft-ietf-ipsec-isakmp-gss-auth-07.txt

If you have any trouble finding [GSS], please check here.

[MS-DTYP] Microsoft Corporation, "Windows Data Types", January 2007.

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-GPSI] Microsoft Corporation, "Group Policy: Software Installation Protocol Extension", August 2007.

[MS-KILE] Microsoft Corporation, "Kerberos Protocol Extensions", January 2007.

[MS-NLMP] Microsoft Corporation, "NT LAN Manager (NTLM) Authentication Protocol Specification", June 2007.

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", January 2007.

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

[RFC3986] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", 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.ietf.org/rfc/rfc4234.txt

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

[XML] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Fourth Edition)", W3C Recommendation, September 2006, http://www.w3.org/TR/REC-xml

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

[XMLSCHEMA1.1/2] Peterson, D., Ed., Biron, P.V., Ed., Malhotra, A., Ed., and Sperberg-McQueen, C.M., Ed., "XML Schema 1.1 Part 2: Datatypes", W3C Recommendation, February 2006, http://www.w3.org/TR/xmlschema11-2/

[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

[MS113996] Microsoft Corporation, "INFO: Mapping NT Status Error Codes to Win32 Error Codes", March 2005, http://support.microsoft.com/kb/113996

[MSDN-EVENT] Microsoft Corporation, "Event Logging", http://msdn2.microsoft.com/en-us/library/aa363652.aspx

[MSDN-EVENTS] Microsoft Corporation, "Event Schema", http://msdn2.microsoft.com/en-us/library/aa385201.aspx

[MSDN-FILEATT] Microsoft Corporation, "GetFileAttributes", http://msdn2.microsoft.com/en-us/library/aa364944.aspx

1.3 Protocol Overview (Synopsis)

1.3.1 Background

Event logs allow applications or the operating system to store historical information that may be 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 may 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 may 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.

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 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 events 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 also provides methods for configuring event logs and publishers. Additionally, the protocol provides methods for converting events into localized messages suitable for display to users.

Queries 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. 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.

1.5 Prerequisites/Preconditions

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

1.6 Applicability Statement

The EventLog Remoting Protocol Version 6.0 is well-suited for reading event logs.<1> Event logs can be used for many purposes; for example, recording local security events and application start/stop events.

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 UUID may still exist, and the client may 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 MUST be treated in a case-insensitive manner, MUST be limited to 255 characters, and MUST NOT begin with the character \. No restrictions other than these exist on the characters

that are included in a channel name. However, channel names SHOULD<2> 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 character \. The set of publisher names also includes all event sources (for more information, see [MSDN-EVENTS]). Apart from these restrictions, there are no other character restrictions on publisher names. However, publisher names SHOULD <3> be prefixed with a unique value (such as the name of the entity that created the publisher) so that the channel 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<4> reuse those values with their indicated meanings. Choosing any other value runs the risk of a collision in the future. For a mapping of NT status error codes to Win32 error codes, see [MS113996].

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}
RPCEndpoint Name	Eventlog

2 Messages

The following sections specify how the EventLog Remoting Protocol Version 6.0 messages are encapsulated on the wire, and the common EventLog Remoting Protocol Version 6.0 data types.

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 (0x9) and Kerberos [MS-KILE] (0x10) as the RPC authentication service, as specified in [MS-RPCE].

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 (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 {
  DWORD m_error;
  DWORD m_subErr;
  DWORD 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;
  } RpcVariant;
} EvtRpcVariant;
```

type: Indicates the actual type of the union.

flags: This flag MUST be set to either 0x0000 or 0x0001. If set to 0x0001, this flag 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	Flag indicating that no instance of a **EvtRpcVariant** was changed by the client.
0x0001	Flag indicating that a **EvtRpcVariant** was been changed by the client.

RpcVariant: Data type to be passed.

nullVal: MUST be set to 0x00000000.

booleanVal: **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 stored as a **BooleanArray**.

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

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

stringArray: An array of strings stored as a **StringArray**.

guidArray: An array of GUIDs 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** 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 {
  [string] 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. Text XML 1.0 is specified in [XML10]. Here, BinXml encodes an XML document such that the original XML text can be faithfully reproduced from the encoding. For information on 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 by the following Augmented Backus-Naur Form (ABNF), as specified in [RFC4234]).

In addition to defining the layout of the binary XML binary large objects (BLOBs), the following ABNF 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. For example:

```
Attribute = AttributeToken Name AttributeCharData
 ; Emit as per Element Rule
```

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 here in the text to delineate the characters that are sent on the wire. For example, an instruction might specify that '/>' should be 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:	Major version of BinXml. MUST be set to 1.
MinorVersion:	Minor version of BinXml. MUST be set to 1.
Flags:	Reserved value in 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 after ElementByteLength making up the entire element definition, and including EndElementToken/CloseEmptyElementToken for the element.
AttributeListByteLength:	The number of bytes in the attribute list after AttributeListByteLength up to but not including the CloseStartElementToken or CloseEmptyElementToken, typically used for jumping to the end of the enclosing start element tag.
AttributeCharData:	Character data appearing within an attribute value.
SubstitutionId:	A zero-based positional identifier into the set of substitution values. 0 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 generated by performing an MD5 hash of the binary representation of Name (in which NameNumChars * 2 is the hash input length).
NameNumChars:	The number of Unicode characters for the NameData, not including the null terminator.

Entity	Description
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 within the element start tag.
ValueTextToken:	A value of 0x45 indicates that more data can be expected to follow in the current content of the element/attribute; a value of 0x05 indicates that no more such data follows.
AttributeToken:	A value of 0x46 indicates that there will be 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/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/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/attribute; a value of 0x09 indicates that no more such data follows.
TemplateId:	The raw data of the GUID identifying a template definition.
NumValues:	The number of substitution values making up the Template Instance Data.
ValueByteLength:	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 for the template definition.
ValueType:	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, but 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 substition, 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 character ' ' and the text, as specified by

the Name rule in 2.2.12, character '=', 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

Type	Output format
	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 '-', two digit month '-', two digit day 'T', two digit hour ':', two digit seconds '.' and 3 digit milliseconds 'Z'. For example, 2006-10-20T03:23:54.248Z is 3:23:34 am of October 20, 2006.
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 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]. It must also conform to the following XML schema, as specified in [XMLSCHEMA1.1/2].

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. But client applications do not need to interpret these values to call these functions. For information on event descriptors, see sections 1.8.3 and 2.2.18.

```
<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]:

```
<?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"/>
</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.
- **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]):

```
<?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	List of 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.

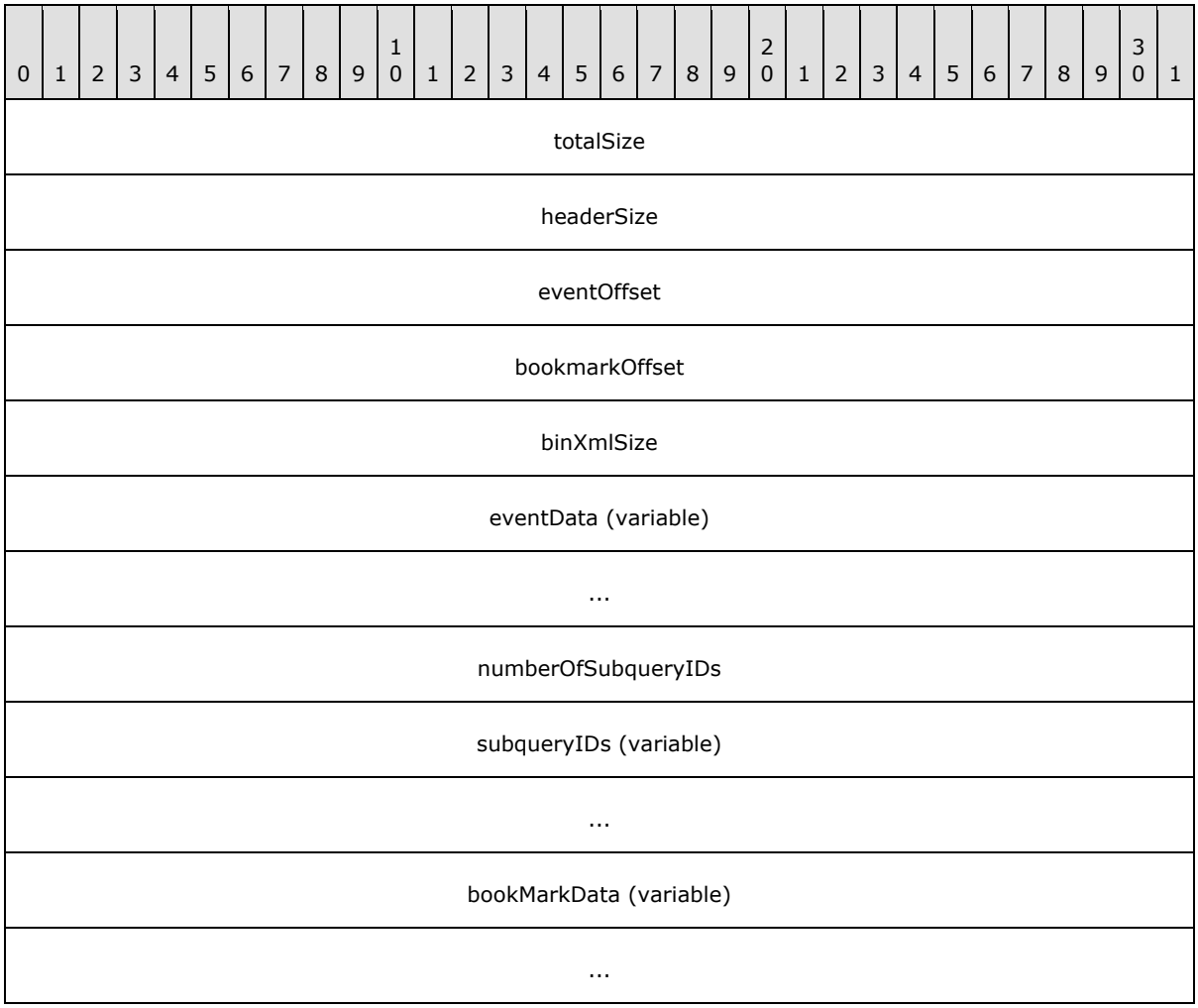
Attributes	Description
ID	Defines the ID of a subquery so that a consumer can determine what subquery out of many caused the record to be included in a result set. Multiple subqueries using the same IDs are not distinguished in the result set. For information on subquery IDs, see section 2.2.17.
Path	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

Attributes	Description
	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. 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.

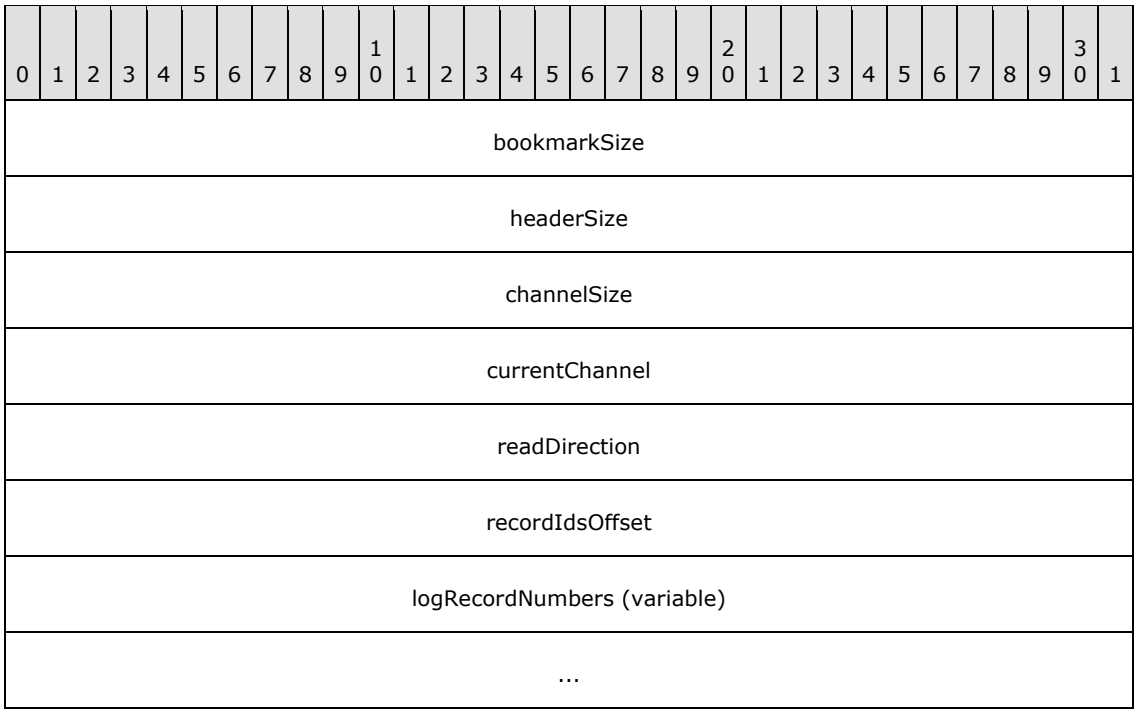
eventData (variable): A byte-array that contains variable length BinXml data.

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).



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**.

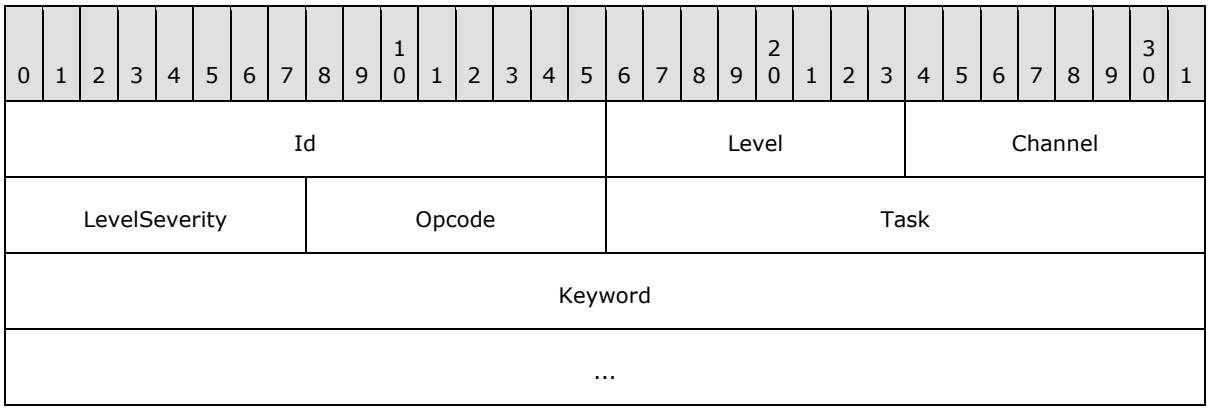
logRecordNumbers (variable): An array of 64-bit unsigned integers that contain the record numbers for each of the channels or backup event logs. The order of the records numbers MUST match the order of the channels or backup event logs in the query (for example, the first channel in the query corresponds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

2.2.18 Event Descriptor Structure

Each event has an EVENT_DESCRIPTOR that identifies what kind of an event it is. Each publisher has its own set of EVENT_DESCRIPTOR values, and they may overlap with the values in other publishers because it is the combination of publisher and EVENT_DESCRIPTOR that needs to be unique.

Client applications often use this structure when rendering BinXml into text. To use some of the rendering methods, this structure must be passed to the server so that it can locate the proper localized strings. The individual values are embedded in the BinXml. The location of the values is as specified by the Event XML schema, as specified in section 2.2.13.

The Event Descriptor Structure is custom-marshaled. All integer fields in this structure MUST be in little-endian byte order (that is, least significant byte first).



Id (2 bytes): A 16-bit unsigned integer that contains the event identifier.

Level (1 byte): An 8-bit unsigned integer that contains the version of the event. The version indicates a revision to the event definition.

Channel (1 byte): This can be set to an arbitrary value by the client, and MUST be ignored by the server on receipt.

LevelSeverity (1 byte): An 8-bit unsigned integer that contains the severity of the event.

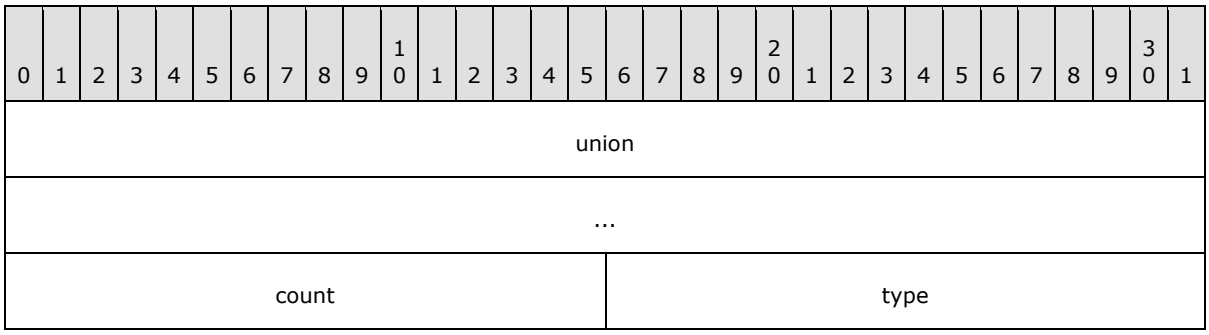
Opcode (1 byte): An 8-bit unsigned integer that indicates the operation being performed at the time the event is written.

Task (2 bytes): A 16-bit unsigned integer that identifies a logical component of the application that is writing the event.

Keyword (8 bytes): A 64-bit unsigned integer that contains the mask that is specified when the event is written. The keyword can contain one or more provider-defined keywords, standard keywords, or both.

2.2.19 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).



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

count (2 bytes): Not used. Can be set to an arbitrary value, and MUST be ignored on receipt.

type (2 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.

2.2.20 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;
```

2.2.21 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.

2.3 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_CHANNEL_COUNT	1024
MAX_RPC_BOOKMARK_LENGTH	(MAX_PAYLOAD / sizeof(**WCHAR**))
MAX_RPC_PUBLISHER_ID_LENGTH	2048
MAX_RPC_PROPERTY_BUFFER_SIZE	MAX_PAYLOAD

Name	Value
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**))
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

The following sections specify details of the EventLog Remoting Protocol Version 6.0, including abstract data models and message processing events and sequencing rules.

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 may also contain event specific data.

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.

The publisher may be registered within the system and have an identifier. The server keeps a table of registered publishers, and each is associated with specific channels.

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

Also, publishers may 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.

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 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.

Channels are registered in the system and have an identifier. The server keeps a table of registered channels.

Channel identifiers can be obtained through the protocol 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. These properties may be observed or modified through this protocol.

3.1.1.4 Logs

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

A log may also exist as an external file. In this case, the log is identified by using the file system path of that log file.

The events contained within log files, that are associated with channels, can be exported into an external log file by using this protocol. Also through this protocol, information about the log files can be obtained, and the log files themselves can be managed.

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

3.1.1.5 Queries

Events within log files can be queried through the protocol. 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 server MUST maintain a table of queries. For each query, the server MUST keep track of a cursor in the result set, as well as in the filters associated with the query.

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

3.1.1.6 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.

The server MUST keep a table of ongoing subscriptions. For each subscription, the server MUST keep track of what events have been delivered as well as the filters associated with the subscription.

3.1.1.7 Handles

The server MUST maintain a single table of handles associating client connections to internal server states related to the connection. An example of such a state is the current cursor position in the result set of a query.

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

- CONTEXT_HANDLE_REMOTE_SUBSCRIPTION: The state includes the filters being used, and keeps track of what has been delivered.
- CONTEXT_HANDLE_LOG_QUERY: The state includes the filter as well as the cursor in the result set.
- CONTEXT_HANDLE_LOG_HANDLE: The state includes an actual system handle to an open file.
- CONTEXT_HANDLE_PUBLISHER_METADATA: The state includes the identity of the publisher as well as the locale to be used.
- CONTEXT_HANDLE_EVENT_METADATA_ENUM: The state includes the identity of the publisher as well as the position of the enumeration.
- CONTEXT_HANDLE_OPERATION_CONTROL: The state includes the method calls that can be controlled by the handle.

3.1.1.8 Localized String Table

The server MUST have a table of localized strings for each publisher and a default table.

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

Method	Description
	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
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 should be removed.

Method	Description
	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
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 NT Status Error Codes to Win32 Error Codes, see [MS113996]. All error values MUST<5> be treated the same, unless specified otherwise.

Within the sections that follow, 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 such that the original XML text can be faithfully 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, rather than as an encoding of XML. However, it is common for third-party applications to convert from binary XML to text XML independent of the protocol, after the data have been received. 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 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

Where the binary bytes 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 that may be 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 (set of data) values may remain in native form, and only needs to be converted to text when actually rendering the BinXml encoding into XML text (if ever).

A BinXml encoding of an XML fragment by using templates along with 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 the main BinXml topic (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 zero-based whereas the substitution tokens are one-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
<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 EOF token (0x00) for the Template Instance. Immediately following the Template Definition is information on 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 above 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 type of the substitution value is NULL (see Type System below), and this is how the processor of the BinXml 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 (non null) 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]), and an EVT: prefix specifies types defined in the event.xsd (as specified in section 2.2.13).

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
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

BinXml type	Meaning	Canonical XML representation
BinXmlTyp e	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:

```
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

Text	0C
61 63 65 00	

And the resultant XML text representation of this encoding is:

```
<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.

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.

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, 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
);
```

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
EvtSubscribeLoose 0x00001000	Succeed even if not all channels are valid.
EvtSubscribeStrict 0x00010000	Fail if any events are missed for reasons such as log clearing.
EvtSubscribePull 0x10000000	Subscription is going to be a pull subscription.

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. The server MUST set the name element to the name of the channel 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.

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 non-zero value.

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 *channelPath* argument specifies a channel that does not exist.
- The *query* parameter is syntactically incorrect. 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**) in the *flags* parameter, the server MUST fail the method.

  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.
- 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 MAY return this same error code if bookmarkXML is otherwise valid, but indicates an event that does not exist in the log.
- The server SHOULD fail the method if both the path and query parameters are NULL<7>.

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.
- Add the handles to its internal tables.

If any of the checks above fail, the server MUST NOT create the context handles or add them to the table.

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 0 and ignored on receipt.

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 an array of 32-bit unsigned integers that contain 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 on success; otherwise, it MUST return an implementation-specific non-zero value.

Note The value of ERROR_SUCCESS is 0x00000000.

In response to this request from the client, the server MUST first validate the handle. The server SHOULD fail the operation if it has no state for the handle or if the handle is not valid<8>.

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 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 internal state to keep track of what events have been delivered to the subscription corresponding to the table.

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 is 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 0 and ignored on receipt.

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 an array of 32-bit unsigned integers that contain 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.

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

- Validate the handle. The server MUST fail the operation if it has no state for the handle.
- 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.
- 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.
  - 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 MUST update its internal state to keep track of the events received by the client so that subsequent calls can retrieve the rest of the result set. That is, 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 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 non-zero value.

In response to this request from the client, the server MUST first validate the handle. The server SHOULD fail the operation if it has no state for the handle<9>.

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. If there are no events meeting that criteria, the server MUST NOT complete this operation.

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, 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
);
```

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 that the query result set should be returned even if there are errors resulting from the query. For example, consider the case of a structured XML query that specifies multiple channels, there may be cases in which some channels are valid while others are not. A query 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 not all channels are 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 non-zero 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 non-zero values MUST be treated the same. If the method succeeds, the server MUST set all of 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 non-zero value.

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 MUST interpret the *path* to be either a channel name or file path name, depending on the *flags* parameter.

The server SHOULD validate the flags values, and return an error if any of the following conditions occur:<10>

- Neither EvtQueryChannelPath or EvtQueryFilePath is set.
- Both EvtQueryChannelPath and EvtQueryFilePath are set.
- Neither 0x00000100 or 0x00000200 is set.
- Both 0x00000100 and 0x00000200 are set.

- Any flag not specifically defined above is set.
- At least one of the query and path parameters are non-NULL.

The server MUST fail the method if the *query* argument is syntactically incorrect. 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; hence 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).

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 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; STATUS_ACCESS_DENIED (0xC0000022) if the channel exists but the user does not have read access to the channel; or some other relevant error in case of other errors)<11>. 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.

Next, the server MUST verify that the caller has read access to the file, and MUST fail the method if the caller does not have read access.

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 a **CONTEXT_HANDLE_OPERATION_CONTROL** and return it in the *opControl* parameter. If successful, the server MUST update 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 table.

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.

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 0 and be ignored on receipt.

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.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

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*. The server MUST continue the attempt 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 internal state to keep track of the query's location in the result set.

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 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: Number of records in the result set to move by. If the number is positive, then 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 and 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)** should start its seek. The origin flags (the first four flags that follow) are mutually exclusive, while the last flag can be set independently. The *pos* parameter specifies the offset used in the definitions of these flags.

Value	Meaning
EvtSeekRelativeToFirst 0x00000001	Offset is relative to the first entry in the result set, and SHOULD be nonnegative. So, if an offset of 0 is specified, the cursor is to be moved to the first entry in the result set.
EvtSeekRelativeToLast 0x00000002	Offset is relative to the last entry in the result set, and SHOULD be nonpositive. So, if an offset of 0 is specified, the cursor is to be moved to the last entry in the result set.
EvtSeekRelativeToCurrent 0x00000003	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	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. Note that the presence of the EvtSeekStrict flag MAY influence the behavior of this flag as specified below.
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 remaining in the result set and the *pos* parameter specifies 100 with the **EvtSeekRelativeToCurrent** flag

Value	Meaning
	set, the 99th record will be 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.

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 on success; otherwise, it MUST return an implementation-specific non-zero value.

Note The value of ERROR_SUCCESS is 0x00000000.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

The server SHOULD validate that the sign of the *pos* parameter makes sense with respect to the search direction. That is, the server SHOULD return an error if a negative *pos* value is specified along with the EvtSeekRelativeToFirst flag<12>.

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 follow:

- EvtSeekRelativeToFirst
- EvtSeekRelativeToLast
- EvtSeekRelativeToCurrent
- EvtSeekRelativeToBookmark

If the above check succeeds, the server MUST attempt to move the cursor within the result set. The server MUST interpret the movement in terms of the result set rather than just raw events in the log. Note that because there is a filter, the result set can be smaller than the events in the logs. The server MUST update its internal state based on the movement within the result set.

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 an 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.

propertyId: A 32-bit unsigned integer that indicates what file property needs to be retrieved.

Value	Meaning
EvtLogCreationTime 0x00000000	A **FILETIME** containing the creation time of the file.
EvtLogLastAccessTime 0x00000001	A **FILETIME** containing the last access time of the file.
EvtLogLastWriteTime 0x00000002	A **FILETIME** containing the last write time of the file.
EvtLogFileSize 0x00000003	An unsigned 64-bit integer containing the size of the file.
EvtLogAttributes 0x00000004	An unsigned 32-bit integer containing the attributes of the file. The attributes are implementation specific, and clients MUST<13> treat all values equally.
EvtLogNumberOfLogRecords 0x00000005	An unsigned 64-bit integer containing the number of records in the file.
EvtLogOldestRecordNumber 0x00000006	An unsigned 64-bit integer containing the oldest record number in the file.
EvtLogFull 0x00000007	A **BOOLEAN** value; MUST be true if the log is full, and MUST be false otherwise.

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 non-zero value.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

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 above checks succeed, the server MUST attempt to return the request information. The server MUST pack the return data into a single BinXmlVariant structure, as specified in section 2.2.19. 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 a live event log, 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 0 and ignored on receipt.

error: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific non-zero value.

Return Values: The method returns 0 (ERROR_SUCCESS) on success; otherwise, it returns a non-zero error code.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

The server MUST verify that the **channelPath** parameter specifies a correct channel name. The server MUST fail the call if the **channelPath** parameter is invalid.

If the *backupPath* is non-NULL and non-empty, the server MUST validate the path and fail the call if it is not a file path or if it specifies a file that already exists. If the *backupPath* is valid, the server MUST attempt to back up the log to the path specified in *backupPath* before the log is cleared. The method MUST fail and not clear the log if the backup does not succeed.

If the *backupPath* is NULL or empty, the method MUST NOT attempt to back up the event log.

If the above 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.

If all events are successfully deleted ("cleared") then the server MUST return 0 (ERROR_SUCCESS) indicating success. Otherwise, it MUST return an implementation-specific nonzero value.

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 channels or backup event logs specified in the query are present.

The server MAY ignore unrecognized flag combinations<14>.

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 non-zero value.

In response to this request from the client, the server MUST verify that the *channelPath* parameter specifies either a correct channel name (in the case in which the flags parameter is 0x00000001) or

a valid path to a backup event log. The server MUST fail the call if the *channelPath* parameter is invalid.

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 an error if the flag validation fails<15>.

The server MUST verify that the *query* parameter is a valid XPath expression, and fail the operation if it is not. For information on XPath filters supported by this protocol, see section 2.2.15.

The server MUST verify that *backupPath* is a valid path, and fail the method if it is not valid or if it specifies a file that already exists.

Next, the server MUST verify that the caller has read access to the file and MUST fail the method if the caller does not have read access.

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. There is no server state that needs to be updated by this method.

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

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. 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 view the results with localized strings.

```
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 0 and ignored on receipt.

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 non-zero 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 non-zero value.

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. The server MUST fail the operation if the *logFilePath* parameter is invalid.

Next, the server MUST verify that the caller has read access to the file and MUST fail the method if the caller does not have read access.

If the checks above are successful, the server MUST attempt to attach localized information to a file. The information MUST be sufficient so that the events can be rendered via the **EvtRpcMessageRender (section 3.1.4.31)** method, even if the file is moved to another computer. There is no server state that needs to be updated by this method.

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

3.1.4.19 EvtRpcOpenLogHandle (Opnum 17)

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

```
error_status_t EvtRpcOpenLogHandle(
  [in, range(1, MAX_RPC_CHANNEL_PATH_LENGTH), string]
    LPCWSTR channel,
  [in] DWORD flags,
  [out, context_handle] PCONTEXT_HANDLE_LOG_HANDLE* handle,
  [out] RpcInfo* error
);
```

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. 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.

In response to this request from the client, the server MUST first validate the *channel* parameter. 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 backup event log. The server MUST fail the operation if the validation of the channel parameter fails.

Next the server MUST verify that the caller has read access to the file and MUST fail the method if the caller does not have read access.

If the above checks succeed, the server MUST attempt to create a CONTEXT_HANDLE_LOG_HANDLE. If successful, the server MUST update its handle table to track the issued handle. If any of the checks above fail, the server MUST NOT create the context handle or add it to the table. 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] DWORD flags,
  [out] DWORD* numChannelPaths,
  [out, size_is(,*numChannelPaths), range(0, MAX_RPC_CHANNEL_COUNT), string]
    LPWSTR** channelPaths
);
```

flags: A 32-bit unsigned integer that MUST be set to 0 and ignored on receipt.

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.

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 if the caller does not have access.

If the above check succeeds, the server MUST return the list of channels. 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, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in] DWORD flags,
  [out] EvtRpcVariantList* props
);
```

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 0 and ignored on receipt.

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

<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>EvtRpcVarTypeBoolean
0</td><td>Enabled. If true, the channel can accept new events.</td></tr>
<tr><td>EvtRpcVarTypeUInt32
1</td><td>Channel Isolation. One of three values:
<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Application. Use security settings of Application channel.</td></tr><tr><td>1</td><td>System. Use security settings of System channel.</td></tr><tr><td>2</td><td>Custom. The channel has its own explicit security settings.</td></tr></table></td></tr>
<tr><td>EvtRpcVarTypeUInt32
2</td><td>Channel type. One of four values:
<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>Admin</td></tr><tr><td>1</td><td>Operational</td></tr><tr><td>2</td><td>Analytic</td></tr><tr><td>3</td><td>Debug</td></tr></table></td></tr>
<tr><td>EvtRpcVarTypeString
3</td><td>OwningPublisher. Name of the publisher that defines and registers the channel with the system.</td></tr>
<tr><td>EvtRpcVarTypeBoolean
4</td><td>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).</td></tr>
<tr><td>EvtRpcVarTypeString
5</td><td>Access. A Security Descriptor Description Language (SDDL) string, as specified in [MS-DTYP], that represents access permissions to the channels.</td></tr>
<tr><td>EvtRpcVarTypeBoolean
6</td><td>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.</td></tr>
<tr><td>EvtRpcVarTypeBoolean
7</td><td>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.</td></tr>
<tr><td>EvtRpcVarTypeUInt64
8</td><td>MaxSize. The value that indicates at which point the size (in bytes) of the event log file should stop increasing. When the size is greater than or equal to this value, the file growth stops.</td></tr>
<tr><td>EvtRpcVarTypeString
9</td><td>LogFilePath. File path to the event log file for the channel.</td></tr>
<tr><td>EvtRpcVarTypeUInt32</td><td>Level. Events with a level property less than or equal to this specified</td></tr>
</table>

<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>10</td><td>value are logged to the channel.</td></tr>
<tr><td>EvtRpcVarTypeUInt64
11</td><td>Keywords. Events with a keyword bit contained in the Keywords bitmask set are logged to the channel.</td></tr>
<tr><td>EvtRpcVarTypeGuid
12</td><td>ControlGuid. A GUID that can be used to identify a publisher writing events to the channel.</td></tr>
<tr><td>EvtRpcVarTypeUInt64
13</td><td>BufferSize. Size of the events buffer (in kilobytes) used for asynchronous event delivery.</td></tr>
<tr><td>EvtRpcVarTypeUInt32
14</td><td>MinBuffers. The minimum number of buffers used for asynchronous event delivery.</td></tr>
<tr><td>EvtRpcVarTypeUInt32
15</td><td>MaxBuffers. The maximum number of buffers used for asynchronous event delivery.</td></tr>
<tr><td>EvtRpcVarTypeUInt32
16</td><td>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.</td></tr>
<tr><td>EvtRpcVarTypeUInt32
17</td><td>ClockType. One of two values:<table><tr><th>Value</th><th>Description</th></tr><tr><td>0</td><td>SystemTime. Use the system time (which is low-resolution on most platforms) for an event time stamp.</td></tr><tr><td>1</td><td>Query Performance Counter. Use a high-resolution time stamp for an event time stamp.</td></tr></table></td></tr>
<tr><td>EvtRpcVarTypeUInt32
18</td><td>SIDType. One of two values:<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0</td><td>The publisher's SID is not written to events automatically.</td></tr><tr><td>1</td><td>The publisher's SID is written to events automatically.</td></tr></table></td></tr>
<tr><td>EvtRpcVarTypeStringArray
19</td><td>PublisherList. List of publishers that can raise events into the channel.</td></tr>
</table>

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific non-zero value.

In response to this request from the client, the server MUST verify that the *channelPath* parameter specifies a valid channel name and MUST fail the method if the parameter is invalid.

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.

If the previous checks succeed, the server MUST attempt to return the list of a channel's properties. The server MUST return the properties for the channel into an **EvtRpcVariant** list with the elements of props, listed in the previous table. 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].

The server MUST NOT update its state as a result of this method.

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, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR channelPath,
  [in] DWORD flags,
  [in] EvtRpcVariantList* props,
  [out] RpcInfo* error
);
```

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.

Value	Meaning
0x00000000	Overwrite existing log file or create new log file.
0x00000001	Only update existing log file.
0x00000002	Always create a new log file. Delete any existing log file.
0x00000003	Only create a new log file. Fail if it already exists.

props: A pointer to an **EvtRpcVariantList (section 2.2.9)** structure containing channel properties.

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.

In response to this request from the client, the server SHOULD verify that the *channelPath* parameter specifies a correct channel name. The server SHOULD fail the method if the *channelPath* parameter is invalid<16>.

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<17>.

If the previous checks succeed, the server MUST attempt to update the channel's properties. The server MUST change its storage of the properties, but not apply the properties until either the server restarts or **EvtRpcAssertConfig** is called.

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] DWORD flags,
  [out] DWORD* numPublisherIds,
  [out, size is(,*numPublisherIds), range(0, MAX RPC PUBLISHER COUNT), string]
    LPCWSTR** publisherIds
);
```

flags: A 32-bit unsigned integer that MUST be set to 0 and ignored on receipt.

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 non-zero value.

In response to this request from the client, 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.

If the above check succeeds, the server MUST attempt to return the list of publishers using its list of publishers (for more information, see section 3.1.1.2).

The server MUST fill in the parameters as specified.

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, string] LPCWSTR channelName,
  [in] DWORD flags,
  [out] DWORD* numPublisherIds,
  [out, size is(,*numPublisherIds), range(0, MAX RPC PUBLISHER COUNT), string]
    LPCWSTR** publisherIds
);
```

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 0 and ignored on receipt.

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.

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.

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.

If the previous checks succeed, the server MUST attempt to return a list of publishers for the channel specified by the *channelName* parameter. 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, 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
);
```

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] Appendix A. This is used later if the *pubMetadata* handle is used for rendering.

flags: A 32-bit unsigned integer that MUST be set to 0 and ignored on receipt.

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 2.2.21 and 5.1.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value.

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 nonNULL and is not the name of a publisher. If the *publisherID* parameter is NULL, the server MUST use the default Localized String Table (section 3.1.1.8).

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.

If the previous checks succeed, the server MUST attempt to return the publisher data via the *pubMetadataProps* and create a CONTEXT_HANDLE_PUBLISHER_METADATA for the publisher. The server MUST store the *locale* value as part of the handle's state. The server MUST update its handle table.

The server MUST return an **EvtRpcVariantList** (for more information, see section 2.2.9) that contains 29 **EvtRpcVariants**. 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
1	EvtVarTypeString	ResourceFilePath
2	EvtVarTypeString	ParameterFilePath
3	EvtVarTypeString	MessageFilePath
7	EvtVarTypeStringArray	ChannelReferencePath
8	EvtVarTypeUInt32Array	ChannelReferenceIndex
9	EvtVarTypeUInt32Array	ChannelReferenceID
10	EvtVarTypeUInt32Array	ChannelReferenceFlags
11	EvtVarTypeUInt32Array	ChannelReferenceMessageID

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

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 0 and ignored on receipt.

pubMetadataProps: Pointer to an **EvtRpcVariantList (section 2.2.9)** structure. This list MUST contain multiple entries.

Return Values: The method MUST return ERROR_SUCCESS on success; otherwise, it MUST return an implementation-specific non-zero value.

Note The value of ERROR_SUCCESS is 0x00000000.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

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 MUST fail.

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.

When *propertyID* = 0x00000004, the following entries MUST be set in *pubMetadataProps*.

Index	Type	Description
4	EvtVarTypeString	HelpLink

When *propertyID* = 0x00000005, the following entries MUST be set in *pubMetadataProps*.

Index	Type	Description
05	EvtVarTypeUInt32	PublisherMessageID

When *propertyID* = 0x0000000C, the following entries MUST be set in *pubMetadataProps*.

Index	Type	Description
13	EvtVarTypeStringArray	LevelName
14	EvtVarTypeUInt32Array	LevelValue
15	EvtVarTypeUInt32Array	LevelMessageID

When *propertyID* = 0x00000010, the following entries MUST be set in *pubMetadataProps*.

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*.

Index	Type	Description
22	EvtVarTypeStringArray	OpcodeName
23	EvtVarTypeUInt32Array	OpcodeValue
24	EvtVarTypeUInt32Array	OpcodeMessageID

When *propertyID* = 0x00000019, the following entries MUST be set in *pubMetadataProps*.

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 2.2.21 and 5.1.

flags: A 32-bit unsigned integer that MUST be set to 0 and ignored on receipt.

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.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

If the previous check succeeds, the server MUST attempt to create a CONTEXT_HANDLE_EVENT_METADATA_ENUM handle. The handle must be initialized so that it can be used to enumerate the publisher's events. The server MUST update its handle table to keep track of the new handle. If the previous check fails, the server MUST NOT create the context handle or add it to the table.

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 2.2.21 and 5.1.

flags: A 32-bit unsigned integer that MUST be set to 0x00000000 and ignored on receipt.

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.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

The server MUST verify that event metadata entries remain in the publisher metadata that have not yet been enumerated. If the enumeration has already returned the metadata for every event, the method SHOULD fail<18>. 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 0.<19>

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 return an array of **EvtRpcVariantList** (section 2.2.9) objects, with an **EvtRpcVariantList** for each event's metadata. Each **EvtRpcVariantList** MUST contain the following nine EvtVariant entries.

Index	Type	Description
0	EvtVarTypeUInt32	ID
1	EvtVarTypeUInt32	Version
2	EvtVarTypeUInt32	Channel index
3	EvtVarTypeUInt32	Level
4	EvtVarTypeUInt32	Opcode
5	EvtVarTypeUInt32	Task
6	EvtVarTypeUInt32	Keywords
7	EvtVarTypeUInt64	MessageID
8	EvtVarTypeString	Template

If the preceding checks succeed and the server successfully creates the array of **EvtRpcVarialList** objects, the server MUST update the handle state 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 the handle state.

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, range(1, MAX_RPC_CHANNEL_PATH_LENGTH), string]
    LPCWSTR path,
  [in] DWORD flags
);
```

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.
EvtRpcChannelName 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.

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. In that case, it must validate that the *path* specifies a known channel. Otherwise, it must interpret *path* as a publisher name, and validate that the *path* specifies a known publisher. The server SHOULD fail the operation if the validation of *path* fails<20>.

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<21>.

If the above checks succeed, the server MUST attempt to update its state for the channel or publisher specified. This method MUST support adding or changing information.

Note that this protocol does not include a method for changing publisher data. The client MAY provide this functionality.<22>

The server MUST determine the state of the channels and publishers at startup. The states MAY 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-dependant configurable settings.

The server MUST NOT change its operating state until either this method is called or until the server restarts.

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 configuration should be removed.

```
error_status_t EvtRpcRetractConfig(
  [in, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR path,
  [in] DWORD flags
);
```

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.
EvtRpcChannelName 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.

In response to this request from the client, the server SHOULD first validate the *path* parameter<23>. The server MUST interpret the *path* parameter as a channel name if the flags parameter is equal to 0x00000000; otherwise, the server must interpret *path* as a publisher name. The server SHOULD fail the operation if the validation of *path* fails.

The server MUST verify that the caller has delete access to the information and MUST fail the method if the caller does not have delete access<24>.

If the above checks succeed, the server MUST attempt to update its state for the channel or publisher specified.

The server determines the state of the channels and publishers at startup. The server MUST NOT change its operating state until either this method is called or until the server restarts.

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.

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.

eventId: Pointer to an **EVENT_DESCRIPTOR structure**, as specified in section 2.2.18. The server MUST ignore this value if EvtFormatMessageId is specified as the flags parameter.

messageId: A 32-bit unsigned integer that specifies the required message. This is an alternative to using the *eventID* parameter that may be used by a client application that has obtained the value through some method outside of 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.<25>

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 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.
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 of the string in bytes 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 size needed in bytes (including null termination).

string: A pointer to a byte-array 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 non-zero values MUST be treated the same. If the method succeeds, the server MUST set all of 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 non-zero value.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle.

If the check above is successful, 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 server MUST define values for the following:

- levels 0 through 15
- task 0
- opcodes 0 through 9, and 240
- the following keywords
  - 0
  - 0x1000000000000
  - 0x2000000000000
  - 0x4000000000000
  - 0x8000000000000
  - 0x10000000000000
  - 0x20000000000000
  - 0x40000000000000
  - 0x80000000000000.

for levels 0 through 15. If the level requested is in that range, the server's list of strings MUST be used. The server MUST NOT change any state.

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, 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
);
```

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 section 2.2.18.

messageId: A 32-bit unsigned integer that contains the messageId for the case in which *eventID* is NULL, and, therefore, there is no *eventID* from which to determine the messageId. If *eventID* is non-NULL, messageId MUST be sent as 0xFFFFFFFF, and the server MUST ignore this 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 0.

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.

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, whereas the **EvtRpcMessageRender** (section 3.1.4.31) method only uses the default strings in the case of level, task, opcode, and keyword values that fall in certain ranges.

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.

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 not in its handle table. For more information on handle security and authentication considerations, see sections 2.2.21 and 5.1.

If the above check succeeds, the server MUST attempt to update its internal state so that the handle is removed from the handle table.

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.

In response to this request from the client, the server MUST first validate the handle. The server MUST fail the operation if it has no state for the handle. For information on handle security and authentication considerations, see sections 2.2.21 and 5.1.

If the above check succeeds, the server MUST attempt to cancel the outstanding call associated with this handle. For information, see section 3.1.4.

Depending on race conditions, the canceled call MAY return one of the following error codes:

- ERROR_CANCELLED (0x000004C7).
- RPC_S_CALL_CANCELLED (0x0000071A).
- An implementation-dependent error code.

In response to this call, the server MUST NOT remove the associated handle from its handle table.

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 2.2.21 and 5.1.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value.

In response to this request from the client, for a successful operation, the server MUST attempt to create a CONTEXT_HANDLE_OPERATION_CONTROL handle. If it cannot create the handle, the

server MUST fail the operation; otherwise, the server MUST update its internal tables to track the issued handle.

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, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string]
    LPCWSTR logName,
  [in] LCID locale,
  [in] DWORD flags,
  [out, string] LPWSTR* displayName
);
```

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: Only value is 0x100.

Value	Meaning
0x100	If set, instructs the server to pick the best locale, if the locale specified by the *locale* parameter is not present.

displayName: Returned display name.

Return Values: The method MUST return ERROR_SUCCESS on success; otherwise, it MUST return an implementation-specific non-zero value.

Note The value of ERROR_SUCCESS is 0x00000000.

In response to this request from the client, for a successful operation, the server MUST attempt to retrieve a display name for a channel. The server verifies that the channel name, as specified by the *logName* parameter, is a known channel. If the *logName* parameter does not specify a known channel, the server MUST fail the method.

If the display name is not present in the specified *locale*, the server makes a best effort attempt as long as the 0x100 bit is set in the flags field. For example, if the user requests U.S. English ("en-US") and that dialect of English is not available on the server but British English ("en-GB") is, the server MAY return the British locale version of the channel name.

The server SHOULD validate the flags to ensure that no flags are present other than 0x100<26>.

Otherwise, the server must fail the method.<27>

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.

4 Protocol Examples

The following sections describe several operations used in common scenarios to illustrate the function of the EventLog Remoting Protocol Version 6.0.

4.1 Query Sample

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.

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, 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 a log query result enumerator for that channel, and this enumerator only returns the events specified by the query expression. In this case, the "*" filter selects all events in the channel.

   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 may be used to cancel a currently executing control handle.

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.

   The client enumerates through the events, using multiple calls to the **EvtRpcQueryNext** (section 3.1.4.13) method, until it is no longer interested in 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** may 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.

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 Bookmark Sample

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>
```

4.3 Simple BinXml Sample

The following is a simple 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.

Token offset	Token type	Comments on encoding
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>.
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 '&' 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.

4.4 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>
```

4.5 BinXml Sample 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, note that the outer template substitutions are all optional, and that some values of that template are NULL, so some of the BinXml elements/attributes are not present in the XML text below.

```
<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 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 - FragmentHeaderToken	Version1.1, Flags = 0. We are at "document" level here and we can expect 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

Token offset	Token type	Comments on encoding
		after the value spec entries (at offset 0x55D). Offset 0x55D + 0x1C6 bytes leaves 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 indeed BinXmlType. The index into the value spec is 0x13.
0x506	0x04 - EndElementToken	End <UserData>.
0x507	0x04 - EndElementToken	End <Event>.
0x508	0x00 - EOFToken	EOF for the outer template definition.
0x5A0	0x0F - FragmentHeaderToken	This is actually the last value specified in the outer TemplateInstance, but because this value is itself BinXml, it starts with an (optional) header token and will end with an

Token offset	Token type	Comments on encoding
		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 EOF Token for the inner fragment containing the TemplateInstance.
0x722	0x00 - EOFToken	EOF for the inner TemplateInstance.
0x723	0x00 - EOFToken	EOF for the outer TemplateInstance.

5 Security

The following sections specify security considerations for implementers of the EventLog Remoting Protocol Version 6.0.

5.1 Security Considerations for Implementers

Implementers MUST take care 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 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.

```
import "ms-dtyp.idl";

#define MAX PAYLOAD                    (2 * 1024 * 1024)
#define MAX RPC QUERY LENGTH           (MAX PAYLOAD / sizeof(WCHAR))
#define MAX RPC CHANNEL NAME LENGTH    512
#define MAX_RPC_QUERY_CHANNEL_SIZE     512
#define MAX_RPC_EVENT_ID_SIZE          256
#define MAX RPC FILE PATH LENGTH       32768
#define MAX RPC CHANNEL PATH LENGTH    32768
#define MAX RPC BOOKMARK CHANNEL COUNT 1024
#define MAX RPC BOOKMARK LENGTH        (MAX PAYLOAD / sizeof(WCHAR))
#define MAX_RPC_PUBLISHER_ID_LENGTH    2048
#define MAX_RPC_PROPERTY_BUFFER_SIZE   MAX_PAYLOAD
#define MAX RPC FILTER LENGTH          MAX RPC QUERY LENGTH
#define MAX RPC RECORD COUNT           1024
#define MAX RPC EVENT SIZE             MAX PAYLOAD
#define MAX RPC BATCH SIZE             MAX PAYLOAD
#define MAX_RPC_RENDERED_STRING_SIZE   MAX_PAYLOAD
#define MAX_RPC_CHANNEL_COUNT          8192
#define MAX_RPC_PUBLISHER_COUNT        8192
#define MAX RPC EVENT METADATA COUNT   256
#define MAX RPC VARIANT LIST COUNT     256
#define MAX RPC BOOL ARRAY COUNT       (MAX PAYLOAD / sizeof(BOOL))
#define MAX RPC UINT32 ARRAY COUNT     (MAX PAYLOAD / sizeof(UINT32))
#define MAX_RPC_UINT64_ARRAY_COUNT     (MAX_PAYLOAD / sizeof(UINT64))
#define MAX_RPC_STRING_ARRAY_COUNT     (MAX_PAYLOAD / 512)
#define MAX RPC GUID ARRAY COUNT       (MAX PAYLOAD / sizeof(GUID))
#define 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)] 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
{
    [string] LPWSTR name;
    DWORD status;
} EvtRpcQueryChannelInfo;


error status t EvtRpcRegisterRemoteSubscription(
    [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, 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, 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, range(1, MAX_RPC_CHANNEL_NAME_LENGTH), string] LPCWSTR
        path,
    [in] DWORD flags );

error status t EvtRpcRetractConfig(
    [in, range(1, MAX RPC CHANNEL NAME LENGTH), string] LPCWSTR
        path,
    [in] DWORD flags );

error status t EvtRpcOpenLogHandle(
    [in, range(1, MAX_RPC_CHANNEL_PATH_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] DWORD flags,
    [out] DWORD* numChannelPaths,
    [out, size is(,*numChannelPaths),
        range(0, MAX RPC CHANNEL COUNT), string]
        LPWSTR** channelPaths );

error status t EvtRpcGetChannelConfig(
    [in, range(1, MAX RPC CHANNEL NAME LENGTH), string]
        LPCWSTR channelPath,
    [in] DWORD flags,
    [out] EvtRpcVariantList* props );

error status t EvtRpcPutChannelConfig(
    [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] DWORD flags,
    [out] DWORD* numPublisherIds,
    [out, size_is(,*numPublisherIds),
        range(0, MAX_RPC_PUBLISHER_COUNT), string]
        LPWSTR** publisherIds );

error status t EvtRpcGetPublisherListForChannel(
    [in, string] 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, 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,
    [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*
            eventMetaDatEnum );

    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, range(1, MAX RPC CHANNEL NAME LENGTH), string] LPCWSTR logName,
        [in] LCID locale,
        [in] DWORD flags,
        [out, string] LPWSTR* displayName);
}
```

7 Appendix B: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows Server 2008
- Windows Vista

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 1.6: The EventLog Remoting Protocol Version 6.0 is supported only on Windows Vista and Windows Server 2008, which implement both EventLog Remoting Protocol Version 6.0 and the original EventLog Remoting Protocol.

<2> 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].

<3> 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].

<4> Section 1.8.4: Windows uses only the values specified in [MS-ERREF] section 2.3.

<5> Section 3.1.4: All errors are specified in [MS-ERREF], as specified in 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 Windows Vista and Windows Server 2008, the server does not validate the path and query parameters directly, and will return ERROR_EVT_INVALID_QUERY 0x00003A99) or ERROR_EVT_INVALID_CHANNEL_PATH (0x00003A98).

<8> Section 3.1.4.9: In Windows Vista and Windows Server 2008s, 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 contains the proper signature. As such, it is possible to fool the server, especially if the handle has been obtained from a different method in this specification.

<9> Section 3.1.4.11: In Windows Vista and Windows Server 2008s, 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 contains the proper signature. As such, it is possible to fool the server, especially if the handle has been obtained from a different method in this specification.

<10> Section 3.1.4.12: In Windows Vista and Windows Server 2008s, the server does not validate the flags. It will ignore any unrecognized flags; will assume that the path is a file if not specified; and will iterate from oldest to newest if a direction flag is unspecified.

<11> Section 3.1.4.12: In Windows Vista and Windows Server 2008, the server may omit the invalid channels.

<12> Section 3.1.4.14: In Windows Vista and Windows Server 2008s, the sign of the flag is not validated by the server. In the case of the EvtSeekRelativeToFirst flag with a negative value, the cursor of the result set remains set at the first record; in the case of EvtSeekRelativeToLast with a positive value, the cursor remains at the last record.

<13> Section 3.1.4.15: For more information on attributes, see [MSDN-FILEATT].

<14> 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.

<15> Section 3.1.4.17: In Windows Vista and Windows Server 2008s, 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.

<16> Section 3.1.4.22: In Windows Vista and Windows Server 2008s, the server does not validate the path parameter, and will start a new, partially configured registration if supplied with an invalid name. In this case, the sub error fields of the error struct MAY contain ERROR_INVALID_PARAMETER (0x00000057.)

<17> Section 3.1.4.22: Permissions for the **EvtRpcAssertConfig (Opnum 15)**, **EvtRpcRetractConfig (Opnum 16)**, and **EvtRpcPutChannelConfig (Opnum 21)** methods are controlled by ACLs on registry keys that are in the following branches of the registry:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers

For example, ACLs on the following registry key would provide access control in the configuration for the "TaskScheduler/Operational" channel:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels\Microsoft-Windows-TaskScheduler/Operational

<18> Section 3.1.4.28: In Windows Vista and Windows Server 2008s, the method does not fail when there is no metadata to return.

<19> Section 3.1.4.28: 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.

<20> Section 3.1.4.29: In Windows Vista and Windows Server 2008s, 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.

<21> Section 3.1.4.29: Permissions for the **EvtRpcAssertConfig (Opnum 15)**, **EvtRpcRetractConfig (Opnum 16)**, and **EvtRpcPutChannelConfig (Opnum 21)** methods are controlled by ACLs on registry keys that are in the following branches of the registry:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers

For example, ACLs on the following registry key would provide access control in the configuration for the "TaskScheduler/Operational" channel:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels\Microsoft-Windows-TaskScheduler/Operational

<22> Section 3.1.4.29: In a Windows implementation, the client provides functionality to change publisher data.

<23> Section 3.1.4.30: In Windows Vista and Windows Server 2008, 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.

<24> Section 3.1.4.30: Permissions for the **EvtRpcAssertConfig (Opnum 15)**, **EvtRpcRetractConfig (Opnum 16)**, and **EvtRpcPutChannelConfig (Opnum 21)** methods are controlled by ACLs on registry keys that are in the following branches of the registry:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers

For example, ACLs on the following registry key would provide access control in the configuration for the "TaskScheduler/Operational" channel:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Channels\Microsoft-Windows-TaskScheduler/Operational

<25> Section 3.1.4.31: For example, in the Windows implementation, substitution parameters are denoted by "%1", "%2", etc. an event message may be "Error number %1 occurred on disk drive %2." To format this message, the client could specify the eventId which denotes this event in the eventId parameter, and supply the strings "5" and "C:" in the values parameter. If the client supplied a buffer large enough in the strings parameter, the server would set that buffer to "Error number 5 occurred on disk drive C:"

<26> Section 3.1.4.36: In Windows Vista and Windows Server 2008s, the server does not validate the *flags* parameter. The server responds to flag 0x100, and ignores all others.

<27> Section 3.1.4.36: This API only succeeds if called for a channel that is also exposed by the Eventlog Remote Protocol. If this is called on other channels, the method fails.

8 Index

U

V

W