[MS-EERR]:
ExtendedError Remote Data Structure

Revision Summary

Date	Revision History	Revision Class	Comments
03/14/2007	1.0		Version 1.0 release
04/10/2007	1.1		Version 1.1 release
05/18/2007	1.2		Version 1.2 release
06/08/2007	1.2.1	Editorial	Revised and edited the technical content.
07/10/2007	2.0	Major	Added new normative reference.
08/17/2007	2.0.1	Editorial	Revised and edited the technical content.
09/21/2007	2.1	Minor	Updated the technical content.
10/26/2007	3.0	Major	Converted document to unified format.
01/25/2008	3.0.1	Editorial	Revised and edited the technical content.
03/14/2008	3.0.2	Editorial	Revised and edited the technical content.
06/20/2008	3.1	Minor	Updated the technical content.
07/25/2008	3.2	Minor	Updated the technical content.
08/29/2008	3.3	Minor	Updated the technical content.
10/24/2008	4.0	Major	Updated and revised the technical content.
12/05/2008	4.1	Minor	Updated the technical content.
01/16/2009	4.1.1	Editorial	Revised and edited the technical content.
02/27/2009	4.1.2	Editorial	Revised and edited the technical content.
04/10/2009	4.1.3	Editorial	Revised and edited the technical content.
05/22/2009	4.2	Minor	Updated the technical content.
07/02/2009	4.2.1	Editorial	Revised and edited the technical content.
08/14/2009	4.2.2	Editorial	Revised and edited the technical content.
09/25/2009	5.0	Major	Updated and revised the technical content.
11/06/2009	5.0.1	Editorial	Revised and edited the technical content.
12/18/2009	5.0.2	Editorial	Revised and edited the technical content.
01/29/2010	5.1	Minor	Updated the technical content.
03/12/2010	5.1.1	Editorial	Revised and edited the technical content.

Date	Revision History	Revision Class	Comments
04/23/2010	5.1.2	Editorial	Revised and edited the technical content.
06/04/2010	6.0	Major	Updated and revised the technical content.
07/16/2010	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
08/27/2010	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
10/08/2010	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
11/19/2010	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
01/07/2011	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
02/11/2011	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
03/25/2011	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
05/06/2011	6.0	No change	No changes to the meaning, language, or formatting of the technical content.
06/17/2011	6.1	Minor	Clarified the meaning of the technical content.

Contents

1 Introduction

This specification for encoding extended error information assumes that the reader has familiarity with the concepts and the requirements that are detailed in [MS-RPCE] and [C706].

The purpose of the encoding that this specification defines is to allow a software agent on one network node to communicate a rich (or extended) error to a software agent on another network node. This specification does not define how an extended error is transmitted between network nodes. A protocol outside this specification MUST be used for that purpose. This specification only defines the encoding rules for an extended error.

1.1 Glossary

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

de-serialize
error sequence
immediate error cause
Interface Definition Language (IDL)
marshal
Microsoft Interface Definition Language (MIDL)
Network Data Representation (NDR)
remote procedure call (RPC)
RPC transfer syntax
serialize
Unicode
unmarshal

The following terms are specific to this document:

error record: A structured description of an occurrence of an error. For more information, see section 1.3.

interface: This term is used exactly as specified in [C706] section "Introduction to the RPC API" in Part 2.

root error: The last error in an **error sequence**. For more information, see section 1.3.

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

References to Microsoft Open Specification documents do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

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

[ISO/IEC-8859-1] International Organization for Standardization, "Information Technology -- 8-Bit Single-Byte Coded Graphic Character Sets -- Part 1: Latin Alphabet No. 1", ISO/IEC 8859-1, 1998, http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=28245&ICS1=35&ICS2=40&ICS3=

Note There is a charge to download the specification.

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

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

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

1.2.2 Informative References

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

[MS-RPCH] Microsoft Corporation, "Remote Procedure Call over HTTP Protocol Specification".

1.3 Overview

In complex distributed systems, a situation may arise where an error encountered on one network node must be communicated to another network node. A protocol that is used to transmit data between network nodes usually has some provisions to transmit errors in its messages, but often the error that is being communicated is a single unsigned integer or a single unsigned integer plus a short string. As the complexity of the system and/or the number of network nodes that are involved grows, a single unsigned integer and/or a short string may prove insufficient for quick and efficient troubleshooting of all possible scenarios.

This specification defines an encoding for a rich, structured error called an extended error. After the extended error is encoded, it must be transmitted between network nodes by a protocol outside this specification.

The extended error itself is used for troubleshooting a malfunctioning system and is intended to be used by a human reader or an automated failure diagnostic system. This specification does not prescribe what the extended error should be; it specifies the fields and field values that are used for encoding the extended error (see section 2). Protocols and systems are free to create and encode any extended error a support engineer or an expert user of the system may find useful to troubleshoot a malfunctioning system.

1.3.1 Extended Error Data Model

An extended error is one or more **error records** from an **error sequence**. Each error record in the error sequence contains up to four values that software agents can use to encode additional information about the error that occurred. These values are called parameters for the error in the error record. For example, if a file cannot be found, which causes a failure in a system, a parameter in the error record might be the name of the file that was not found.

Besides the four parameters, the error record contains the following data elements: a generating component, a detection location, and an error code.

The generating component is a unique numeric value that identifies the component or protocol layer where the error or failure occurred. The generating component should be unique within all implementations of this protocol.

The detection location is a numeric value that is unique within a given generating component and identifies the location in the component or protocol layer where the error occurred. Location can be any identifier inside a component or protocol layer that unambiguously describes where the error occurred or was detected. For example, a software agent may assign one detection location for each module or function inside that software agent. Alternately, a software agent may use line numbers to identify the location where the failure occurred or was detected. Any detection location is meaningful only within the context of a specific generating component; thus, the generating component is part of the namespace definition for a detection location.

The error code is an implementation-specific numeric value that specifies the error that occurred.

1.4 Relationship to Protocols and Other Structures

This specification uses type serialization, as specified in [MS-RPCE] section 2.2.6, to do the actual encoding of the extended error. In turn, [MS-RPCE] and [MS-RPCH] use this specification to transmit extended errors. The processing rules and the placement of the encoded extended error inside the [MS-RPCE] and [MS-RPCH] messages are defined in [MS-RPCE] sections 2.2.2.8 and 2.2.2.9 and in [MS-RPCH] section 2.1.2.1.

1.5 Applicability Statement

This specification is applicable in complex, distributed systems where the benefit of quick and efficient troubleshooting outweighs the cost of the increase in message size that transmitting additional troubleshooting information causes. Because this specification makes no assumptions about network topology or network communication, it is applicable in a broad range of scenarios.

1.6 Versioning and Capability Negotiation

None.

1.7 Vendor-Extensible Fields

The generating component and detection location as specified in section 1.3 are vendor-extensible. Generating components in the inclusive range of 0 to 255 are reserved by Microsoft. A vendor SHOULD define new generating components by using any value that is not reserved by Microsoft. This specification does not prescribe how vendors can avoid collisions in the generating components they choose.

A vendor SHOULD NOT use a detection location from a generating component that is not provided by that vendor.

2 Structures

2.1 Transport

This specification defines only encoding rules and does not define how the encoded data is transmitted on the network. As such, it does not have a transport. It relies upon other protocols that use it (see section 1.4) to carry it as its transport.

2.2 Structure Syntax

This section defines the syntax for encoding the extended errors.

2.2.1 Common Types

This section defines the types and structures used by this specification.

2.2.1.1 EEAString

The **EEAString** structure encodes strings of ANSI characters, as specified in [ISO/IEC-8859-1], that contain troubleshooting information.

```
typedef struct tagEEAString {
  short nLength;
  [size_is(nLength)] BYTE* pString;
} EEAString;
```

nLength: This field MUST contain the size of **pString** in bytes.

pString: A NULL-terminated ANSI string that contains troubleshooting information.

2.2.1.2 EEUString

The **EEUString** structure encodes **Unicode** strings that contain troubleshooting information. The **EEComputerName** structure uses this type.

```
typedef struct tagEEUString {
  short nLength;
  [size_is(nLength)] unsigned short* pString;
} EEUString;
```

nLength: This field MUST contain the length of **pString** in characters.

pString: A NULL-terminated Unicode string that contains troubleshooting information.

2.2.1.3 BinaryEEInfo

The **BinaryEEInfo** structure encodes binary data that contains troubleshooting information.

```
typedef struct tagBinaryEEInfo {
  short nSize;
  [size_is(nSize)] unsigned char* pBlob;
```

```
} BinaryEEInfo;
```

nSize: This field MUST contain the size of **pBlob** in bytes.

pBlob: Binary data that contains troubleshooting information.

2.2.1.4 ExtendedErrorParamTypesInternal

The **ExtendedErrorParamTypesInternal** enumeration defines the values that are valid for the **Type** field in the **ExtendedErrorParam** structure.

```
typedef  enum tagExtendedErrorParamTypesInternal
{
  eeptiAnsiString = 1,
  eeptiUnicodeString = 2,
  eeptiLongVal = 3,
  eeptiShortValue = 4,
  eeptiPointerValue = 5,
  eeptiNone = 6,
  eeptiBinary = 7
} ExtendedErrorParamTypesInternal;
```

eeptiAnsiString: The **ANSIString** member of the union is valid.

eeptiUnicodeString: The **UnicodeString** member of the union is valid.

eeptiLongVal: The **LVal** member of the union is valid. **LVal** is used to encode a long.

eeptiShortValue: The **IVal** member of the union is valid. **IVal** is used to encode a short.

eeptiPointerValue: The **PVal** member of the union is valid. **PVal** is used to encode an __int64.

eeptiNone: No additional details are present in this parameter.

eeptiBinary: The **Blob** member of the union is valid.

2.2.1.5 ExtendedErrorParam

The **ExtendedErrorParam** structure contains a parameter, as described in section 1.3.1, that provides additional details about the error record.

```
typedef struct tagParam {
  ExtendedErrorParamTypesInternal Type;
  [switch_type(short), switch_is(Type)]
    union {
    [case(1)]
      EEAString AnsiString;
    [case(2)]
      EEUString UnicodeString;
    [case(3)]
      long LVal;
    [case(4)]
      short IVal;
    [case(5)]
```

```
      __int64 PVal;
    [case(6)]
      ;
    [case(7)]
      BinaryEEInfo Blob;
  };
} ExtendedErrorParam;
```

Type: Indicates which member of the union is valid. **ExtendedErrorParamTypesInternal** lists all of the possible values.

AnsiString: A parameter of type **EEAString**.

UnicodeString: A parameter of type **EEUString**.

LVal: This parameter MUST be used to encode long values that contain troubleshooting information.

IVal: This parameter MUST be used to encode integer values that contain troubleshooting information.

PVal: This parameter MUST be used to encode 64-bit integer values that contain troubleshooting information.

Blob: A parameter of type **BinaryEEInfo**.

2.2.1.6 EEComputerNamePresent

The **EEComputerNamePresent** enumeration defines the allowed values for the **Type** field in the **EEComputerName** structure.

```
typedef  enum tagEEComputerNamePresent
{
  eecnpPresent = 1,
  eecnpNotPresent
} EEComputerNamePresent;
```

eecnpPresent: **Name** member of the **EEComputerName** structure is valid and contains a network node identifier.

eecnpNotPresent: This structure does not contain a network node identifier.

2.2.1.7 EEComputerName

The **EEComputerName** structure identifies the network node on which the error record was generated.

```
typedef struct tagEEComputerName {
  EEComputerNamePresent Type;
  [switch_type(short), switch_is(Type)]
    union {
    [case(1)]
      EEUString Name;
    [case(2)]
```

```
    ;
  };
} EEComputerName;
```

Type: Indicates the contents of a union.

Value	Meaning
eecnpPresent 1	Network Node Identifier **Name** member of the union is valid and contains a network node identifier.
eecnNotPresent 2	No Network Node Identifier This structure does not contain a network node identifier.

Name: Unicode string that identifies the network node on which the error record was generated. The format in which the network node is identified is implementation-specific, and this information MUST be used for display purposes only. This specification does not define what the format is. Software agents who use this structure SHOULD use a network node identifier that is unique within a specific topology and is descriptive to a human reader. If **Type** is equal to **eecnpNotPresent**, the error record MUST be interpreted as generated on the local network node.

2.2.1.8 ExtendedErrorInfo

The **ExtendedErrorInfo** structure represents an error record.

```
typedef struct tagExtendedErrorInfo {
  ExtendedErrorInfo* Next;
  EEComputerName ComputerName;
  unsigned long ProcessID;
  __int64 TimeStamp;
  unsigned long GeneratingComponent;
  unsigned long Status;
  unsigned short DetectionLocation;
  unsigned short Flags;
  short nLen;
  [size_is(nLen)] ExtendedErrorParam Params[];
} ExtendedErrorInfo;
```

Next: An error record for the **immediate error cause** for this error record. For the **root error**, it MUST be set to NULL.

ComputerName: Network node identifier as specified in section 2.2.1.7.

ProcessID: The ID of the process in which the error occurred.

TimeStamp: Time at which the error record was generated, which is expressed as the number of 100-nanosecond intervals since January 1, 1601. It MUST be interpreted as Coordinated Universal Time (UTC).

GeneratingComponent: Component or protocol layer identifier where the error occurred as described in section 1.3.1.

Status: Error code as described in section 1.3.1.

DetectionLocation: Location where the error occurred as described in section 1.3.1.

Flags: One or more flags that specify the presence or absence of other error records in the error sequence.

Value	Meaning
0x0000	All of the error records from the error sequence are present in the encoding.
0x0001	One or more error records from the error sequence before the current record are not present in the encoding.
0x0002	One or more error records from the error sequence after the current record are not present in the encoding.

nLen: Number of elements in the Params array. MUST be less than or equal to 4.

Params: Array of error parameters as described in the data model in section 1.3.1.

2.2.2 Extended Error Interface

The Extended Error Interface supports two operations: encoding and decoding.

It does not contain any **remote procedure call (RPC)** methods. It only contains a type that MUST be encoded/decoded by using the type serialization functionality as specified in [MS-RPCE] section 2.2.6.

2.2.2.1 Encoding an Extended Error

The encoding of the extended error is the output of **marshaling** the first element of the error sequence by using type serialization version 1, as specified in [MS-RPCE] section 2.2.6. Because the error records are linked by the **Next** field of the **ExtendedErrorInfo** structure, marshaling the first element marshals the entire error sequence.

2.2.2.2 Decoding an Extended Error

The decoding of the extended error is done by **unmarshaling** the encoded extended error by using type serialization version 1, as specified in [MS-RPCE] section 2.2.6. Any violation of this specification MUST cause the entire decoding to fail.

2.2.3 Well-Known Detection Locations

This specification defines the following well-known detection locations and generating components that automated troubleshooting software agents may use for automatic failure diagnosis. If an implementation uses these detection locations and generating components, it MUST use them to encode error information whose meaning is consistent with the meaning in the following table.<1>

Detection location	Generating component	Description
0x000005A0	0x0000000E	An attempt by an RPC over HTTP proxy was made to connect to an RPC over HTTP server, and the connect attempt failed. For more information, see [MS-RPCH] section 3.2.3 and section 3.2.4.

3 Structure Examples

3.1 Using the Data Model with a Fictitious Extended Error

The following example illustrates how the data model is used with a fictitious extended error taken from a Microsoft Windows® environment. In this example, the RPC runtime encountered a failure reading a registry key and generated an error record by using the following field values:

Next: Because this failure is a root cause failure, the software agent will not link this to an immediate error cause and will set the **Next** field to NULL.

ComputerName: The failure originated on the local node, and the **Type** field in the **ComputerName** structure is set to eecnNotPresent.

TimeStamp: This field is set to the current time at the occurrence of the failure.

GeneratingComponent: This field is set to 0x00000049, which is a Microsoft reserved generating component.

Status: This field is set to 0x00000002, which is the Win32 error code for the ERROR_FILE_NOT_FOUND error.

DetectionLocation: This field is set to 0x00000BF0, which uniquely identifies the place in the RPC runtime where this registry key is being read.

Flags: This field is set to 0x0000.

nLen: This field is set to 0x0001 as one parameter with additional details is present.

Params: An array of one element, which has the following field values:

Type: This field is set to "eeptiUnicodeString", which indicates that the parameter is of type **EEUString**. The **pString** member of the **UnicodeString** structure is set to "\Software\Policies\Microsoft\Windows NT\Rpc\RestrictRemoteClients", which is the name of the registry key that the RPC runtime tried to open.

4 Security Considerations

This specification has no security protection. Software agents who use this specification SHOULD evaluate the sensitivity of the data that is being encoded and provide protection from tampering and information disclosure if sensitive data is being encoded and transmitted through a public network.

5 Appendix A: Full IDL

For ease of implementation, this specification provides the full **IDL**, where "ms-dtyp.idl" is the IDL found in [MS-DTYP] Appendix A.

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

[
uuid(14a8831c-bc82-11d2-8a64-0008c7457e5d),
version(1.0),
pointer_default(unique)
]
interface ExtendedError
{

typedef struct tagEEAString
{
short nLength;
[size_is(nLength)] byte *pString;
} EEAString;

typedef struct tagEEUString
{
short nLength;
[size_is(nLength)] unsigned short *pString;
} EEUString;

typedef struct tagBinaryEEInfo
{
short nSize;
[size_is(nSize)] unsigned char *pBlob;
} BinaryEEInfo;

typedef enum tagExtendedErrorParamTypesInternal
{
eeptiAnsiString = 1,
eeptiUnicodeString,
eeptiLongVal,
eeptiShortVal,
eeptiPointerVal,
  eeptiNone,
eeptiBinary
} ExtendedErrorParamTypesInternal;


typedef struct tagParam
{
ExtendedErrorParamTypesInternal Type;
[switch_type(short),switch_is(Type)] union {
[case(1)] EEAString AnsiString;
[case(2)] EEUString UnicodeString;
[case(3)] long LVal;
[case(4)] short IVal;
[case(5)] __int64 PVal;
[case(6)] ;
[case(7)] BinaryEEInfo Blob;
};
} ExtendedErrorParam;
```

```
typedef enum tagEEComputerNamePresent
{
eecnpPresent = 1,
eecnpNotPresent
} EEComputerNamePresent;

typedef struct tagEEComputerName
{
EEComputerNamePresent Type;
[switch_type(short),switch_is(Type)] union {
[case(1)] EEUString Name;
[case(2)] ;
};
} EEComputerName;

typedef struct tagExtendedErrorInfo
{
struct tagExtendedErrorInfo * Next;
EEComputerName ComputerName;
unsigned long ProcessID;
__int64 TimeStamp;
unsigned long GeneratingComponent;
unsigned long Status;
unsigned short DetectionLocation;
unsigned short Flags;
short    nLen;
[size_is(nLen)] ExtendedErrorParam Params[];
} ExtendedErrorInfo;

typedef ExtendedErrorInfo *ExtendedErrorInfoPtr;
}
```

6 Appendix B: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft Windows® 2000 operating system
- Windows® XP operating system
- Windows Server® 2003 operating system
- Windows Vista® operating system
- Windows Server® 2008 operating system
- Windows® 7 operating system
- Windows Server® 2008 R2 operating system

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

<1> Section 2.2.3: Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2 use these generating components and detection locations to provide automatic diagnosis of failures.

7 Change Tracking

This section identifies changes that were made to the [MS-EERR] protocol document between the May 2011 and June 2011 releases. Changes are classified as New, Major, Minor, Editorial, or No change.

The revision class **New** means that a new document is being released.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- The removal of a document from the documentation set.
- Changes made for template compliance.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **Editorial** means that the language and formatting in the technical content was changed. Editorial changes apply to grammatical, formatting, and style issues.

The revision class **No change** means that no new technical or language changes were introduced. The technical content of the document is identical to the last released version, but minor editorial and formatting changes, as well as updates to the header and footer information, and to the revision summary, may have been made.

Major and minor changes can be described further using the following change types:

- New content added.
- Content updated.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.

- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.
- Content removed for template compliance.
- Obsolete document removed.

Editorial changes are always classified with the change type **Editorially updated.**

Some important terms used in the change type descriptions are defined as follows:

- **Protocol syntax** refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.
- **Protocol revision** refers to changes made to a protocol that affect the bits that are sent over the wire.

The changes made to this document are listed in the following table. For more information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
1.2 References	Added explanatory statement regarding the removal of the publishing year from Microsoft Open Specification document references.	N	Content updated.

8 Index