



IPDR/XDR Encoding Format

Version 3.6

March 19, 2007

© 1999-2007 IPDR.org, Inc.

Preface

Contacts

For general questions regarding this document and referrals to technical experts for detailed questions, please contact:

IPDR.org Editor-in-Chief:
Steve Cotton
scotton@ipdr.org

Abstract

This document specifies the IPDR/XDR Encoding Format.

Change History

Version	Date	Name	Description
Initial Draft	3/19/07	Steve Cotton	Add ipAddr type
3.5.2 Draft-1	1/24/07	Amit Kleinmann	Addition of support of User Defined Types, Terminology changes to IPDR Types, as well as overall editorial changes
3.5.2 Draft-2	1/25/07	Amit Kleinmann	Change the Structured flag bit to User Defined Type flag bit, remove the array flag bit since it is not needed
3.5.2 Draft-3	1/30/07	Amit Kleinmann	Remove the useless ability to define XDR arrays of indefinite length Correct the XDR Examples General editorial changes and clarification
3.6	3/2/2007	S. Cotton	Production Release

Acknowledgements

Ty Roach, ACE*COMM

Greg Weber, Cisco Systems

Jeff Meyer, Hewlett Packard

Tal Givoly, Amdocs

Amit Kleinmann, Amdocs

Shai Gotlib, Amdocs

Steve Cotton, Cotton Management Consulting

Table of Contents

1	GLOSSARY	6
1.1	Terminology	6
1.2	Acronyms	6
2	INTRODUCTION	7
3	REQUIREMENTS	8
3.1	Space Efficient	8
3.2	Efficient parsing and processing	9
3.3	Low Complexity	9
3.4	Existing standards work	9
3.5	Performance and size characteristics	9
3.6	Well defined mapping	9
3.7	Extension and expansion	10
3.8	Self-describing	10
4	XDR STRUCTURE DEFINITION	11
5	COMPACT FORMAT DEFINITION	15
5.1	Compact Document Header	15
5.1.1	Version	16
5.1.2	Recorder Info	16
5.1.3	Creation Time	16
5.1.4	Default Namespace	16
5.1.5	Other Namespaces	16
5.1.6	Service Definitions	17
5.1.7	Document UUID	17
5.2	Stream Elements	18
5.2.1	Record Descriptor	18
5.2.2	Attribute Name	20
5.2.3	Attribute Type	20
5.2.4	Elementary Types	21
5.2.5	User Defined Types	22
5.2.6	Data Representation of IPDR Types using IPDR XDR encoding	23
5.2.7	IPDR Record Data	38
5.3	Document End	39
6	REFERENCES	40

Table of Figures

Figure 1: Attribute Type ID - Bit structure 20

1 Glossary

1.1 Terminology

Attribute Type (a.k.a. Type, Data Type, or IPDR Data Type)

A (non-empty) set of attribute values, that represents a potential for conveying information. While only attribute values are actually conveyed, their attribute type governs the domain of possibilities. The values of such an attribute type, in turn, are the set of permitted contents of that attribute. It is by selecting one particular attribute rather than the others, that the sender of an IPDR message (or IPDR file) conveys information.

An IPDR Data Type can be either Elementary Type or User Defined Type.

Compound Type

A User Defined Type that is non scalar. It is either Array Type or Structured Type.

EPOCH

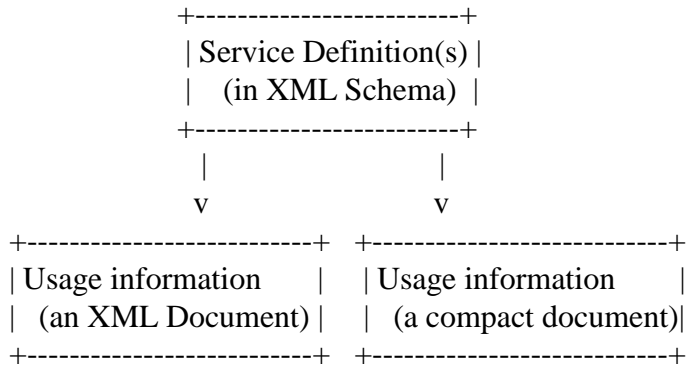
An instant chosen as the origin of a particular [time](#) scale. It serves as a reference point from which time is measured. In the context of IPDR in general, and this specification in particular, it is defined as: midnight, Universal Time, on January 1, 1970 (as defined by the Unix epoch, Mac OS X and Java). Thus the [EPOCH](#) is the time 00:00:00 UTC on [January 1, 1970](#).

1.2 Acronyms

ASCII	American Standard Code for Information Interchange
CDR	Call detail Records
CRANE	Common Reliable Accounting for Network Elements
GUID	Globally Unique IDentifier
IPDR	Internet Protocol Detail Record
UUID	Universal Unique Identifier
XDR	External Data Representation
XML	Extensive Markup Language

2 Introduction

IPDR/XDR is an encoding format that offers a compact and efficient representation of usage accounting data described by IPDR/Service Definitions. The relationship between IPDR/XDR and IPDR/XML is shown below.



The IPDR/XDR encoding represents integers and floating point values and other binary data such as addresses and timestamps in a binary format as opposed to XML [5] that uses an ASCII format. This increases the efficiency of reading and writing these values. The identification of usage attributes is done using the same ASCII names used in the XML format [5]. These names are defined in the Service Definitions. XML namespace concepts that prevent name collisions are also supported.

The primary space savings comes from the separation of description of the usage attributes (i.e., their name and type) from the actual event values.

In situations where multiple instances of the same event format occur, this separation allows description information to be written only once in the document. Only the values are written for every event. This produces approximately four times improvement in storage space and processing efficiency based on initial observations.

The IPDR/XDR format allows for both encoding and decoding entities to process the documents in a stream-oriented fashion. That is, a decoding entity does not need to read to the end of the document before beginning to extract information. Similarly, an encoding entity does not need to have all the information in memory before beginning to write the document.

This streaming property can be critical when processing large sets of accounting information.

3 Requirements

The IPDR Protocol Working Group set the following requirements for a compact encoding format:

- The format should be more space efficient than the XML equivalent (preferably by several times).
- The format should allow for efficient parsing and processing.
- Complexity should be minimized in order to spur adoption and inter-working systems, and aid in debugging.
- The format should ideally draw from existing standards work.
- The format should provide performance and size characteristics that are comparable to known vendor proprietary formats.
- There must be well defined mapping from the service definition to the encoding.
- The format should allow for easy extension and expansion of the defined services.
- The format should be self-describing. That is, knowing an encoding is an IPDR document, a system should be able to extract the recorded usage information without additional external control.

3.1 Space Efficient

The exact savings is dependent on the length of the attribute names from the service definition, the ratio of Elementary Types to string oriented data, and the number of occurrences of similar usage events in a document.

Consider a record which contains the following five usage attributes:

```
<IPDR xsi:type="AA-Type">
  <subscriberId>joe</subscriberId>
  <ipAddress>192.168.2.64</ipAddress>
  <nasIdentifier>nas1.foo.com</nasId>
  <acctInputOctets>13444</acctInputOctets>
  <acctOutputOctets>7777</acctOutputOctets>
</IPDR>
```

Not including blanks and linefeeds, this record occupies 216 bytes.

The compact encoding of this record is as follows:

```
00 00 00 03      6A 6F 65 C0      A8 02 40 00      00 00 0C 6E      61 73 6C 2E      66 6F 6F 2E
length = 3      j o e 192      168 2 64      length = 12      n a s 1 .      f o o .
63 6F 6D 00      00 34 84 00      01 2F D1
c o m      1 3 4 4 4      7 7 7 7 7
```

The compact encoding takes 35 bytes and is about 6 times more compact than the equivalent XML encoding.

In this example, the one time header costs in either document format are not shown or counted. If only one or two usage events are encoded in an instance document, then the compact format

savings will be significantly less than this factor. However, as the number of events of a similar type increases, the savings approaches this value.

3.2 Efficient parsing and processing

Because numeric values are represented in binary format, they may be memory transferred directly into variables. In contrast, ASCII representations of numbers need to be parsed or formatted from or to their ASCII equivalents. This is what yields the processing savings in addition to the memory savings.

The descriptor format allows processing entities to operate in a table driven approach for the production and consumption of different record types.

3.3 Low Complexity

There are thirteen Basic Types to encode and decode. These types correspond to the types used by most modern programming languages. Note that these types include unsigned integral types as well as signed types.

Please note: the Java language does not natively support unsigned types, but can accommodate them by upcasting to the larger integral type (e.g., unsigned int to long), for all types except unsigned long.

Record descriptors have a simple name based format. The overall document structure is closely related to XDR [4] which allows the format to be described in a C-like language, which is concise and well understood.

3.4 Existing standards work

The encoding policy for Elementary values is based on XDR [4], the service definition format is common with that for XML IPDR Services and is a subset of XML-Schema [5]. Support for multi-byte character content is based on UTF-8 [6].

3.5 Performance and size characteristics

Clear savings in space and processing are apparent from the example cited above. A factor of 4 or 5 in savings is a reasonable rule of thumb.

3.6 Well defined mapping

The names of usage attributes are common in the IPDR/XML and IPDR/XDR models and derive from the same IPDR/Service Definitions. IPDR/Service Definitions are constrained to guarantee this mapping is enabled. This is in one of the motivations for the IPDR/Service Specification guidelines [2], which restrict the use of the standard XML-Schema [5] to ensure the ability to encode information in either IPDR/XML or IPDR/XDR.

3.7 Extension and expansion

The means for extending service definitions by third parties as well as within IPDR.org is inherited from the IPDR/Service Specification guidelines [2]. It simply involves the creation of additional XML-Schema documents to describe the new service and the proper use of XML-Namespaces.

3.8 Self-describing

The document contains versioning information at the front of the header to support changes over time. The document contains references via URI to namespace and service definitions and contains descriptor blocks that identify usage attributes in events according to their XML-Schema based service definition. This produces a fully self-describing document.

Note that this assumes that the service definition URI information is present and accessible to the processing entity.

4 XDR Structure Definition

The compact encoding structure is defined using XDR notation [4]. This provides a concise and readable means to convey the elements of the encoded format. It also provides a means to map to an encoding according to the XDR RFC [4], with one modification - XDR [4] specifies a padding of all encoded types to 4 byte boundaries. The padding to 4 byte boundaries is removed in IPDR 3.5. This means that encodings in compliance with the IPDR 3.0, 3.1 or 3.1.1 formats may differ from those in IPDR 3.5 and beyond. IPDR 3.1 and 3.1.1 did follow this guideline, but analysis of IPDR/Streaming indicated that it was not desirable to preserve. For this reason the version number in IPDR/XDR 3.5 is incremented to 4 from the previous version 3. Note that versions numbers 1 and 2 were not used.

```
/* IPDRDoc.xdr
 *
 *   This file defines the structure of IPDRDocuments in a
 *   compact format. It is expected that IPDRDocuments will
 *   be stored in External Data Representation (XDR)
 *   format according to the structures defined in this
 *   file.
 *
 *   XDR is defined in RFC1832 [4].
 *
 *   In order to accommodate efficient encoding of large
 *   sets of data, the compact IPDR deviates from XDR
 *   when encoding the length field associated with
 *   IPDRRecordData and the set of IPDRStreamElements.
 *
 *   The rules are only modified to remove the 4 byte
 *   boundary padding constraint. I.e. all fields are
 *   placed adjacent to one another with no padding employed.
 *
 *   Note that only data structures are defined here.
 *   This defines a format/structure to encode IPDRDocs it
 *   does not define any transport.
 */

/*
 *   Define our own string type, UTF8String. This emphasizes
 *   that the content may not be ASCII, but may be UTF-8 encoded
 *   representation of Unicode/ISO character set.
 *
 *   Note that strings consisting of basic 7-bit ASCII characters
 *   are unchanged when represented in UTF-8.
 */
typedef opaque UTF8String<>;

/* AttributeDescriptor
 *
 *   An attribute descriptor defines one attribute which will
 *   appear in a record. It contains the attribute name and
 *   the attributes type.
 */
struct AttributeDescriptor {
    UTF8String  attributeName; /* The name of an attribute. This name will either
 * come from the default namespace or it will come
 * from an alternate namespace as defined in the
 * header. If an alternate namespace is used, the name
 * will use XML syntax of the form namespaceId:attributeName
 */
```

```

int    typeId;    /* Describes the type of this attribute. Explicitly
                  * including the type allows us to continue parsing
                  * a record even when the attribute name is unrecognized.
                  */
};

/* RecordDescriptor
 *
 * A record descriptor defines the set of attributes which are
 * carried in each record of a designated type. If different
 * subsets of attributes are carried in different records, then
 * a record descriptor is required for each.
 *
 * The record descriptor contains a sequence of attributeDescriptors
 * which identify the name and type of the attributes which are
 * associated with this record type. The type definition will
 * allow an entity which does not recognize some of the attributes
 * to still successfully parse the document.
 */
struct RecordDescriptor {
    int    descriptorId;    /* The descriptorId is an integer which will allow
                            * individual records in a given document to specify
                            * which descriptor describes their layout.
                            */

    UTF8String    typeName;    /* identifies a record type defined in a service
                                * definition.
                                * Record types specify the optional and mandatory
                                * fields.
                                * Note that there may be multiple descriptors
                                * associated with the same named type if they
                                * use different sets of optional attributes.
                                */

    AttributeDescriptor    attributes<>;    /* Lists the attributes in the order they will appear
                                            * in records in this document. A record which points
                                            * to this descriptor's descriptorId, must exactly
                                            * match the set of attributes identified in this
                                            * list.
                                            */
};

/* IPDRRecordData
 *
 * An individual record is a packed set of attribute values
 * determined by a recordDescriptor. The values are encoded
 * into the octet array according to their Type format
 * and the XDR encoding rules. The overall length of the
 * opaque array is specified as well.
 */
typedef opaque IPDRRecordData<>;

/* IPDRRecord
 *
 * An ipdrRecord structure represents a single recorded usage
 * event. It is typically flat set of unique attributes. The specific
 * attributes are determined by the a recordDescriptor which is
 * referenced by index value.
 *
 * The set of attribute values are encoded according to XDR rules
 * as if a struct had been defined with elements of the type
 * identified in the recordDescriptor and in the order defined
 * by the recordDescriptor.
 *
 * Situations where repeating elements or substructures are present
 * are described in the IPDR/XDR 3.5 documentation.
 */
struct IPDRRecord {
    int    descriptorId;    /* The index value of the descriptor which
                            * describes this records attributes. It must
                            * match the descriptor id in a recordDescriptor

```

```

        * encountered earlier.
        */

    IPDRRecordData data;          /* The packed set of attributes */
};

/* IPDRDocEnd
 *
 * An IPDRDocEnd element is the last element in a sequence of
 * ipdrStreamElement's contained in the doc.
 */
struct IPDRDocEnd {
    int          count;          /* indicates the number of usage elements carried
                                * in this stream of usage events, or -1 if
                                * not used.
                                */

    hyper        endTime;       /* 64-bit time representation, defined to be
                                * the number of milliseconds since
                                * Jan. 1, 1970 0:00 GMT.
                                */
};

/*
 * Types of StreamElements
 */
enum ElementType {
    RECORDDESC = 1,             /* Describes a record structure          */
    IPDRREC    = 2,             /* The values of a single record        */
    DOCEND     = 3,             /* Indicates the end of the element stream */
};

/* IPDRStreamElement
 *
 * An ipdrStreamElement defines the types of non-header
 * records which are contained in an IPDRDoc.
 */
union IPDRStreamElement switch (ElementType kind) {
    case RECORDDESC: RecordDescriptor desc;
    case IPDRREC:    IPDRRecord      rec;
    case DOCEND:    IPDRDocEnd      docEnd;
};

/* NamespaceInfo
 *
 * This associates a namespace URI with a simple identifier. It is used
 * to incorporate namespaces besides default from which attributes
 * are defined.
 */
struct NamespaceInfo {
    UTF8String namespaceURI; /* The URI for an alternate namespace. For example
                              * http://www.foo.com/ipdr/namespace
                              */

    UTF8String namespaceID; /* A string identifier which will be used to prefix attributes
                              * which come from this namespace (e.g. "foo" would
                              * result in attributes being named like "foo:specialAttr"
                              */
};

/* IPDRHeader
 *
 * The IPDRHeader contains general information about this document,
 * including versioning, information about the recorder and
 * references to IPDR Service definitions and their namespaces.
 */
struct IPDRHeader {
    int          version;       /* The compact format version. Moves to 4 for
                                * IPDR 3.5 (was 3 for IPDR 3.0,3.1)
                                */
};

```

```

UTF8String      ipdrRecorderInfo; /* Identification information for the producer
                                   * of this document.  Typcially URI, or blank
                                   */

hyper           startTime;        /* Number of seconds since the Epoch
                                   * (00:00:00 UTC, January 1, 1970)
                                   */

UTF8String defaultNamespaceURI; /* Identifies the default Namespace associated with
                                   * this record.  Those attribute names which are
                                   * unqualified will be assumed to be from this
                                   * namespace
                                   * (e.g. "http://www.ipdr.org/namespace)
                                   */

NamespaceInfo otherNameSpaces<>; /* Identifies additional namespaces from which
                                   * attributes are derived.  This list may be empty.
                                   */

UTF8String serviceDefinitionURIs<>; /* Identifies the set of service definitions from
                                   * which the records are produced.
                                   */

opaque docId<>; /* The UUID associated with this document */
};

/* IPDRDoc
 *
 * An IPDRDoc represents a collection of usage events recorded
 * in a compact binary format consistent with the External
 * Data Representation (XDR defined in RFC1832 [4]).
 *
 * The information content of the compact format is equivalent
 * to that contained in an XML encoded form.
 *
 * Note that the length of the elements array is specified as well.
 */
struct IPDRDoc {
    IPDRHeader      header; /* contains information describing this
                              * IPDR document
                              */

    IPDRStreamElement elements<>; /* The descriptors and ipdrRecords may be
                                   * intermixed.  An ipdrRecord may not refer
                                   * to a descriptor which has not previously
                                   * appeared.  The ipdrDocEnd must be the final
                                   * element.
                                   */
};

```

5 Compact Format Definition

The basic structure is shown below:

```
+-----+
| Header |
+-----+
| Descriptor |
+-----+
| Usage Event |
+-----+
...
+-----+
| Usage Event |
+-----+
| Doc End |
+-----+
```

Multiple descriptors may appear in a single document. The descriptors may be intermixed with usage events. The only requirement is that a descriptor must appear before any usage event which references it.

The Doc End element may appear exactly once and must be the last element in the document.

5.1 Compact Document Header

The document header is structured as follows.

```
/* IPDRHeader
 *
 * The IPDRHeader contains general information about this document,
 * including versioning, information about the recorder and
 * references to IPDR Service definitions and their namespaces.
 */
struct IPDRHeader {
    int version; /* The compact format version. Moves to 4 for
                * IPDR 3.5 (was 3 for IPDR 3.0,3.1)
                */
    UTF8String ipdrRecorderInfo; /* Identification information for the producer
                                * of this document. Typcially URI, or blank
                                */
    hyper startTime; /* Number of seconds since the Epoch
                    * (00:00:00 UTC, January 1, 1970)
                    */
    UTF8String defaultNameSpaceURI; /* Identifies the default Namespace associated with
                                    * this record. Those attribute names which are
                                    * unqualified will be assumed to be from this
                                    * namespace
                                    */
};
```

```
        * (e.g. "http://www.ipdr.org/namespace")
        */
    NamespaceInfo otherNameSpaces<>; /* Identifies additional namespaces from which
        * attributes are derived. This list may be empty.
        */

    UTF8String serviceDefinitionURIs<>; /* Identifies the set of service definitions from
        * which the records are produced.
        */

    opaque docId<>; /* The UUID associated with this document */
};
```

5.1.1 Version

The version number is recorded as a 32-bit integer.

The version number associated with this specification shall be 4.

5.1.2 Recorder Info

The recorder info is encoded as a 32-bit length indicator followed by a sequence of bytes of the specified length. Padding with bytes containing zero are used to ensure the set of bytes ends on a 4-byte boundary (in accordance with XDR encoding policy [4]).

The length specifies the number of bytes (not including any padding). The length does not include the 4-bytes which the length indicator itself occupies.

The content of the Recorder Info is not specified in this document. It may be a URI reference or other information which aids in the identification of the element which created the document.

5.1.3 Creation Time

The creation time is a 64-bit integer value representing the time the recording entity began creating this document. The value represents the number of milliseconds since Jan. 1, 1970 0:00 GMT.

5.1.4 Default Namespace

The Default Namespace is encoded as a 32-bit length indicator followed by a sequence of bytes of the specified length. The Namespace, if present, should be in the form of a URI.

This namespace creates an association between unqualified attribute names and the default namespace, as specified in the "Namespaces in XML" W3C Recommendation.

5.1.5 Other Namespaces

The Other Namespaces should enumerate those namespaces used in addition to the Default Namespace in a given instance document.

If all of the attribute references come from a single namespace, then the Default Namespace will suffice and Other Namespaces will have a length of 0.

Namespaces are defined by the various IPDR/Service Definitions which are used for a given document instance. Note that namespaces may be directly referenced by a schema (via the `targetNamespace` directive) or may be used indirectly when one schema imports another schema from a different namespace.

In either case it is important that all namespaces in use for an instance document be identified by the combination of the Default Namespace field and the Other Namespaces field.

5.1.6 Service Definitions

The Service definitions are encoded as 32-bit integer representing the number of UTF-8 strings which represent the service definition URIs. It is recommended that these URI's be accessible to the reader.

Each Service definition is a 32-bit length indicator followed by a sequence of bytes of the specified length.

It is assumed that the service definition URL's point to instances of XML-Schema documents which are further constrained to conform to the guidelines defined by the IPDR Service Definition Guide [2].

These Service Definitions will contain XML element definitions including typing and descriptive information. This information can be used to by a consuming entity to derive additional details about the individual usage attributes.

The usage attributes referenced in the document descriptors should correspond to entity definitions (with the appropriate namespace qualification) appearing in these Service definitions.

Note however if attributes do not appear in a Service definition or no service definitions are present or the service definition URIs are unreachable, the document itself is still able to be processed. There will merely be less explicitly available information about the attributes. And there is a greater risk that documents may refer to attributes using the same name with a different meaning.

5.1.7 Document UUID

Each document produced by a recorder will have a globally unique identifier (UUID) to aid in the tracking of usage information delivery.

A UUID (also known as GUID) is an identifier that is unique across both space and time, with respect to the space of all UUIDs.

UUID's were originally used in the Apollo Network Computing System and later in the Open Software Foundation's (OSF) specification for Distributed Computing Environment (DCE) [7], and then in Microsoft Windows platforms. The OSF is now known as The Open Group.

The UUID shall be encoded as a 32-bit length specifier (which will always have the value 16) followed by the 16-byte UUID sequence.

5.2 Stream Elements

The compact document format contains a header followed by a sequence of Stream Elements. Stream Elements are one of three types:

- **Descriptor** - a descriptor defines the structure of an IPDR Record. The descriptor is composed of a sequence of Attribute Descriptors which define the name and type of each Usage attribute as it will appear in an record.
- **IPDR Record** - a record consists of a reference to a descriptor followed by a sequence of values encoded in the order defined by the descriptor. Each value is encoded following the rules for the Type identified in the descriptor.
- **Document End** - this structure is used to mark the end of a series of stream elements. It also contains a count of IPDR Records and a timestamp indicating when the recorder completed production of this document.

In the XDR description language presented earlier the sequence of Stream elements is defined in the IPDRDoc structure as:

```
IPDRStreamElement  elements<>;
```

XDR [4] specifies the rules for encoding the variable length array of elements as encoding a 32-bit integer indicating the total number of elements followed by the encoded instances of those elements.

The end of the sequence of elements is unambiguously determined by the presence of a Document End element. After decoding this element the sequence is considered complete.

5.2.1 Record Descriptor

The record descriptor defines the structure of one set of usage events which may appear later in this sequence of stream elements.

It is encoded as follows:

```
/* RecordDescriptor
 *
 * A record descriptor defines the set of attributes which are
 * carried in each record of a designated type.  If different
 * subsets of attributes are carried in different records, then
 * a record descriptor is required for each.
 *
 * The record descriptor contains a sequence of attributeDescriptors
 * which identify the name and type of the attributes which are
 * associated with this record type.  The type definition will
 * allow an entity which does not recognize some of the attributes
 * to still successfully parse the document.
 */
struct RecordDescriptor {
    int  descriptorId;          /* The descriptorId is an integer which will allow
                               * individual records in a given document to specify
                               * which descriptor describes their layout.

```

```

        */
        UTF8String  typeName;          /* identifies a record type defined in a service
        * definition.
        * Record types specify the optional and mandatory
        * fields.
        * Note that there may be multiple descriptors
        * associated with the same named type if they
        * use different sets of optional attributes.
        */

        AttributeDescriptor attributes<>; /* Lists the attributes in the order they will appear
        * in records in this document. A record which points
        * to this descriptor's descriptorId, must exactly
        * match the set of attributes identified in this
        * list.
        */
};

```

The union discriminator field is the mechanism used by XDR [4] to distinguish among different instances of a union type. This field is always represented by a 32-bit integer. Stream elements which are Record Descriptors use the value 1, to distinguish them from other stream element types.

- The descriptor ID is a 32-bit integer identifier used to distinguish between other record types which may appear in the same stream. It is recommended to begin issue descriptor ID's beginning with 1 or 0 and assign sequential id's as new descriptors are added to a given document.
- The typeName is a 32-bit length indicator followed by a sequence of bytes of the specified length. It is intended to refer to a ComplexType definition which appears in a service definition. The typeName may be namespace qualified if the type definition is from one of the other namespaces and not the default namespace.
- The attribute descriptor count is encoded as a 32-bit integer. It represents the number of attribute descriptors which are present.

The order of the attribute descriptors indicates the order in which values for this record type will be encoded in IPDR Record data stream elements.

Each attribute descriptor consists of two fields:

```

/* AttributeDescriptor
 *
 * An attribute descriptor defines one attribute which will
 * appear in a record. It contains the attribute name and
 * the attributes type.
 */
struct AttributeDescriptor {

    UTF8String  attributeName; /* The name of an attribute. This name will either
    * come from the default namespace or it will come
    * from an alternate namespace as defined in the
    * header. If an alternate namespace is used, the name
    * will use XML syntax of the form namespaceId:attributeName
    */

    int  typeId; /* Describes the type of this attribute. Explicitly
    * including the type allows us to continue parsing
    * a record even when the attribute name is unrecognized.

```

```

    */
};

```

5.2.2 Attribute Name

The Attribute Name is encoded as a 32-bit length indicator followed by a sequence of bytes of the specified length.

The attribute name is a character string identifying an attribute. The name is formulated according to the conventions defined in the W3C Recommendation, Namespaces in XML. Specifically the presence of the ":" character is used to indicate a non-default namespace. The substring on the left of the ":" should match a prefix defined in the Other Namespaces section of the header. If no ":" is present then the name is considered to be taken from the default namespace. In either case, the name should match an attribute definition in one of the referenced service definitions. Although the presence of the service definitions are not required, if there is no match, then additional information about the attribute will not be directly known to a consuming application.

An application may have additional information about a named attribute through external means not provided in this specification. However, in this situation it should be noted that the self-describing nature of the document is diminished.

5.2.3 Attribute Type

Attribute Type ID is a 32-bit integer that uniquely identifies specific Attribute Type.

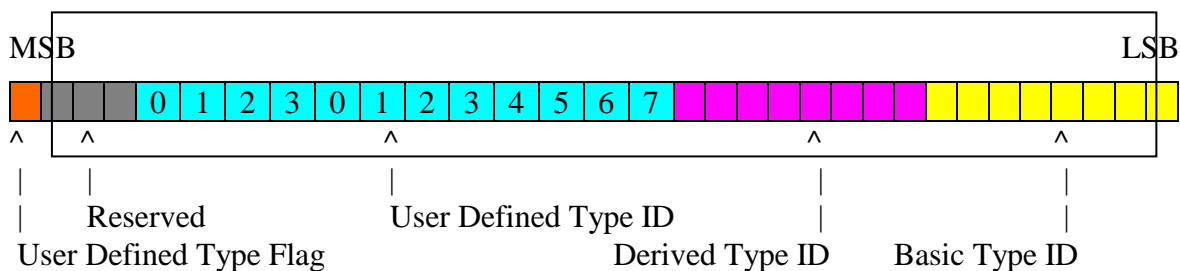


Figure 1: Attribute Type ID - Bit structure

Attribute type is a (non-empty) set of attribute values, and represents a potential for conveying information. While only attribute values are actually conveyed, their attribute type governs the domain of possibilities. The values of such an attribute type, in turn, are the set of permitted contents of that attribute. It is by selecting one particular attribute value of the attribute type, rather than the others, that the sender of an IPDR message (or IPDR file) conveys information.

The attribute type may have only a few attribute values, and therefore be capable of conveying only a few distinctions. An example of such an attribute type is “boolean”, which has only the two attribute values true and false, with nothing in between. On the other hand, some attribute

types, such as “int” and “float”, have an infinite number of attribute values and can thus express arbitrarily fine distinctions.

An attribute type is a **subtype** of another, its parent (attribute type), if its values are a subset of those of the parent. Thus, for example, an attribute type "whole number" whose values are the non-negative integers, is a subtype of “int”. Please note that “whole number” is not an attribute type that is supported by IPDR and only serves here as an example to illustrate this concept.

An IPDR attribute type may be Elementary Type or User Defined Type.

While the IPDR support for Elementary Types is a must, the IPDR support for User Defined Types is a negotiable feature.

5.2.3.1 First Normal Form

The sets of information that can be encoded for a single usage event were required to be in "first normal form" for IPDR 3.0 and 3.1. First normal form is a basic relational database concept that states that the basic properties of tables are that they are made of atomic Elementary units and do not have multiple values for any attribute.

Dividing the information content into different record descriptors and linking the events with some common identifier(s) can still represent usage events that consist of lists of information. Alternatively some simple lists may be carried in string or byte array fields, although this is discouraged, because the self-describing nature of the document, in particular the embedded list, is diminished.

5.2.3.2 Information not in First Normal Form

IPDR's existing service definition model has proven to be well suited to many usage collection scenarios. However, some sets of scenarios, such as multi-party calling, and some existing industry standard formats, such as 3GPP S-CDR's and G-CDR's, require a record which has repeating atomic elements or substructures. XML itself easily supports structures, however, when we defined the IPDR support of Compound Types (as part of User Defined Types - see below) we had to preserve the mapping ability to compact IPDR as well as to maintain backwards compatibility for existing service definitions.

The following sections describe the IPDR model that enables support of both Elementary Types as well as User Defined Types (including Compound Types) in IPDR service definition and IPDR/XML and IPDR/XDR encodings.

Note: that IPDR's current First Normal Form (FNF) restriction should be encouraged for all usage exchange, where it is sufficient. There are key benefits in processing efficiency and database mapping which is erased by more complex structures.

5.2.4 Elementary Types

The Elementary types are the basic building blocks of IPDR, and include types like “boolean” and “int”. An Elementary type will generally be used to describe a single aspect of something.

An IPDR Elementary Type can be either: Basic Type or Derived Type, a Type that is derived from a certain Basic Type. However the basic encoding rules do not differentiate between the Derived Type and the Basic Types it was derived from.

The following tables illustrate the Type IDs for the IPDR Basic Types. The type names are taken directly from XML-Schema Datatypes [5], when available, or have been added as part of the IPDR namespace.

Type	Type ID
int	0x00000021
unsignedInt	0x00000022
long	0x00000023
unsignedLong	0x00000024
float	0x00000025
double	0x00000026
hexBinary or base64Binary ¹	0x00000027
string	0x00000028
boolean	0x00000029
byte	0x0000002a
unsignedByte	0x0000002b
short	0x0000002c
unsignedShort	0x0000002d

The following are the Type IDs for the derived types used in the protocol:

Type	Type ID
dateTime	0x00000122
ipdr:dateTimeMsec	0x00000224
ipdr:ipV4Addr	0x00000322
ipdr:ipV6Addr	0x00000427
ipdr:ipAddr	0x00000827
ipdr:uuid	0x00000527
ipdr:dateTimeUseC	0x00000623
ipdr:macAddress	0x00000723

The typeId is encoded as a 32-bit integer value:

- The high order bit is a flag that is used to indicate User Defined Type.
- The higher of the two low order bytes is used to indicate a derived type, but is not otherwise used by the basic encoding rules.
- Only the low order byte is used to identify the Basic type.

Please note: The low order Elementary Type byte in IPDR/XDR 3.5 uses a different set of type identifiers compared to the IPDR 3.0 and 3.1 specifications. This is used to accommodate the harmonization between the type space of IETF RFC 3243 (CRANE) [2] and the earlier IPDR XDR type space used in version 3.1.1 of IPDR/XDR.

5.2.5 User Defined Types

In contrast to Elementary Types that are built-in IPDR, IPDR allows to define Compound Types through User-Defined Types.

User Defined Type may consist of a combination of one or more Types, which can be used together as a single unit, e.g., a structure, an array or an array of structures.

¹ Please note: base64Binary is a subtype of hexBinary, but they share the same Type ID. This is for backward compatibility with previous versions of the protocol

Users define the hierarchy for the User Defined Types. Nesting is possible.

An IPDR Compound Type is either Array Type or Structured Type:

- A Structured Type is defined in terms of other types - its components - and its values are made up of values of the component types. Each of these components may itself be of Elementary Type or Compound Type, and this nesting can proceed to an arbitrary depth, to suit the needs of the application. All structured types are ultimately defined in terms of Elementary types.
- An Array Type defines a one-dimensional repetition of an attribute of a certain Type (either Elementary Type or Compound Type) within a single record.

The primary purpose of Compound data types is to define containers for other kinds of data (including other compound objects).

Compound Types can be nested, i.e. members of Compound Type can be some other compound Type.

IPDR Support for User Defined Types shall be a negotiable feature

5.2.6 Data Representation of IPDR Types using IPDR XDR encoding

5.2.6.1 Byte Ordering:

A thorough reading of IETF RFC 1832 [4] shows that it utilizes a big-endian byte ordering scheme, that means that bytes are ordered in such a way that most significant bytes precede least significant bytes. RFC 1832 [4] specify that the XDR data byte-order is big-endian

For example, in XDR a 4 byte integer with the value “1” would be represented (in hex octet notation) like this:

```
00 00 00 01
```

Where the first three bytes in memory are 00 and the final byte is a 01. Alternatively, in bit-notation, this would be represented as follows:

Value	00000000	00000000	00000000	00000001
Byte order	0	1	2	3 (bytes offset from starting address)

5.2.6.2 IPDR XDR Basic Types

Here are byte layouts and examples for the thirteen Basic types defined in this document:

int

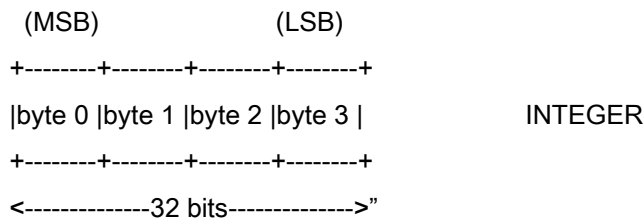
TypeID: 0x00000021

Size in bytes: 4

RFC 1832 [4] has this definition for the int type:

” An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648, 2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. Integers are declared as follows:

int identifier;



Example, in XDR an int with the value “1” would be represented (in hex octet notation) like this:

00 00 00 01

Example, in XDR an int with the value “-2” would be represented (in two’s complement notation) like this:

FF FF FF FE

unsignedInt

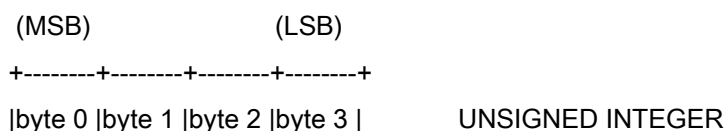
TypeID: 0x00000022

Size in bytes: 4

IETF RFC 1832 [4] has this definition for unsigned int:

“ An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as follows:

unsigned int identifier;



```

+-----+-----+-----+-----+
<-----32 bits----->”

```

Example, in XDR an “unsigned int” with the value “1” would be represented (in hex octet notation) like this:

```
00 00 00 01
```

long and unsigned long

TypeID: 0x000000023 (long)

TypeID: 0x000000024 (unsigned long)

Size in bytes: 8

The “long” and “unsigned long” types are not defined as a type in RFC 1832 [4], however, their analogues are the “hyper” and “unsigned hyper” types. RFC 1832 [4] has this definition for hyper and unsigned hyper:

“ The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations:

hyper identifier; unsigned hyper identifier;

```

(MSB)                                     (LSB)
+-----+-----+-----+-----+-----+-----+-----+-----+
|byte 0 |byte 1 |byte 2 |byte 3 |byte 4 |byte 5 |byte 6 |byte 7 |
+-----+-----+-----+-----+-----+-----+-----+-----+
<-----64 bits----->
HYPER INTEGER
UNSIGNED HYPER INTEGER”

```

Example, in IPDR XDR notation a “long” and an “unsigned long” with the value “1” would be represented (in hex octet notation) like this:

```
00 00 00 00 00 00 00 01
```

float

TypeID: 0x000000025

Size in bytes: 4

RFC 1832 [4] has this definition for the “float” type:

“ The standard defines the floating-point data type "float" (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers. The following three fields describe the single-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.

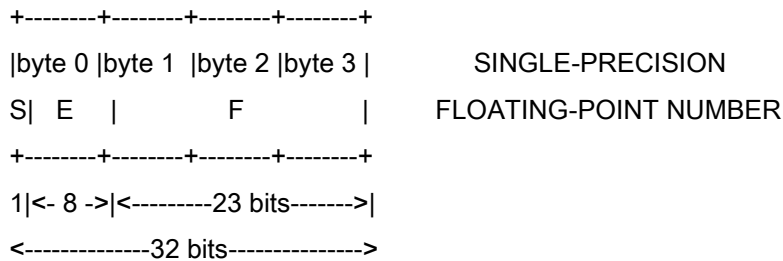
F: The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

Therefore, the floating-point number is described by:

$$(-1)^S * 2^{(E-Bias)} * 1.F$$

It is declared as follows:

float identifier;



Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating-point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) . According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be interpreted within XDR as anything other than "NaN".

Example, in XDR notation, a float with the value “1.0” would be represented as:
3F 80 00 00

In bit notation this would be:

00111111 10000000 00000000 00000000

Showing values for S, E and F:

0 01111111 00000000 00000000 00000000

S E F

S = 0 (a positive number)

E = 127 (127-127 = 0)

F = 0 (because the fractional value is 0)

So the value is 1.0: positive (because S=0), 1 (because 2 to the power of 0 is 1), with 0 as the fractional part of the mantissa.

double

TypeID: 0x00000026

Size in bytes: 8

RFC 1832 [4] has this definition for the “double” type:

“ The standard defines the encoding for the double-precision floating-point data type "double" (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers. The standard encodes the following three fields, which describe the double-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

F: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

$$(-1)^{S} * 2^{(E-Bias)} * 1.F$$

It is declared as follows:

The XML Schema Part 2: Datatypes Second Edition specification [5] contains this definition for `hexBinary`:

“ [Definition:] **hexBinary** represents arbitrary hex-encoded binary data. The value space of **hexBinary** is the set of finite-length sequences of binary octets.

Lexical Representation

hexBinary has a lexical representation where each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code. For example, "0FB7" is a *hex* encoding for the 16-bit integer 4023 (whose binary representation is 111110110111). “

base64Binary

Base64 is a [positional notation](#) using a [base](#) of 64. All well-known variants that are known by the name *Base64* use the printable [ASCII](#) characters A–Z, a–z, and 0–9 in that order for the first 62 digits but the symbols chosen for the last two digits (such as '+' and '/') vary considerably between different systems.

`base64Binary` is a subtype of `hexBinary` (see more about subtype on section 5.3.3 above). For more information on base64 encoding see section 6.8 “Base64 Content-Transfer-Encoding” of RFC 2045 [8].

string

TypeID: 0x00000028

Size in bytes: 4 byte length indicator + number of characters in string

Note that in the IPDR XDR encoding, unlike in the IETF XDR encoding, strings are not padded to 4 byte boundaries.

Here is what RFC 1832 [4] has this to say about the string type (with references to padding to 4 byte boundaries removed):

“ The standard defines a string of *n* (numbered 0 through *n*-1) ASCII bytes to be the number *n* encoded as an unsigned integer (as described above), and followed by the *n* bytes of the string. Byte *m* of the string always precedes byte *m*+1 of the string, and byte 0 of the string always follows the string's length. Counted byte strings are declared as follows:

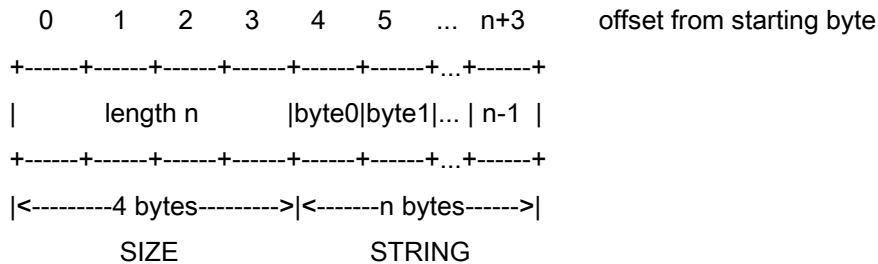
```
string object<m>;
```

or

```
string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be $(2^{32}) - 1$, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

```
string filename<255>;
```



It is an error to encode a length greater than the maximum described in the specification.”

In the IPDR XDR encoding, a string is a UTF8string, which is a string of characters whose length is specified by a 4-byte integer that immediately precedes the string. UTF8string's are constrained to have from 0 to 4294967295 elements. Note that strings are not terminated with a special character, they have the exact length specified by the preceding length indicator.

The string “IPDR organization” would be encoded as:

```
00 00 00 11 49 50 44 52 20 4f 72 67 61 6E 69 7A 61 74 69 6F 6E
length = 17  I P D R   O r g a n i z a t i o n
```

boolean

TypeID: 0x00000029

RFC 1832 [4] has this definition for the Boolean type:

“ Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;"
```

Regarding enum see below section: 5.2.6.4 Enumerated Types, however this is not relevant to Boolean in the IPDR context as is hereby explained.

A boolean in IPDR XDR notation is represented as a 1 byte value (unlike IETF XDR which uses a 4 byte integer) which contains 0 for FALSE and 1 for TRUE.

A boolean with the value of FALSE would be represented in hex octet notation as:
00

A boolean with the value of TRUE would be represented in hex octet notation as:
01

byte

TypeID: 0x0000002a
Size in bytes: 1

An IPDR byte is a signed 8-bit datum that encodes an integer in the range [-128,127]. The byte is represented in two's complement notation. Bytes are declared as follows:

```
byte identifier;
```

```
+-----+
|byte 0| byte
+-----+
<--8 bits-->"
```

Example, in XDR a byte with the value "-1" would be represented (in hex octet notation) like this:

```
FF
```

unsignedbyte

TypeID: 0x0000002b
Size in bytes: 1

An IPDR unsigned byte is an 8-bit datum that encodes a non-negative integer in the range[0,255]. Bytes are declared as follows:

```
byte identifier;
```

```
+-----+
```

```

|byte 0 | byte
+-----+
<--8 bits-->”

```

Example, in XDR an unsigned byte with the value “255” would be represented (in hex octet notation) like this:

FF

short

TypeID: 0x0000002c
Size in bytes: 2

An IPDR short is a 16-bit datum that encodes an integer in the range [-32783, 32782]. The short is represented in two's complement notation. The most and least significant bytes are 0 and 1, respectively. Shorts are declared as follows:

```

short identifier;

(MSB) (LSB)
+-----+-----+
|byte 0 |byte 1 | SHORT
+-----+-----+
<-----16 bits----->”

```

Example, in XDR a short with the value “1” would be represented (in hex octet notation) like this:

00 01

Example, in XDR a short with the value “-2” would be represented (in two's complement notation) like this:

FF FE

unsignedShort

TypeID: 0x0000002d
Size in bytes: 2

An IPDR unsigned short is a 16-bit datum that encodes a non-negative integer in the range [0,65565]. The most and least significant bytes are 0 and 1, respectively. Unsigned shorts are declared as follows:

```

unsigned short identifier;

```

```

(MSB) (LSB)
+-----+-----+
|byte 0 |byte 1 |  UNSIGNED SHORT
+-----+-----+
<-----16 bits----->”

```

Example, in XDR an unsigned short with the value “1” would be represented (in hex octet notation) like this:

```
00 01
```

Example, in XDR a short with the value “256” would be represented (in two’s complement notation) like this:

```
01 00
```

5.2.6.3 IPDR XDR Derived Types

Here are byte layouts and examples for the eight Derived types defined in this document:

dateTime

TypeID: 0x00000122

Derived from: unsignedInt

Size in bytes: 4

The ipdr:dateTime attribute type supports time resolution at the second level.

This type specifies 32 bit value representing UTC time in seconds since EPOCH.

UTC counts time using [SI](#) ([SI prefixes](#) added to the word *second* denote subdivisions of the second such as the [millisecond](#) [seconds](#), and breaks up the span of time into [days](#). UTC days are mostly 86400 s long, but are occasionally 86401 s and could be 86399 s long (though the latter option has never been used [as of December 2006](#)) in order to keep the days synchronized with the rotation of the [Earth](#) (or [Universal Time](#)).

For brevity, the remainder of this section will illustrate the encoding of ipdr:dateTime using UNIX time that increases by exactly 86 400 per day since the EPOCH

This 32 bit counter will suffice (won’t wrap around) at least until the year 2106 (1970+136).

For example, in XDR an “ipdr:dateTime” with the value “2004-09-16T00:00:00Z” (12677 days after the epoch, represented by the Unix time number $12677 \times 86400 = 1095292800$) would be represented (in hex octet notation) like this:

```
41 48 D7 80
```

ipdr:dateTimeMsec

TypeID: 0x00000224
Derived from: unsignedLong
Size in bytes: 8

The ipdr:dateTimeMsec attribute type supports time resolution at the millisecond (one thousandth of a second) level.

This type specifies 64 bit value representing UTC time in milliseconds since EPOCH. UTC counts time using [SI \(SI prefixes\)](#) added to the word *second* denote subdivisions of the second such as the [millisecond](#) [seconds](#), and breaks up the span of time into [days](#). UTC days are mostly 86400 s long, but are occasionally 86401 s and could be 86399 s long (though the latter option has never been used [as of December 2006](#)) in order to keep the days synchronized with the rotation of the [Earth](#) (or [Universal Time](#)).

For brevity, the remainder of this section will illustrate the encoding of ipdr:dateTimeMsec using UNIX time that increases by exactly 86400000 per day since the EPOCH
This 64 bit counter will suffice (won't wrap around) for several million years.

For example, in XDR an "ipdr:dateTimeMsec" with the value "2004-09-16T00:00:00Z" (12677 days after the epoch, represented by the Unix time number $12677 \times 86400 \times 1000 = 1095292800000$) would be represented (in hex octet notation) like this:

```
00 00 00 FF 04 89 CC 00
```

ipdr:ipV4Addr

TypeID: 0x00000322
Derived from: unsignedInt
Size in bytes: 4

IPv4 addresses are 32-bit (4 byte) numbers. An IPv4 address consists of 4 components. Each of the 4 components of the IPv4 address is represented by 1 byte.

For example, in XDR an "ipdr:ipV4Addr" with the value "192.14.6.22" would be represented (in hex octet notation) like this:

```
C0 0E 06 16  
192 14 6 22
```

ipdr:ipV6Addr

TypeID: 0x00000427
Derived from: hexBinary
Size in bytes: 20

IPv6 addresses are 128-bit (16 byte) numbers.

IPv6 are represented using the hexBinary Basic Type where the first 4 bytes indicate the length (16 bytes) and the following 16 bytes of hex data encode the IPv6 address itself.

For example in XDR an “ipdr:ipV6Addr” with the value “1080:0:0:0:8:800:200C:417A” (conventionally expressed using the following text string form: x:x:x:x:x:x:x, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address) would be represented (in hex octet notation) like this:

```
00 00 00 10 10 80 00 00 00 00 00 00 00 08 08 00 20 0C 41 7A
length=16 1080 0 0 0 8 800 200C 417A
```

ipdr:ipAddr

TypeID: 0x00000827

Derived from: hexBinary

Size in bytes: in case of IPv4 - 8
in case of IPv6 – 20

ipdr:ipAddr is either IPv4 address or IPv6 address:

- IPv6 addresses are 128-bit (16 byte) numbers and represented as described above (see: ipdr:ipV6Addr).
- IPv4 addresses are 32-bit (4 byte) numbers and represented as 4 byte indicating the length of 4 + the 4 bytes of hex data encode the IPv4 address itself. For example, in XDR an “ipdr:ipAddr” with the value “192.14.6.22” would be represented (in hex octet notation) like this:

```
00 00 00 04 C0 0E 06 16
length = 4 192 14 6 22
```

ipdr:uuid

TypeID: 0x00000527

Derived from: hexBinary

Size in bytes: 20

UUID is encoded as: 4 byte indicating the length of 16, followed by the 16-byte UUID sequence. For example in XDR an “ipdr:uuid” with the value “6ba7-b810-9dad-11d1-80b4-00c0-4fd4-30c8” would be represented (in hex octet notation) as follows:

```
00 00 00 10 6B A7 B8 10 9D AD 11 D1 80 B4 00 C0 4F D4 30 C8
length=16 6ba7 b810 9dad 11d1 80b4 00c0 4fd4 30c8
```

ipdr:dateTimeUseC

TypeID: 0x00000623

Derived from: long

Size in bytes: 8

The ipdr:dateTimeUseC attribute type supports time resolution at the microsecond (one millionth of a second) level.

This type specifies value representing UTC time in microseconds since EPOCH.

UTC counts time using [SI](#) ([SI prefixes](#) added to the word *second* denote subdivisions of the second such as the microsecond) [seconds](#), and breaks up the span of time into [days](#). UTC days are mostly 86400 s long, but are occasionally 86401 s and could be 86399 s long (though the latter option has never been used [as of December 2006](#)) in order to keep the days synchronized with the rotation of the [Earth](#) (or [Universal Time](#)).

For brevity, the remainder of this section will illustrate the encoding of ipdr:dateTimeMsec using UNIX time that increases by exactly 86400000000 microseconds per day since the EPOCH. This 64 bit counter will suffice (won't wrap around) for over 292,471 years.

For example, in XDR an "ipdr:dateTimeMsec" with the value "2004-09-16T00:00:00Z" (12677 days after the epoch, represented by the Unix time number $12677 \times 86400 \times 1000000 = 1095292800000000$) would be represented (in hex octet notation) like this:

```
00 03 E4 29 BA 44 E0 00
```

ipdr:macAddress

TypeID: 0x00000723

Derived from: long

Size in bytes: 8

MAC addresses are 48-bit (6 byte) numbers and represented as the 48 LSBs of "long" (where the 16 MSBs of that "long" [set to 0](#)).

For example in XDR an "ipdr:macAddress" with the value "00-08-74-4C-7F-1D" would be represented (in hex octet notation) as follows:

```
00 00 00 08 74 4C 7F 1D
```

5.2.6.4 Enumerated Types

RFC 1832 [4] has this definition for the Enumerated type:

" Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, ... } identifier;
```

For example, the three colors red, yellow, and blue could be described by an enumerated type:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration."

An element which is defined to be an enumerated type in the XML schema file could be represented as subtype of either "int", "long", "byte", or "short", and this (parent) type information should be specified in the schema.

5.2.6.5 XML Example

Here is a simple example to demonstrate repeating fields and substructures in a single record. It is loosely based on some of the 3GPP mobile CDR's.

```
<IPDR xsi:type="GatewayExample">
  <subscriber>k123455</subscriber>
  <sourceStationID>station7.region3.foo.com</sourceStationID>
  <qosVolume>
    <volume>100</volume>
    <qosLevel>2</qosLevel>
    <startTime>20020904T13:13:13Z</startTime>
    <endTime>20020904T13:15:13Z</endTime>
  </qosVolume>
  <qosVolume>
    <volume>200</volume>
    <qosLevel>5</qosLevel>
    <startTime>20020904T13:15:14Z</startTime>
    <endTime>20020904T13:19:13Z</endTime>
  </qosVolume>
</IPDR>
```

In this case the element "qosVolume" is a repeating structure. It is repeating, since in the example there are two occurrences of qosVolume. It is structured in that it is not an Elementary type, rather it is made of four components where each component is of Elementary type: volume, qosLevel, startTime and endTime.

5.2.6.6 XDR Example

An XDR equivalent, in hex format is shown.

This assumes that there is a TypeID=0x80010000 associated with the type qosVolumeEvents (User Defined) TypeID=0x80020000 associated with type "GatewayExample".

```
00 00 00 07 6B 31 32 33 34 35 35 00 00 00 18 73 74 61 74 69 6F 6E 37 2E
length = 7 k 1 2 3 4 5 5 length = 24 s t a t i o n 7 .

72 65 67 69 6F 6E 33 2E 66 6F 6F 2E 63 6F 6D 00 00 00 02 00 64 00 02 3D
r e g i o n 3 . f o o . c o m length = 2 100 2

76 06 E9 3D 76 07 61 00 C8 00 05 3D 76 07 62 3D 76 08 51
020904T13:13:13 020904T13:15:13 200 5 020904T13:15:14 020904T13:19:13
```

Breaking this down we see:

```
00 00 00 07 <-- string len
6B 31 32 33 34 35 35 <-- k 1 2 3 4 5 5 subscriber string
0 00 00 24 <-- string len
73 74 61 74 69 6F 6E 37 2E 72 65 67 69 6F 6E 33 2E 66 6F 6F 2E 63 6F 6D <-- s t a t i o n
7 . r e g i o n 3 . f o o . c o m stationid string
00 00 00 02 <-- array length (for qosVolumeEvents)
00 64 <-- volume
```

```

00 02 <-- qosLevel
3D 76 06 E9 <-- 32-bit timestamp in UTC (1031145193 seconds since EPOCH)
3D 76 07 61 <-- 32-bit timestamp in UTC (1031145313 seconds since EPOCH)
00 C8 <-- volume (second qosVolume)
00 05 <-- qosLevel
3D 76 07 62 <-- 32-bit timestamp in UTC (1031145314 seconds since EPOCH)
3D 76 08 51 <-- 32-bit timestamp in UTC (1031145553 seconds since EPOCH)

```

5.2.7 IPDR Record Data

The record data identifies the descriptor that defines the layout of the encoded values followed by the encoded values themselves packed according to the XDR encoding rules for the type associated with each attribute.

```

/* IPDRRecord
 *
 * An ipdrRecord structure represents a single recorded usage
 * event. It is typically flat set of unique attributes. The specific
 * attributes are determined by the a recordDescriptor which is
 * referenced by index value.
 *
 * The set of attribute values are encoded according to XDR rules
 * as if a struct had been defined with elements of the type
 * identified in the recordDescriptor and in the order defined
 * by the recordDescriptor.
 *
 * Situations where repeating elements or substructures are present
 * are described in the IPDR/XDR 3.5 documentation.
 */
struct IPDRRecord {
    int descriptorId; /* The index value of the descriptor which
                     * describes this records attributes. It must
                     * match the descriptor id in a recordDescriptor
                     * encountered earlier.
                     */

    IPDRRecordData data; /* The packed set of attributes */
};

```

- The descriptor id is a 32-bit integer identifier that should match a descriptor id defined previously in this document instance in a Record Descriptor stream element.
- The encoding of the length element diverges from the XDR specification. Ordinarily this would represent the number of bytes that make up the series of encoded values contained in the data. However, because the encoded values may contain variable length byte arrays, and strings that must be UTF-8 encoded, it can be inefficient to calculate the length separately from the process of encoding the values. Therefore the length is represented by the value 0xFFFFFFFF that indicates that the individual values must be decoded according to the specified record descriptor.

The end of the sequence is determined unambiguously upon decoding the last value of the sequence of attributes.

5.3 Document End

The document end identifies the last element in the sequence of stream elements. It also provides summary information about the document. Specifically the number of usage events in the stream and the time when the document was completed are recorded.

```
/* IPDRDocEnd
 *
 * An IPDRDocEnd element is the last element in a sequence of
 * ipdrStreamElement's contained in the doc.
 */
struct IPDRDocEnd {
    int          count;          /* indicates the number of usage elements carried
                                * in this stream of usage events, or -1 if
                                * not used.
                                */

    hyper       endTime;       /* 64-bit time representation, defined to be
                                * the number of milliseconds since
                                * Jan. 1, 1970 0:00 GMT.
                                */
};
```

- The usage event count is encoded as a 32-bit integer value. It should match the number of IPDR Record Data stream elements that appeared in this document.
- The completion time is encoded as a 64-bit integer value representing the number of milliseconds since Jan. 1, 1970 0:00 GMT.

6 References

- [1] IPDR.org v3.5 Document Map Overview.
- [2] IPDR.org Specifications: Service Specification Design Guide 3.5, IPDR Organization, Inc.
- [3] RFC 3423 (CRANE protocol specification).
- [4] XDR: External Data Representation Standard, RFC 1832, August 1995, R. Srinivasan.
- [5] XML-Schema Part 2: Datatypes, W3C, May 2001, <http://www.w3.org/TR/xmlschema-2/>
- [6] UTF-8, a transformation format of ISO 10646, RFC 2279, January 1998, Yergeau, F
- [7] DCE: Remote Procedure Call, Open Group CAE Specification C309, ISBN 1-85912-041-5, August 1994.
- [8] RFC 2045 Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, November 1996