Network Working Group
Request for Comments: 3072
Category: Informational

M. Wildgrube

March 2001

Structured Data Exchange Format (SDXF)

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2001). All Rights Reserved.

IESG Note

This document specifies a data exchange format and, partially, an API that can be used for creating and parsing such a format. The IESG notes that the same problem space can be addressed using formats that the IETF normally uses including ASN.1 and XML. The document reader is strongly encouraged to carefully read section 13 before choosing SDXF over ASN.1 or XML. Further, when storing text in SDXF, the user is encourage to use the datatype for UTF-8, specified in section 2.5.

Abstract

This specification describes an all-purpose interchange format for use as a file format or for net-working. Data is organized in chunks which can be ordered in hierarchical structures. This format is self-describing and CPU-independent.

Table of Contents

1. Introduction

The purpose of the Structured Data eXchange Format (SDXF) is to permit the interchange of an arbitrary structured data block with different kinds of data (numerical, text, bitstrings). Because data is normalized to an abstract computer architecture independent "network format", SDXF is usable as a network interchange data format.

This data format is not limited to any application, the demand for this format is that it is usable as a text format for word-processing, as a picture format, a sound format, for remote procedure calls with complex parameters, suitable for document formats, for interchanging business data, etc.

SDXF is self-describing, every program can unpack every SDXF-data without knowing the meaning of the individual data elements.

Together with the description of the data format a set of functions will be introduced. With the help of these functions one can create and access the data elements of SDXF. The idea is that a programmer should only use these functions instead of maintaining the structure by himself on the level of bits and bytes. (In the speech of object-oriented programming these functions are methods of an object which works as a handle for a given SDXF data block.)

SDXF is not limited to a specific platform, along with a correct preparation of the SDXF functions the SDXF data can be interchanged (via network or data carrier) across the boundaries of different architectures (specified by the character code like ASCII, ANSI or EBCDIC and the byte order for binary data).

SDXF is also prepared to compress and encrypt parts or the whole block of SDXF data.

2. Description of SDXF data format.

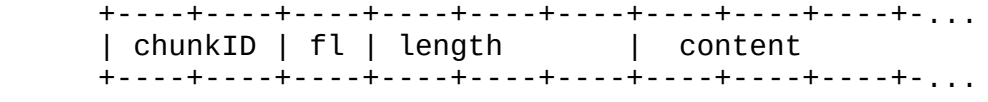
2.1 First we introduce the term "chunk". A chunk is a data structure with a fixed set of components. A chunk may be "elementary" or "structured". The latter one contains itself one or more other chunks.

A chunk consists of a header and the data body (content):

Name	Pos.	Length	Description
chunk-ID	1	2	ID of the chunk (unsigned short)
flags	3	1	type and properties of this chunk
length	4	3	length of the following data
content	7	*)	net data or a list of of chunks

(* as stated in "length". total length of chunk is length+6. The chunk ID is a non-zero positive number.

or more visually:

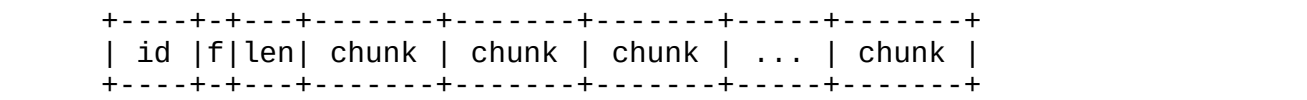


or in ASN.1 syntax:

```
chunk  ::=  SEQUENCE
{
  chunkID INTEGER (1..65535),
  flags   BIT STRING,
  length  OCTET STRING SIZE 3, -- or: INTEGER (0..16777215)
  content OCTET STRING
}
```

2.2 Structured chunk.

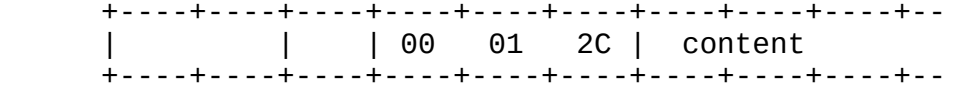
A structured chunk is marked as such by the flag byte (see 2.5). Opposed to an elementary chunk its content consists of a list of chunks (elementary or structured):



With the help of this concept you can reproduce every hierarchically structured data into a SDXF chunk.

2.3 Some Remarks about the internal representation of the chunk's elements:

Binary values are always in high-order-first (big endian) format, like the binary values in the IP header (network format). A length of 300 (=256 + 32 + 12) is stored as

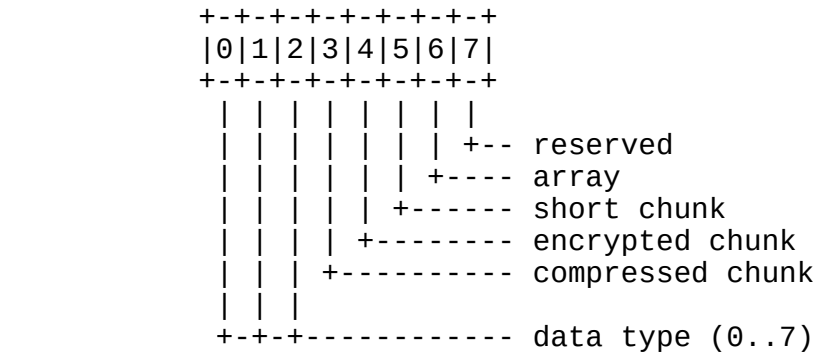


in hexadecimal notation.

This is also valid for the chunk-ID.

2.4 Character values in the content portion are also an object of adaptation: see chapter 4.

2.5 Meaning of the flag-bits: Let us represent the flag byte in this manner:



data types are:

0 -- pending structure (chunk is inconsistent, see also 11.1)
1 -- structure
2 -- bit string
3 -- numeric
4 -- character
5 -- float (ANSI/IEEE 754-1985)

6 -- UTF-8
7 -- reserved

2.6 A short chunk has no data body. The 3 byte Length field is used as data bytes instead. This is used in order to save space when there are many small chunks.

2.7 Compressed and encrypted chunks are explained in chapter 5 and 6.

2.8 Arrays are explained in chapter 7.

2.9 Handling of UTF-8 is explained in chapter 9.

2.10 Not all combinations of bits are allowed or reasonable:

- the flags 'array' and 'short' are mutually exclusive.
- 'short' is not applicable for data type 'structure' and 'float'.
- 'array' is not applicable for data type 'structure'.

3. Introduction to the SDXF functions

3.1 General remarks

The functionality of the SDXF concept is not bounded to any programming language, but of course the functions themselves must be coded in a particular language. I discuss these functions in C and C++, because in the meanwhile these languages are available on almost all platforms.

All these functions for reading and writing SDXF chunks uses only one parameter, a parameter structure. In C++ this parameter structure is part of the "SDXF class" and the SDXF functions are methods of this class.

An exact description of the interface is given in chapter 8.

3.2 Writing a SDXF buffer

For to write SDXF chunks, there are following functions:

```
init    -- initialize the parameter structure
create  -- create a new chunk
leave   -- "close" a structured chunk
```

3.3 Reading a SDXF buffer

For to read SDXF chunks, there are following functions:

```
init    -- initialize the parameter structure
enter   -- "go into" a structured chunk
next    -- "go to" the next chunk inside a structured chunk
extract -- extract the content of an elementary chunk into
           user's data area
leave   -- "go out" off a structured chunk
```

3.4 Example:

3.4.1 Writing:

For demonstration we use a reduced (outlined) C++ Form of these functions with polymorph definitions:

```
void create (short chunkID); // opens a new structure,
void create (short chunkID, char *string);
     // creates a new chunk with dataType character, etc.)
```

The sequence:

```
SDXF x(new); // create the SDXF object "x" for a new chunk
             // includes the "init"
x.create (3301);   // opens a new structure
x.create (3302, "first chunk");
x.create (3303, "second chunk");
x.create (3304);   // opens a new structure
x.create (3305, "chunk in a structure");
x.create (3306, "next chunk in a structure");
x.leave ();        // closes the inner structure
x.create (3307, "third chunk");
x.leave ();        // closes the outer structure
```

creates a chunk which we can show graphically like:

```
3301
 |
 +--- 3302 = "first chunk"
 |
 +--- 3303 = "second chunk"
 |
 +--- 3304
 |      |
 |      +--- 3305 = "chunk in a structure"
 |      |
 |      +--- 3306 = "next chunk in a structure"
 |
 +--- 3307 = "last chunk"
```

3.4.2 Reading

A typically access to a structured SDXF chunk is a selection inside a loop:

```
SDXF x(old); // defines a SDXF object "x" for an old chunk
x.enter ();  // enters the structure

while (x.rc == 0) // 0 == ok, rc will set by the SDXF functions
{
  switch (x.chunkID)
  {
    case 3302:
      x.extract (data1, maxLength1);
                // extr. 1st chunk into data1
      break;

    case 3303:
      x.extract (data2, maxLength2);
                // extr. 2nd chunk into data2
      break;

    case 3304:  // we know this is a structure
      x.enter (); // enters the inner structure

      while (x.rc == 0) // inner loop
      {
        switch (x.chunkID)
        {
          case 3305:
            x.extract (data3, maxLength3);
                      // extr. the chunk inside struct.
```

```
              break;
            case 3306:
              x.extract (data4, maxLength4);
                        // extr. 2nd chunk inside struct.
              break;
          }
          x.next (); // returns x.rc == 1 at end of structure
        } // end-while
        break;

      case 3307:
        x.extract (data5, maxLength5);
                  // extract last chunk into data
        break;
      // default: none - ignore unknown chunks !!!

    } // end-switch
    x.next (); // returns x.rc = 1 at end of structure
  } // end-while
```

4. Platform independence

The very most of the computer platforms today have a 8-Bits-in-a-Byte architecture, which enables data exchange between these platforms. But there are two significant points in which platforms may be different:

a) The representation of binary numerical (the short and long int and floats).

b) The representation of characters (ASCII/ANSI vs. EBCDIC)

Point (a) is the phenomenon of "byte swapping": How is a short int value 259 = 0x0103 = X’0103’ be stored at address 4402?

The two flavours are:

```
4402 4403
01   03    the big-endian, and
03   01    the little-endian.
```

Point (b) is represented by a table of the assignment of the 256 possible values of a Byte to printable or control characters. (In ASCII the letter "A" is assigned to value (or position) 0x41 = 65, in EBCDIC it is 0xC1 = 193.)

The solution of these problems is to normalize the data:

We fix:

(a) The internal representation of binary numerals are 2-complements in big-endian order.

(b) The internal representation of characters is ISO 8859-1 (also known as Latin 1).

The fixing of point (b) should be regarded as a first strike. In some environment 8859-1 seems not to be the best choice, in a greek or russian environment 8859-5 or 8859-7 are appropriate.

Nevertheless, in a specific group (or world) of applications, that is to say all the applications which wants to interchange data with a defined protocol (via networking or diskette or something else), this internal character table must be unique.

So a possibility to define a translation table (and his inversion) should be given.

Important: You construct a SDXF chunk not for a specific addressee, but you adapt your data into a normalized format (or network format).

This adaption is not done by the programmer, it will be done by the create and extract function. An administrator has take care of defining the correct translation tables.

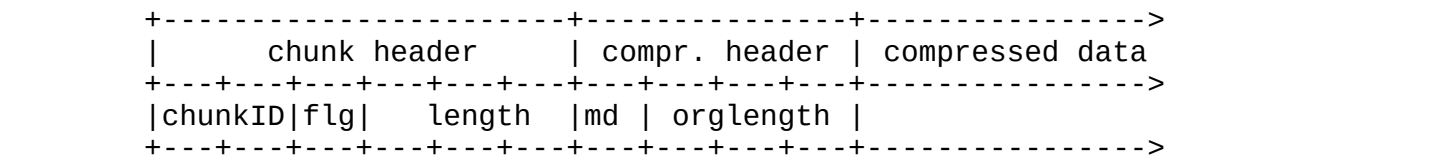
5. Compression

As stated in 2.5 there is a flag bit which declares that the following data (elementary or structured) are compressed. This data is not further interpretable until it is decompressed. Compression is transparently done by the SDXF functions: "create" does the compression for elementary chunks, "leave" for structured chunks, "extract" does the decompression for elementary chunks, "enter" for structured chunks.

Transparently means that the programmer has only to tell the SDXF functions that he want compress the following chunk(s).

For choosing between different compression methods and for controlling the decompressed (original) length, there is an additional definition:

After the chunk header for a compressed chunk, a compression header is following:



- 'orglength' is the original (decompressed) length of the data.

- 'md' is the "compression method": Two methods are described here:

  # method 01 for a simple (fast but not very effective) "Run Length 1" or "Byte Run 1" algorithm. (More then two consecutive identical characters are replaced by the number of these characters and the character itself.)

    more precisely:

    The compressed data consists of several sections of various length. Every section starts with a "counter" byte, a signed "tiny" (8 bit) integer, which contains a length information.

    If this byte contains the value "n",
    with n >= 0 (and n <128), the next n+1 bytes will be taken unchanged;
    with n < 0 (and n > -128), the next byte will be replicated -n+1 times;
    n = -128 will be ignored.

    Appending blanks will be cutted in general. If these are necessary, they can be reconstructed while "extract"ing with the parameter field "filler" (see 8.2.1) set to space character.

  # method 02 for the wonderful "deflate" algorithm which comes from the "zip"-people.
    The authors are:
    Jean-loup Gailly (deflate routine),
    Mark Adler (inflate routine), and others.

    The deflate format is described in [DEFLATE].

  The values for the compression method number are maintained by IANA, see chap. 12.1.

6. Encryption

As stated in 2.5 there is a flag bit which declares that the following data (elementary or structured) is encrypted. This data is not interpretable until it is decrypted. En/Decryption is transparently done by the SDXF functions, "create" does the encryption for elementary chunks, "leave" for structured chunks, "extract" does the decryption for elementary chunks, "enter" for structured chunks. (Yes it sounds very similar to chapter 5.) More then one encryption method for a given range of applications is not very reasonable. Some encryption algorithms work with block ciphering algorithms. That means that the length of the data to encrypt must be rounded up to the next multiple of this block length. This blocksize (zero means non-blocking) is reported by the encryption interface routine (addressed by the option field *encryptProc, see chapter 8.5) with mode=3. If blocking is used, at least one byte is added, the last byte of the lengthening data contains the number of added bytes minus one. With this the decryption interface routine can calculate the real data length.

If an application (or network connect handshaking protocol) needs to negotiate an encryption method it should be used a method number maintained by IANA, see chap. 12.2.

Even the en/decryption is done transparently, an encryption key (password) must be given to the SDXF functions. Encryption is done after translating character data into, decryption is done before translation from the internal ("network-") format.

If both, encryption and compression are applied on the same chunk, compression is done first - compression on good encrypted data (same strings appears as different after encryption) tends to zero compression rates.

7. Arrays

An array is a sequence of chunks with identical chunk-ID, length and data type.

At first a hint: in principle a special definition in SDXF for such an array is not really necessary:

It is not forbidden that there are more than one chunk with equal chunk-ID within the same structured chunk.

Therefore with a sequence of SDX_next / SDX_extract calls one can fill the destination array step by step.

If there are many occurrences of chunks with the same chunk-ID (and a comparative small length), the overhead of the chunk-packages may be significant.

Therefore the array flag is introduced. An array chunk has only one chunk header for the complete sequence of elementary chunks. After the chunk header for an array chunk, an array header is following:

This is a short integer (big endian!) which contains the number of the array elements (CT). Every element has a fixed length (EL), so the chunklength (CL) is CL = EL * CT + 2.

The data elements follows immediately after the array header.

The complete array will be constructed by SDX_create, the complete array will be read by SDX_extract.

The parameter fields (see 8.2.1) ’dataLength’ and ’count’ are used for the SDXF functions ’extract’ and ’create’:

Field ’dataLength’ is the common length of the array elements, ’count’ is the actual dimension of the array for ’create’ (input).

For the ’extract’ function ’count’ acts both as an input and output parameter:

Input : the maximum dimension
output: the actual array dimension.

(If output count is greater than input count, the ’data cutted’ warning will be responded and the destination array is filled up to the maximum dimension.)

8. Description of the SDXF functions

8.1 Introduction

Following the principles of Object Oriented Programming, not only the description of the data is necessary, but also the functions which manipulate data - the "methods".

For the programmer knowing the methods is more important than knowing the data structure, the methods has to know the exact specifications of the data and guarantees the consistence of the data while creating them.

A SDXF object is an instance of a parameter structure which acts as a programming interface. Especially it points to an actual SDXF data chunk, and, while processing on this data, there is a pointer to the actual inner chunk which will be the focus for the next operation.

The benefit of an exact interface description is the same as using for example the standard C library functions: By using standard interfaces your code remains platform independent.

8.2 Basic definitions

8.2.1 The SDXF Parameter structure

All SDXF access functions need only one parameter, a pointer to the SDXF parameter structure:

First 3 prerequisite definitions:

```
typedef short int      ChunkID;
typedef unsigned char  Byte;

typedef struct Chunk
{
  ChunkID    chunkID;
  Byte       flags;
  char       length [3];
  Byte       data;
} Chunk;
```

And now the parameter structure:

```
typedef struct
{
  ChunkID  chunkID;       // name (ID) of Chunk
  Byte    *container;     // pointer to the whole Chunk
  long     bufferSize;    // size of container
  Chunk   *currChunk;     // pointer to actual Chunk
  long     dataLength;    // length of data in Chunk
  long     maxLength;     // max. length of Chunk for SDX_extract
  long     remainingSize; // rem. size in cont. after SDX_create
  long     value;         // for data type numeric
  double   fvalue;        // for data type float
  char    *function;      // name of the executed SDXF function
  Byte    *data;          // pointer to Data
  Byte    *cryptkey;      // pointer to Crypt Key
  short    count;         // (max.) number of elements in an array
  short    dataType;      // Chunk data type / init open type
  short    ec;            // extended return-code
```

```
    short    rc;            // return-code
    short    level;         // level of hierarchy
    char     filler;        // filler char for SDX_extract
    Byte     encrypt;       // Indication if data to encrypt (0 / 1)
    Byte     compression;   // compression method
                            //   (00=none, 01=RL1, 02=zip/deflate)
} SDX_obj, *SDX_handle;
```

Only the "public" fields of the parameter structure which acts as input and output for the SDXF functions is described here. A given implementation may add some "private" fields to this structure.

8.2.2 Basic Functions

All these functions works with a SDX_handle as the only formal parameter. Every function returns as output ec and rc as a report of success. For the values for ec, rc and dataType see chap. 8.4.

1. SDX_init : Initialize the parameter structure.

   input : container, dataType, bufferSize (for dataType = SDX_NEW only)
   output: currChunk, dataLength (for dataType = SDX_OLD only), ec, rc,
   the other fields of the parameter structure will be initialized.

2. SDX_enter : Enter a structured chunk.
   You can access the first chunk inside this structured chunk.
   input : none
   output: currChunk, chunkID, dataLength, level, dataType, ec, rc

3. SDX_leave : Leave the actual entered structured chunk.
   input : none
   output: currChunk, chunkID, dataLength, level, dataType, ec, rc

4. SDX_next : Go to the next chunk inside a structured chunk.
   input : none
   output: currChunk, chunkID, dataLength, dataType, count, ec, rc

   At the end of a structured chunk SDX_next returns rc = SDX_RC_failed and ec = SDX_EC_eoc (end of chunk)
   The actual structured chunk is SDX_leave'd automatically.

5. SDX_extract : Extract data of the actual chunk.
   (If actual chunk is structured, only a copy is done, elsewhere
   the data is converted to host format.)
   input / output depends on the dataType:

   if dataType is structured, binary or char:
     input : data, maxLength, count, filler
     output: dataLength, count, ec, rc

   if dataType is numeric (float resp.):
     input : none
     output: value (fvalue resp.), ec, rc

6. SDX_select : Go to the (next) chunk with a given chunkID.
     input : chunkID
     output: currChunk, dataLength, dataType, ec, rc

7. SDX_create : Creating a new chunk (at the end of the actual
   structured chunk).
     input : chunkID, dataLength, data, (f)value, dataType,
             compression, encrypt, count
     update: remainingSize, level
     output: currChunk, dataLength, ec, rc

8. SDX_append : Append a complete chunk at the end of the actual
   structured chunk).
     input : data, maxLength, currChunk
     update: remainingSize, level
     output: chunkID, chunkLength, maxLength, dataType, ec, rc

8.3 Definitions for C++

This is the specification of the SDXF class in C++: (The type 'Byte' is defined as "unsigned char" for bitstrings, opposed to "signed char" for character strings)

```
class C_SDXF
{
  public:

  // constructors and destructor:
  C_SDXF  ();                          // dummy
  C_SDXF  (Byte *cont);                // old container
  C_SDXF  (Byte *cont, long size);     // new container
  C_SDXF  (long size);                 // new container
  ~C_SDXF ();
  // methods:
```

```
     void init  (void);                   // old container
     void init  (Byte *cont);             // old container
     void init  (Byte *cont, long size);  // new container
     void init  (long size);              // new container

     void enter   (void);
     void leave   (void);
     void next    (void);
     long extract (Byte *data, long length);    // chars, bits
     long extract (void);                       // numeric data
     void create  (ChunkID);                    // structured
     void create  (ChunkID, long value);        // numeric
     void create  (ChunkID, double fvalue);     // float
     void create  (ChunkID, Byte *data, long length);// binary
     void create  (ChunkID, char *data);             // chars
     void set_compression (Byte compression_method);
     void set_encryption  (Byte *encryption_key);

     // interface:

     ChunkID  id;        // see 8.4.1
     short    dataType;  // see 8.4.2
     long     length;    // length of data or chunk

     long     value;
     double   fvalue;
     short    rc;  // the raw return code       see 8.4.3
     short    ec;  // the extended return code  see 8.4.4

     protected:
     // implementation dependent ...

   };
```

8.4 Common Definitions:

8.4.1 Definition of ChunkID:

```
   typedef short ChunkID;
```

8.4.2 Values for dataType:

```
   SDX_DT_inconsistent     = 0
   SDX_DT_structured       = 1
   SDX_DT_binary           = 2
   SDX_DT_numeric          = 3
   SDX_DT_char             = 4
   SDX_DT_float            = 5
```

```
   SDX_DT_UTF8             = 6

    data types for SDX_init:
   SDX_OLD                 = 1
   SDX_NEW                 = 2
```

8.4.3 Values for rc:

```
   SDX_RC_ok               = 0
   SDX_RC_failed           = 1
   SDX_RC_warning          = 1
   SDX_RC_illegalOperation = 2
   SDX_RC_dataError        = 3
   SDX_RC_parameterError   = 4
   SDX_RC_programError     = 5
   SDX_RC_noMemory         = 6
```

8.4.4 Values for ec:

```
   SDX_EC_ok              =  0
   SDX_EC_eoc             =  1 // end of chunk
   SDX_EC_notFound        =  2
   SDX_EC_dataCutted      =  3
   SDX_EC_overflow        =  4
   SDX_EC_wrongInitType   =  5
   SDX_EC_comprerr        =  6 // compression error
   SDX_EC_forbidden       =  7
   SDX_EC_unknown         =  8
   SDX_EC_levelOvflw      =  9
   SDX_EC_paramMissing    = 10
   SDX_EC_magicError      = 11
   SDX_EC_not_consistent  = 12
   SDX_EC_wrongDataType   = 13
   SDX_EC_noMemory        = 14
   SDX_EC_error           = 99 // rc is sufficiently
```

8.5 Special functions

Besides the basic definitions there is a global function (SDX_getOptions) which returns a pointer to a global table of options.

With the help of these options you can adapt the behaviour of SDXF. Especially you can define an alternative pair of translation tables or an alternative function which reads these tables from an external resource (p.e. from disk).

Within this table of options there is also a pointer to the function which is used for encryption / decryption: You can install your own encryption algorithm by setting this pointer.

The options pointer is received by:

```
SDX_TOptions *opt = SDX_getOptions ();
```

With:

```
typedef struct
{
 Byte            *toHost;        // Trans tab net -> host
 Byte            *toNet;         // Trans tab host -> net
 int              maxlevel;      // highest possible level
 int              translation;   // translation net <-> host
                                 // is in effect=1 or not=0
 TEncryptProc    *encryptProc;   // alternate encryption routine
 TGetTablesProc  *getTablesProc; // alternate routine defining
                                 // translation Tables
 TcvtUTF8Proc    *convertUTF8;   // routine to convert to/from UTF-8
}  SDX_TOptions;

typedef long TencryptProc (
  int   mode,   // 1= to encrypt, 2= to decrypt, 3= encrypted length
  Byte *buffer, // data to en/decrypt
  long  len,    // len: length of buffer
  char *passw); // Password

 // returns length of en/de-crypted data
 // (parameter buffer and passw are ignored for mode=3)
 // returns blocksize for mode=3 and len=0.
 // blocksize is zero for non-blocking algorithms

typedef int TGetTablesProc (Byte **toNet, Byte **toHost);
 // toNet, toHost: pointer to output params.  Both params
 // points to translation tables of 256 Bytes.
 // returns success: 1 = ok, 0 = error.

typedef int TcvtUTF8Proc
( int   mode,     // 1 = to UTF-8, 2 = from UTF-8
  Byte *target, int *targetlength,  // output
  Byte *source, int sourcelength);  // input
// targetlength contains maximal size as input param.
// returns success: 1 = ok, 0 = no conversion
```

9. 'Support' of UTF-8.

Many systems supports [UTF-8] as a character format for transferred data. The benefit is that no fixing of a specific character set for an application is needed because the set of 'all' characters is used, represented by the 'Universal Character Set' UCS-2 [UCS], a double byte coding for characters.

SDXF does not really deal with UTF-8 by itself, there are many possibilities to interprete an UTF-8 sequence: The application may:

- reconstruct the UCS-2 sequence,
- accepts only the pure ASCII character and maps non-ASCII to a special 'non-printable' character.
- target is pure ASCII, non-ASCII is replaced in a senseful manner (French accented vowels replaced by vowels without accents, etc.).
- target is a specific ANSI character set, the non-ASCII chars are mapped as possible, other replaced to a 'non-printable'.
- etc.

But SDXF offers an interface for the 'extract' and 'create' functions:

A function pointer may be specified in the options table to maintain this possibility (see 8.5). Default for this pointer is NULL: No further conversions are done by SDXF, the data are copied 'as is', it is treated as a bit string as for data type 'binary'.

If this function is specified, it is used by the 'create' function with the 'toUTF8' mode, and by the 'extract' function with the ' fromUTF8' mode. The invoking of these functions is done by SDXF transparently.

If the function returns zero (no conversion) SDXF copies the data without conversion.

10. Security Considerations

Any corruption of data in the chunk headers denounce the complete SDXF structure.

Any corruption of data in a encrypted or compressed SDXF structure makes this chunk unusable. An integrity check after decryption or decompression should be done by the "enter" function.

While using TCP/IP (more precisely: IP) as a transmission medium we can trust on his CRC check on the transport layer.

11. Some general hints

1. A consistent construction of a SDXF structure is done if every "create" to a structured chunk is closed by a paired "leave". While a structured chunk is under construction, his data type is set to zero - that means: this chunk is inconsistent. The SDX_leave function sets the datatype to "structured".

2. While creating an elementary chunk a platform dependent transformation to a platform independent format of the data is performed - at the end of construction the content of the buffer is ready to transport to another site, without any further translation.

3. As you see no data definition in your programming language is needed for to construct a specific SDXF structure. The data is created dynamically by function calls.

4. With SDXF as a base you can define protocols for client / server applications. These protocols may be extended in downward compatibility manner by following two rules:

   Rule 1: Ignore unknown chunkIDs.

   Rule 2: The sequence of chunks should not be significant.

12. IANA Considerations

The compression and encryption algorithms for SDXF is not fixed, SDXF is open for various algorithms. Therefore an agreement is necessary to interprete the compression and encryption algorithm method numbers. (Encryption methods are not a semantic part of SDXF, but may be used for a connection protocol to negotiate the encryption method to use.)

Following two items are registered by IANA:

12.1 COMPRESSION METHODS FOR SDXF

The compressed SDXF chunk starts with a "compression header". This header contains the compression method as an unsigned 1-Byte integer (1-255). These numbers are assigned by IANA and listed here:

compression method	Description	Hints
01	RUN-LENGTH algorithm	see chap. 5
02	DEFLATE (ZIP)	see [DEFLATE]
03-239	IANA to assign	
240-255	private or application specific	

12.2 ENCRYPTION METHODS FOR SDXF

An unique encryption method is fixed or negotiated by handshaking. For the latter one a number for each encryption method is necessary. These numbers are unsigned 1-Byte integers (1-255). These numbers are assigned by IANA and listed here:

encryption method	Description
01-239	IANA to assign
240-255	private or application specific

12.3 Hints for assigning a number:

Developers which want to register a compression or encrypt method for SDXF should contact IANA for a method number. The ASSIGNED NUMBERS document should be referred to for a current list of METHOD numbers and their corresponding protocols, see [IANA]. The new method SHOULD be a standard published as a RFC or by a established standardization organization (as OSI).

13. Discussion

There are already some standards for Internet data exchanging, IETF prefers ASN.1 and XML therefore. So the reasons for establish a new data format should be discussed.

13.1 SDXF vs. ASN.1

The demand of ASN.1 (see [ASN.1]) is to serve program language independent means to define data structures. The real data format which is used to send the data is not defined by ASN.1 but usually BER or PER (or some derivates of them like CER and DER) are used in this context, see [BER] and [PER].

The idea behind ASN.1 is: On every platform on which a given application is to develop descriptions of the used data structures are available in ASN.1 notation. Out off these notations the real language dependent definitions are generated with the help of an ASN.1-compiler.

This compiler generates also transform functions for these data structures for to pack and unpack to and from the BER (or other) format.

A direct comparison between ASN.1 and SDXF is somehow inappropriate: The data format of SDXF is related rather to BER (and relatives). The use of ASN.1 to define data structures is no contradiction to SDXF, but: SDXF does not require a complete data structure to build the message to send, nor a complete data structure will be generated out off the received message.

The main difference lies in the concept of building and interpretation of the message, I want to name it the "static" and "dynamic" concept:

- ASN.1 uses a "static" approach: The whole data structure must exists before the message can be created.

- SDXF constructs and interpretes the message in a "dynamic" way, the message will be packed and unpacked step by step by SDXF functions.

The use of static structures may be appropriate for a series of applications, but for complex tasks it is often impossible to define the message as a whole. As an example try to define an ASN.1 description for a complex structured text document which is presented in XML: There are sections and paragraphs and text elements which may recursively consist of sections with specific text attributes.

13.2 SDXF vs. XML

On the one hand SDXF and XML are similar as they can handle any recursive complex data stream. The main difference is the kind of data which are to be maintained:

- XML works with pure text data (though it should be noted that the character representation is not standardized by XML). And: a XML document with all his tags is readable by human. Binary data as graphic is not included directly but may be referenced by an external link as in HTML.

In XML there is no strong separation between informational and control data, escape characters (like "<" and "&") and the <![CDATA[...]]> construction are used to distinguish between these two types of data.

- SDXF maintains machine-readable data, it is not designed to be readable by human nor to edit SDXF data with a text editor (even more if compression and encryption is used). With the help of the SDXF functions you have a quick and easy access to every data element. The standard parser for a SDXF data structure follows always a simple template, the "while - switch -case ID - enter/extract" pattern as outlined in chap. 3.4.2.

Because of the complete different philosophy behind XML and SDXF (and even ASN.1) a direct comparison may not be very senseful, as XML has its own right to exist next to ASN.1 (and even SDXF).

Nevertheless there is a chance to convert a XML data stream into a SDXF structure: As a first strike, every XML tag becomes a SDXF chunk ID. An elementary sequence <tag>pure text</tag> can be transformed into an elementary (non-structured) chunk with data type "character". Tags with attributes and sequences with nested tags are transformed into structured chunks. Because XML allows a tag sequence everywhere in a text stream, an artificially "elementary text" tag must be introduced:
If <t> is the tag for text elements, the sequence:

<t>this is a text <attr value='bold'>with</attr> attributes</t>

is to be "in thought" replaced by:

<t><et>this is a text </et><attr value='bold'><et>with</et></attr>
<et> attributes</et></t>

(With "et" as the "elementary text" tag)

This results in following SDXF structure:

```
ID_t
|
+-- ID_et = " this is a text "
|
+-- ID_attr
|   |
|   +-- ID_value = "bold"
|   |
|   +-- ID_et = "with"
|
+-- ID_et = " attributes"
```

ID_t and ID_et may be represented by the same chunk ID, only distinguished by the data type ("structured" for <t> and "character" for <et>)

Binary data as pictures can be directly imbedded into a SDXF structure instead referencing them as an external link like in HTML.

14. Author's Address

Max Wildgrube
Schlossstrasse 120
60486 Frankfurt
Germany

EMail: max@wildgrube.com

15. Acknowledgements

I would like to thank Michael J. Slifcak (mslifcak@iss.net) for the supporting discussions.

16. References

[ASN.1] Information processing systems - Open Systems Interconnection, "Specification of Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, International Standard 8824, December 1987.

[BER] Information Processing Systems - Open Systems Interconnection - "Specification of Basic Encoding Rules for Abstract Notation One (ASN.1)", International Organization for Standardization, International Standard 8825-1, December 1987.

[DEFLATE] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[IANA] Internet Assigned Numbers Authority, http://www.iana.org/numbers.htm

[PER] Information Processing Systems - Open Systems Interconnection -"Specification of Packed Encoding Rules for Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, International Standard 8825-2.

[UCS] ISO/IEC 10646-1:1993. International Standard -- Information technology -- Universal Multiple-Octet Coded Character Set (UCS)

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

17. Full Copyright Statement

Copyright (C) The Internet Society (2001). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.