ASN.1 by simple words

Yury Strozhevsky (http://www.strozhevsky.com)

2012, August

Introduction

Long time I am involved in working with ASN.1. I had been working in cryptography's product development, as well as in product development for telecom. Both of the area are using ASN.1 in their major activities.
But when I'd been working in that areas I heard many times - ASN.1 is a very complicated stuff, let's use professional compilers or even other languages instead of ASN.1. One of the reason for such opinions is that we do not have enough information about ASN.1. Yes, we do not have enough info! ASN.1 is a "grandfather" for almost all modern formats of data transferring, but we still have leak of information about ASN.1 where the format is described by simple words combined with many examples.

In this article I will try to add all descriptions in simplest form I ever heard, plus I will put together all the useful examples I made. The article is only about encoding of simple types - REAL, INTEGER, OBJECT IDENTIFIER, all string types, BOOLEAN, NULL, SEQUENCE and SET. In the article I put detailed descriptions of encoding procedures for each of these types. Moreover, the article combined together with C++ file (for Windows platform) where are all the examples. In the file with examples I also put additional material, not described in the article body.

The article is all about latest ASN.1 standard. The ASN.1 standard and all sub-standards could be downloaded in one file via link http://www.itu.int/rec/T-REC-X.680-X.693-200811-I/en. By default in article I am describing BER encoding (Basic Encoding Rules).

Also I must mention that your any comments are really welcome. Feel free to mail me directly to yury@strozhevsky.com.

In the most books I spend reading material goes from simplest for understanding down to hardiest. In the article I reversed the order - firstly I will discuss about most interesting and complicated types, and then will finish with simplest. This will help on early stage understand all nuances of ASN.1 encoding.

Chapter 1. Common rules for ASN.1 encoding.

But first of all I need to describe basic rules of ASN.1 encoding for all types. Few words about aims of the ASN.1 standard. The standard's main aim is to give ability for communications between heterogeneous systems. In order to achieve the aim ASN.1 describes detailed requirements for all parts of encoding, the standard has requirements even for bit positions. Additionally I must say that initially ASN.1 has a binary format only, nowadays the format has also XML format. The article is dedicated to give a rough overview about all parts of ASN.1 encoding in binary format only.

Encoded data in ASN.1 consists of octet sequence, where the octets go one-by-one without any delimiters. Inside the octet sequence data types is described by special blocks of octets - identification block and length block.

In fact all data types, encoded in ASN.1 BER format, consists of three common blocks:

1. Identification block (unlimited number of octets);
2. Length block (unlimited number of octets);
3. Value block (unlimited number of octets);

Under specific circumstances (using of "indefinite length") there is one additional block - block designating the end of value block (two octets, 00 00). Detailed description of the block I will give later in the chapter.

Let's move to detailed description of all ASN.1 blocks of data.

The identification block consists from 1 octet in minimum. Format of the first octet is:

- Bits 8 and 7 (bigger bits, used to be placed at left side of bit sequence) encode class of tag for encoded type;
- Bit 6 must be set to 0 if value block encodes only one, primitive value (primitive form of encoding) and must be set to 1 if value block encodes inline ASN.1-encoded blocks (constructive form of encoding);
- Bits 5 to 1 encodes tag number for encoded type;

In case of tag number is less or equal to 30 there are no additional octets in identification block. But if tag number is bigger or equal to 31 the bits from 5 to 1 of first octet must be set to 1 and value of tag number will be encoded in a number of octets, goes right after first octet of identification block. The tag number in this case is encoded by indexes from expansion in base 128. In each octet for tag number the biggest bit (bit 8) must be set to 1, expect the latest octet (the method of encoding completely the same with encoding of SID for OBJECT IDENTIFIER, see bellow).

The length block consists of 1 octet in minimum. The encoded length is the count of octets in value block only. The value in first octet for length block can not be bigger than 128. If the value is bigger then such length will be encoded by octet's sequence. So, if encoding length is bigger than 128 then bit 8 in first octet must be set to 1 and remaining 7 octets will encode a number of octets, in which will be encoded the length value.

For example if the length is L = 201 then length block will consist of two octets:

1000 0001 (81)
1100 1001 (C9)

Besides explicit length in ASN.1 we can encode implicit length of value block (indefinite length). In this case first octet in length block must be 80_{256} and the end of value block will be flagged by two octets $(00\ 00)_{256}$.

Chapter 2. Encoding of REAL type.

A theory at the beginning. A numbers with floating point used to be represented by three parts: mantissa, base and exponent. Also its could be represented by the formula: REAL = (mantissa)*(base)$^{(exponent)}$. Because both mantissa and exponent may have negative and positive values the formula may designate both huge and very small values.

In a contract with standard floating point representation (IEEE 754) ASN.1 has no limits for mantissa and exponent value (both that values can be encoded by unlimited number of octets). There are only restrictions on base value, which can be 2, 8, 16 or 10.

In addition to common blocks of encoded value (identification block, length block and value block) there are three specific blocks for encoding REAL values inside common value block:

- Information octet;

- Block of exponent value (unlimited number of octets);
- Block of mantissa value (unlimited number of octets);

Information octet conveys the following info:

- Two biggest bits 8 and 7 in combination convey the following info:
  - If bit 8 = 1 then encoded REAL value is encoded in binary format (base value is one from set 2,8 or 16);
  - If bit 8 = 0 and bit 7 = 0 then encoded REAL value is encoded in decimal format (base value equal to 10);
  - If bit 8 = 0 and bit 7 = 1 then encoded REAL value is a "special value" (NaN, INFINITE etc.);
- If bit 7 = 0 then encoded REAL value is positive value, otherwise - negative;
- Combination of bits 6 and 5 designates value of base:
  - 00 - base is equal to 2;
  - 01 - base is equal to 8;
  - 10 - base is equal to 16;
  - 11 - reserved for future releases of ASN.1 standard;
- Bits 4 and 3 encode the "scaling factor" (F, see bellow) in binary code;
- Bits 2 and 1 encode exponent format of encoded block:
  - 00 - next to the informational octet is the only octet with exponent value;
  - 01 - next two octets encode exponent value;
  - 10 - next three octets encode exponent value;
  - 11 - next octet encodes the value of subsequent octets, in which the exponent value is encoded;

The exponent value is encoded by integer number, there are no restrictions on number of octets for the integer. But the integer may be as positive, as well as a negative. Firstly I need to describe the process of encoding positive and negative integer values.
All integer values in ASN.1 are encoded by indexes before appropriate degrees of 256 in expansion in base 256. For example the expansion for 32639_{10} will be: $32639_{10} = 127*256^1 + 127*256^0$. So, appropriate indexes are 127_{10} and 127_{10}. Its used to be transferred to hexadecimal form, so the indexes will be 7F and 7F. Hence integer value 32639_{10} is encoded by two octets 7F and 7F.

The method above may encodes unlimited positive integers. But how to encode negative?

There are special rules for encoding of negative integers. For example I will encode the same value 32639, but now its will be negative value (-32639). Encoding of negative integers based on encoding not one, but two integers in the same octets. First integer I will name "base integer" and second integer I will name "subtrahend integer". On decoding value of initially encoded integer can be find from formula: x = ("base integer" - "subtrahend integer"). The "x" value will be negative in case when "subtrahend integer" is bigger than "base integer".
The "base integer" and "subtrahend integer" are encoded by the following rules:

- Assume we have ASN.1-encoded integer value in N bits;
- Then "subtrahend integer" is an integer value, is encoded by the N bits, but with all bits set to 0 except the biggest bit (only one bit set to 1);
- The "base integer" also is encoded by N bits, but in "base integer" the biggest bit set to 0 (only biggest bit set to 0, remaining bits are from initially encoded integer);
- As a consequence the "subtrahend integer" always bigger than "base integer" (the biggest index in expansion for "base integer" is equal to 0, and equal to 1 for "subtrahend integer");

Going back to encoding of example value (-32639). As we have seen previously the 32639 value is encoded by two octets (7F 7F). So, "subtrahend value" in this case also will consists of two octets. Accordingly to rule for "subtrahend integer" only biggest bit must be set to 1 and hence "subtrahend integer" is encoded by two octets $(80\ 00)_{256} = 32768_{10}$. The "base integer" could be found from simple formula: "base integer" - 32768 = (-32639). After calculation "base integer" = 129_{10}. This value is encoded by one octet $(81)_{256}$. But accordingly to rule for "base integer" this value must has the same amount of bits with "subtrahend integer". Hence we will encode "base integer" with two bits $(00\ 81)_{256}$. After setting the biggest bit to 1 we got final encoding for (-32639) - $(80\ 81)_{256}$.

Now a pleasant information: in modern computer's environment the integers already are encoded by the rules, described above. Hence there is no need for additionally encoding of negative or positive integers - just copy it octet-by-octet. One note - the "subtrahend integer" in a standard computer's environment used to be too big, so not all the octets of integer should be put in ASN.1 encoding (see bellow the chapter about encoding of INTEGER type).

Mantissa for a REAL value is encoded as a positive integer. The sign for mantissa is encoded by the separate bit in information octet. The rules for encoding integer value for mantissa are completely the same with rules for encoding exponent value (see above).

Examples of encoding REAL values in ASN.1:

For example I will encode value 0.15625. In the fist case I will encode the value with base = 2. An expansion in base 2 will be: $0.15625_{10} = 1*2^{-3} + 1*2^{-5}$. Hence mantissa will be $M = 101_2$ and exponent E = -5. Information octet in this case will be $1000\ 0000_2 = 80_{16}$. Exponent value will be encoded by one octet $(FB)_{256}$. So, now we got all three parts of encoding REAL value 0.15626: $(80\ FB\ 05)_{256}$. As we know the REAL type has UNIVERSAL class and tag number 9. Hence fully encoded ASN.1 REAL value 0.15625 consists of five octets: $(09\ 03\ 80\ FB\ 05)_{256}$.

Now let's encode the same value 0.15625 but in base=8. Expansion in base 8 for the value will be: $0.15625_{10} = 1*8^{-1} + 2*8^{-2}$. Hence mantissa $M = 12_8 = (001\ 010)_2 = 0A_{256}$ (for encoding each number in base =8 we need only 3 bits). Exponent value will be E = -2 which is encoded by one octet FE_{256}. Information octet will be $1001\ 0000_2 = 90_{256}$. So, fully encoded value will be: $(09\ 03\ 90\ FE\ 0A)_{256}$.

In this example I will encode the same value 0.15625, but with base = 16. Expansion for this case will be: $0.15626_{10} = 2*16^{-1} + 8*16^{-2}$. Hence mantissa $M = 28_{16} = (0010\ 1000)_2$, and exponent E = -2, which is encoded by FE_{256} octet. Now put additional requirement: the mantissa value must be "normalized", i.e. must be no zeros at left side of bit sequence (or in other words mantissa must be odd - if lowest bit in expansion is 1 then encoded value is always odd). How can we meet such requirement? Of course by change exponent value. In case of expansion in base 2 everything looks great: by changing exponent on 1 we shift mantissa to left or right (as everyone knows when an integer number multiply by 2 its equal to shifting the encoding for this integer to left side on one bit; when the integer has divided by 2 its equal to shifting to right side on one bit). But in case of expansions in base 8 and 16 changing of exponent on 1 will shift mantissa for 3 and 4 bits (for each number in base 8 we need 3 bits, and in case of base=16 - 4 bits). Hence not in all cases mantissa value for base 8 and 16 can be "normalized" by changing exponent value only. For more "precise tune" the "scaling factor" (F) was introduced. With a help from "scaling factor" we can shift mantissa value to left up to 3 bits. In fact the F value is needed only on step of decoding. During decoding the real mantissa is calculated by

formula: $M = N * 2^F$. So, if mantissa will be multiply by 2^3 then it will be shifted left to 3 bits. The process of "normalization" for our current example will consists of these steps:

1. We have initial mantissa $(0010\ 1000)_2$;
2. In order to "normalize" the value we need to shift it to 3 bits right. Hence we got F = 3 and mantissa $N = 101_2 = 05_{256}$;
3. On decoding we will calculate real value of mantissa: $M = N * 2^3 = (0010\ 1000)_2$;

Hence fully encoded ASN.1 REAL value 0.15625 in base=16 with meet "normalization" requirement will be:

$(09\ 03\ AC\ FE\ 05)_{256}$

All the examples above were about binary encoding of REAL type (encoding in base 2, 8 or 16). But there is one another base - base 10. In fact in this base the REAL type is encoded not as a "floating point number with mantissa end exponent", but as a simple string representation of REAL value.
In decimal encoding (encoding in base 10) there are three "number representation" forms (NR1, NR2 and NR3). All these forms are described in ISO 6093. The standard is not free and instead of ISO 6093 you can use an "ancestor" of the standard - ECMA-63, which can be easily found in Internet.
There are some restrictions in encoding in the decimal form. First group of the restrictions describes requirements to character set for REAL value representation. Valid characters are:

1. Symbols, designating numbers (0-9) (code 30-39 respectively);
2. Space symbol (code 20);
3. "Full stop" symbol - "." (as a "decimal mark") (code 2E);
4. Comma symbol - "," (as a "decimal mark") (code 2C);
5. The "exponent symbol" - "E" (code 45);
6. Symbol "-" (code 2D);
7. Symbol "+" (code 2B);

All other characters are prohibited by standard (during decoding of encoded REAL value with such prohibited symbols all decoders must rise an error).

Examples of encoding for REAL type in decimal form:

Firstly I will encode simple 1 value. In case of encoding in NR1 form this value will be encoded by simple string "1" (or "+1"). In case of using NR2 form this value must be encoded with "decimal mark". Hence all strings bellow can encode 1:

1. "1,"
2. "+1.0"
3. "1,00000"
4. " 1.0" (the beginning of encoded string may consists of unlimited number of space symbols);

Now will represent 1 in NR3 form. In the form 1 must have as well as a "decimal mark", as an exponent value. Accordingly to standard 1 value may be represented in NR3 form by string - "1.E+0", i.e. exponent value for 1 value must always be 0.

Besides common REAL values ASN.1 can encode a "special values":

- PLUS-INFINITY
- MINUS-INFINITY
- NOT-A-NUMBER
- minus zero

All these "special values" are encoded by just one information octet, without octets for mantissa and exponent:

- PLUS-INFINITY - 40_{256}
- MINUS-INFINITY - 41_{256}
- NOT-A-NUMBER - 42_{256}
- minus zero - 43_{256}

Chapter 3. Encoding of OBJECT IDENTIFIER type.

In fact the OBJECT IDENTIFIER consists of set of unsigned integers with delimiter symbol ".". Examples of OBJECT IDENTIFIER: 0.1.1; 1.1.1; 2.1234.1234.1234.1234. Encoding of entire OBJECT IDENTIFIER value consists of encoding for each of the unsigned integers, which OBJECT IDENTIFIER consists of.

Let's give a name to the unsigned integers - sub identifiers, or SID. So, all the SIDs are encoded by the same rules, except first two SIDs. There are specific rules for these specials SIDs:

- The very first SID can be 0, 1 or 2 only. Other values are prohibited. In fact the SID1 is excluded from OBJECT IDENTIFIER encoding, instead the value of SID1 can be deducted from encoding of SID2;
- For the second SID (SID2) ASN.1 has the following restrictions:
    - If SID1 has value 0 or 1 then SID2 must be in range 0-39 (boundaries included);
    - If SID1 has value 2 then SID2 has no limits (of course the SID2 value must be bigger or equal than 0);
- Instead of encoding the initial value of SID2 the ASN.1 encodes a new value accordingly to formula: SID1*40 + SID2. From the result of this formula both SID1 and SID2 can be successfully deducted;
    - If OBJECT IDENTIFIER is "0.39" then the formula will be 0*40 + 39 = 39;
    - If OBJECT IDENTIFIER is "1.0" then the formula will be 1*40 + 0 = 40;
    - If OBJECT IDENTIFIER is "1.39" then the formula will be 1*40 + 39 = 79;
    - If OBJECT IDENTIFIER is "2.0" then the formula will be 2*40 + 0 = 80;
    - If OBJECT IDENTIFIER is "2.39" then the formula will be 2*40 + 39 = 119;
    - If OBJECT IDENTIFIER is "2.339" then the formula will be 2*40 + 339 = 419;

Each SID is encoded separately from others. All SIDs are encoded one-by-one, without any additional breaks and special delimiters. The encoding rules are same for all SIDs, except fisrt two SIDs (see above).

Any SID has (in theory) unlimited value. Hence encoding for each SID may take unlimited number of octets. Usually in ASN.1 add additional block with length value in case of encoding unknown amount of octets. But for SID encoding ASN.1 has other option - the bigger bit in each octet is a kind of "flag", describing is the octet the end octet in sequence or not. Re-phrase the rule: in the octet sequence, encoding one SID, the very first bit in each octet must be set to 1, except the latest octet in the sequence. Hence for encoding of value of SID ASN.1 gives remaining 7 bits. Because of this before encoding we must expand SID in base 128.

Examples of SID encoding:

- SID = $643_{10} = 5*128^1 + 3*128^0 = (\ 05\ 03\ )_{128}$. Accordingly to rules all octets in encoding sequence must have bit 8 set to 1, except the latest octet, got that SID = 643_{10} is encoded by two octets $(\ 85\ 03\ )_{256}$;

- SID = 113549_{10} = $6*128^2 + 119*128^1 + 13*128^0$ = (06 77 0D)$_{128}$. Applying rules with bit 8 and getting SID = 113549_{10} is encoded by three octets (86 F7 0D)$_{256}$;
- SID = 49152_{10} = $3*128^2$. In this expansion in base 128 we got no low degree's indexes (these indexes are equal to 0). For the first view the SID must be encoded by only one octet (03). But in the SID encoding ASN.1 does not store value of exponent for expansion in base 128. Hence the expansion for the sake of encoding must be: ($3*128^2$ + $0*128^1$ + $0*128^0$) = (03 00 00)$_{128}$. After applying of rules with 8 bit got that SID = 113549_{10} is encoded by three octets (83 80 00)$_{256}$;

During the encoding of SID user must use smallest number of octets. For example SID = 643 should not be expanded as ($0*128^2 + 5*128^1 + 3*128^0$) = (00 05 03)$_{256}$ and hence encoded by the octet sequence (80 85 03)$_{256}$. The simplest rule - first octet in the sequence encoding SID should not be 80_{256}.

Chapter 4. Encoding of INTEGER type.

The encoding of INTEGER type was already described in chapter about REAL encoding. But in this chapter we will review the encoding of INTEGER type again, together with additional explanations and examples.

In ASN.1 integer values can be as positive, as well as negative. Each of integer has no value limits, and can be encoded in unlimited number of octets.

Each integer is encoded by sequence of octets. Value of these octets are weight before appropriate degree of 256. In other words integer is encoded by expansion in base 256. For example integer value 8388607 is encoded by the following schema:

- Expand the value in base 256: 8388607 = $127*256^2 + 255*255^1 + 255*256^0$;
- Hence we got following "weight" before appropriate degree of 256: 127, 255 and 255;
- Represent all these weights in hexadecimal form: 7F FF FF;

Encoding of negative integers has its own rules. In order to encode negative integer in ASN.1 encoded not one integer, but two. First integer its let's say "base integer", and second integer is a "subtrahend integer" which is must be subtracted from "base integer". Hence on the step of decoding value must be evaluated from formula ("base integer" - "subtrahend integer"). It is easy to find that the decoding value will be negative if "subtrahend integer" will be bigger than "base integer".

Detailed rules were already described in chapter for encoding of REAL type, please check the chapter for further details.

Examples of encoding integer values:

- Encoded ASN.1 integer has value 80_{256} = $128*256^0$ = 128_{10} = $(1000\ 0000)_2$. Here the "base integer" is defined by lowest 7 bits, which is all zero and hence "base integer" has value 0. The "subtrahend integer" is defined by all masked bits of initial encoded values, except the first one (8th bit). So, the "subtrahend integer" has value $(1000\ 0000)_2$ = 80_{256} = 128_{10}. After applying the formula, described before, we can find that initially encoded integer value was 0-128 = $(-128)_{10}$.
- In order to encode positive value (+128) we need to represent expansion of the value in form $0*256^1 + 128*256^0$, or add leading zero value octet. In this case "base integer" will be 128 and "subtrahend integer" will be 0 (the value encoded in ASN.1 by two octets (00 80)$_{256}$);

- Encode negative integer (-136). The expansion of the value is $136_{10} = 88_{256} = (1000\ 1000)_2$. As we can see the expansion has bit 8 set to 1 already. In this case the "subtrahend integer" must be one octet bigger than initial encoded value. In our case "subtrahend integer" has value $(80\ 00)_{256} = 128*256^1 + 0*256^0 = 32768_{10}$. Lets calculate "base integer" for our example: x - 32768 = -136, and x = 32626 = $127*256^1 + 120*256^0 = (7F\ 78)_{256}$. After setting biggest bit in "base integer" to 1 got final encoding for (-136): $(FF\ 78)_{256}$.
- Encode integer value (-8388607). Expansion for the modulus value will be $8388607_{10} = 127*256^2 + 255*256^1 + 255*256^0 = (7F\ FF\ FF)_{256}$. Because here biggest bit has value 0 then "subtrahend integer" will be $(80\ 00\ 00)_{256} = 128*256^2 + 0*256^1 + 0*256^0 = 8388608_{10}$. Then calculating "base integer": x - 8388608 = -8388607; x = 1. This "base integer" can be expanded as one octet 01_{256}. But keeping in mind that "subtrahend integer" must consists of three octets $(80\ 00\ 00)_{256}$ we need to represent "base integer" also in three octets $(00\ 00\ 01)_{256}$. Hence finally value (-8388607) will be encoded by three octets $(80\ 00\ 01)_{256}$.

In fact we have simple requirement for integer values encoding: biggest 9 bits should not have the same value (all 1 or all 0). If all biggest 9 bits are 0 then biggest octet can be deleted from encoded value. If all biggest 9 bits are 1 then encoded value is a negative value and can be re-encoded with less octets in "subtrahend integer".

For example we will again encode integer value (-128). Lets assume "subtrahend integer" as $(80\ 00)_{256} = 32768_{10}$. The "base integer" in this case will be $x = 32640_{10} = (7F\ 80)_{256}$. After setting the biggest bit to 1 we will find final encoding: $(FF\ 80)_{256}$. Here, as you can see, biggest 9 bits are equal to 1. Previously we have encoded the value with smaller number of octet - $(80)_{256}$. So, biggest octet may be removed from encoding without any problems for decoding steps.

At the end of this chapter I again would like to remind to readers that in modern digital environment we already got all integers encoded in the described format. The difference is that outside of ASN.1 integer numbers always have biggest possible "subtrahend integer". For example if integer value takes 4 octets then integer value (-128) will be encoded as $(FF\ FF\ FF\ 80)_{256}$. Or, in other words, here we will have "subtrahend integer" equal to $(80\ 00\ 00\ 00)_{256} = 2147483648_{10}$.

Chapter 5. Encoding of string types.

ASN.1 has many different string type. Here is the full list of those types:

- NumericString;
- PrintableString;
- TeletextString (equal to T61String);
- VideotexString;
- VisibleString;
- IA5String;
- GraphicString;
- GeneralString;
- UniversalString;
- BMPString;
- UTF8String;

Some of them are deprecated now and not using any more (for example VideotexString). For each string type there is a specific set of characters together with set of control characters.

Most useful string types now are those based on Unicode standard (ISO 10646).

UniversalString type (tag's class UNIVERSAL, tag number 28, encoding form - primitive). In the string type each character is encoded by 4 octets.

BMPString type (tag's class UNIVERSAL, tag number 30, encoding form - primitive). Subset of Unicode character set, each symbol is encoded by 2 octets.

UTF8String type (tag's class UNIVERSAL, tag number 12, encoding form - primitive). Native Unicode, but with additional post-processing, allowing to encode each symbol in varieties number of characters.

Chapter 6. Encoding of date and time types.

In fact all the "date and time types" are simple UTF-8 strings in specific formats, packed in specialized tags. All the formats for each type are described by ISO 8601 standard.

UTCTime type. The simplest date-time type. In the type we can encode only date and UTC (Universal Coordinated Time) time. The format of encoded UTF-8 string may be like YYMMDDHHMMSSZ (where YY is two digits of year (without a century number), MM - value for month, DD - value for day, HH - value for hours (in 24-hours format), MM - value for minutes, SS - value for seconds, Z - special flag), or the format may be like YYMMDDHHMMSS+hhmm, or YYMMDDHHMMSS-hhmm (hh - difference in hourse between UTC and local time, mm - difference in minutes between UTC and local time).

GeneralizedTime type. Extension of UTCTime type, in which all 4 digits of year are encoded, plus added ability to encode fractional values for any time part (hours, minutes or seconds). Hence the format of encoded UTF-8 string may be one of those:

- YYYYMMDDHHMMSSZ;
- YYYYMMDDHHMMSS+hhmm;
- YYYYMMDDHHMMSS-hhmm;
- YYYYMMDDHHMMSS.nn (number of fractional digits is not limited);
- YYYYMMDDHHMM.nn (number of fractional digits is not limited);
- YYYYMMDDHH.nn (number of fractional digits is not limited);

All the type described bellow are types specified in the latest version of X.680 only.

TIME type. Encodes only time part in format HH:MM:SS, or in format HHMMSS.

TIME-OF-DAY type. The same with TIME type, except that format can be HHMMSS only.

DATE type. Encodes only date part in format YYYYMMDD.

DATE-TIME type. Encodes both date and time parts in format YYYYMMDDHHMMSS. If, for instance, SS is equal to 00 then the SS can be omitted.

DURATION type. Encodes duration between two time points in format nnYnnMnnDTnnHnnMnnS.

Chapter 7. Encoding of bit sequences (bit strings).

Under name "bit sequences" I assume two types - BIT STRING and OCTET STRING.

The BIT STRING type is dedicated to be a store for smallest pieces of information - bits. In fact there is no encoding - the value block is equal with "encoded" bits. With one exclusion - in the first octet of value block ASN.1 encodes value of "unused bits". Value of "unused bits" is within range 0-7. The "unused bits" is a lower bits in bit sequence (the bits from right side of the bit string). For example assume we should encode bit sequence 0000 1111 = 0F. If put as "unused bits" value 4 then after decoding value of bit sequence will be 0000 (right four bits will be considered as "unused" and hence deleted from final result).

During encoding of BIT STRING type user can use primitive or constructive form of encoding. In primitive form of encoding BIT STRING's encoded value block contains in first octet value for "unused bits" and then all octets of encoded bit sequence, without a breaks. In constructive form of encoding initial bit sequence is divided on smaller parts, which are encoded each in separate primitive encoded BIT STRINGs. In other words one bit sequence is encoded in many bit sequences. On decoding all these small internal sequences must be joined back to one huge bit sequence. One note: in the constructive form of BIT STRING encoding value of "unused bits" must be set in the latest encoded internal BIT STRING only (please see next example).

For example of constructive encoding for BIT STRING assume we have bit sequence 0B 0B 0F and number of "unused bits" is 4. Now let's divide the initial bit sequence down to 3 smaller bit sequences (first subsequence - 0B, second subsequence - 0B and third subsequence - 0F). When each bit subsequence is encoded as a BIT STRING in primitive form we got the following octets, encoded initial BIT STRING in constructive form:

```
23 0C
      03 02 00 0B
      03 02 00 0B
      03 02 04 0F
```

Here the first octet is an information octet for BIT STRING and constructive form. Second octet is a full length of encoded BIT STRING (12_{10}). Next 4 octets are encoded first subsequence (03 - information octet for BIT STRING type and primitive form of encoding; 02 - full length of value block; 00 - number of "unused bits"; 0B - the encoded subsequence). Next 4 octets are encoded second subsequence (are equal with first subsequence). But next 4 octets, which are encoded third subsequence are more interesting. Interesting part here in that only in the latest encoded subsequence we have valid number of "unused bits" (it is 4 in this example). Its because the "unused bit" are the lowest bits in bit sequence (bits on the right side of bit sequence, or in other words at the end of bit sequence) and that is why only the latest encoded subsequence contains the number of "unused bits". On the decoding process all the bit subsequences must be joined in initially coded bit sequence - 0B 0B 0 (in the latest subsequence 4 latest bits must be removed as considered "unused").

The OCTET STRING type is dedicated to contain simple sequences of octets, nothing else. In other words in the OCTET STRING we may store anything! So, we may encode even a file body (gigabytes or whatever). But with one note: in case of using "indefinite length" for OCTET STRING encoder must inspect encoded octet's stream and in case in the stream we have 00 00 octets divide encoded octet sequence down to two separate subsequences (and use constructive form of encoding of course).

Chapter 8. Encoding of prefixed types.

Sometime is necessary to distinguish two same encoded types, which is encoded on-by-one (for example inside SET type, where order of internal types is unknown). In this situation in ASN.1 user must apply additional "prefixes" to existing types in order to discriminate their from each other. In fact such "prefixes" creates not a simple prefix, but a new ASN.1 local type. Let's describe it in details.

All the standard types has its own class of tag, which is equal to UNIVERSAL. But there are remaining classes of tag - Context-specific, Application and Private. So, in order to discriminate two same standard classes encoded value of standard classes put inside a new type, with class of tag other than UNIVERSAL. Its may be considered as an additional "shell" for standard types.

There are two types of such "shells" - EXPLICIT encoding and IMPLICIT encoding. In first case (EXPLICIT) the value block for new (shell) type consists of fully encoded value for standard type (the new type has constructive form of encoding). In second case (IMPLICIT) the value block for new (shell) type is equal with value block from standard type (only value block is copied from standard type). So, in this case user substitute the standard type with a new one, with new class of tag and tag number. In case of IMPLICIT encoding the new (shell) type must has primitive encoding form.

Now some examples of notation, describing "prefixed types".

Type1 ::= [0] BOOLEAN

This notation describing Type1 which has value of tag's class equal to "Context-specific" $(10)_2$, tag number 0 and constructive form of encoding. The value block for Type1 consists of fully encoded value for BOOLEAN type.

Type2 ::= [PRIVATE 2] IMPLICIT BOOLEAN

This notation describing Type1 which has value of tag's class equal to "PRIVATE" $(11)_2$, tag number 2 and primitive form of encoding. The value block for Type2 consists of value block from encoded BOOLEAN type.

In other words by default in ASN.1 as a tag's class we have "Context-specific" and constructive form of encoding (EXPLICIT). So, the example for Type1 may be re-written in equal form (but the form is not a valid ASN.1 notation, only for example!):

Type1 ::= [CONTEXT 0] EXPLICIT BOOLEAN

By the way, full equivalent of standard type may has this notation (also not a valid ASN.1 notation, only for example!):

BOOLEAN_Eq ::= [UNIVERSAL 1] IMPLICIT BOOLEAN

Chapter 9. Encoding of SEQUENCE type.

The SEQUENCE type attendant to store sequence of internally encoded types. The order of internally encoded types must be always the same (must be well-known for both "transmitter" and "receiver" parts for ASN.1 conversation).

Encoding of the SEQUENCE type always has "constructive form" (see chapter 1 about basic rules of ASN.1 encoding).

Example of encoding for SEQUENCE type. Assume we have two values: one has INTEGER type and value -128, other has REAL type and value 0.15625 (in form of expansion in base 2). As you already know from previous chapters first value encoded by three octets 02 01 80, and second values encoded by another octet sequence 09 03 80 FB 05. Hence SEQUENCE with these two values encoded will have following octets:

30 08 02 01 80 09 03 80 FB 05

Here we have information octet as a first octet of the sequence. In this information octet encoded info about constructive form of encoding and "tag" bellows to SEQUENCE type. Second octet is the octet encoded length of "value block". Next three octets encode first, INTEGER value. Then remaining 5 octets encode REAL value.

Chapter 10. Encoding of SET type.

Encoding of SET type is equal to encoding of SEQUENCE type except that order or internally encoded types can vary. In other words if we will put in SET internally encoded types from chapter of encoding SEQUENCE then we may have two equal variants of encoding:

Variant 1: 31 08 02 01 80 09 03 80 FB 05
Variant 2: 30 08 09 03 80 FB 05 02 01 80

In the second variant encoded integer value placed before encoded real value.

Type SET can encode two (or more) same internal types. On decoding value of the same types can be designated by using of "prefixed types". See chapter 8 about the "prefixed types" encoding and using.

Chapter 11. Encoding of BOOLEAN type.

Type BOOLEAN can convey only two value - TRUE or FALSE. When BOOLEAN convey FALSE then content block consists only from one octet, 00. When BOOLEAN convey TRUE value the in content block consists from one octet which value is anything but zero. So, both of bellow examples encode the same value TRUE:

Example 1: 01 01 01
Example 2: 01 01 FF

Chapter 12. Encoding of NULL type.

Value of NULL type is always the same and then always encoded by same octets, 05 00, where first octet is the information octet for NULL type and second octet is length value, which is zero.