ASN.1 Application

version 1.2

Kenneth Lundin

1997-12-01

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

ASN1 User's Guide

An ASN1 compiler and runtime support functions for Erlang.

1.1 ASN1

Introduction

Features

The ASN1 application contains the features listed below:

- ASN.1 compiler for Erlang.
- Generates code to be used by programs sending and receiving ASN.1 specified data
- Encoding rules supported are both BER and the aligned variant of PER.

Refer to the Release Notes for information of each release of ASN.1.

Overview

ASN.1 (Abstract Syntax Notation 1) defines the abstract syntax of information. The purpose of ASN.1 is to have a platform independent language to express types using a standardized set of rules for the transformation of values of a defined type, into a stream of bytes. This stream of bytes can then be sent on a communication channel set up by the lower layers in the stack of communication protocols e.g. TCP/IP or encapsulated within UDP packets. This way, two different applications written in two completely different programming languages running on different computers with different internal representation of data can exchange instances of structured data types (instead of exchanging bytes or bits). This makes programming faster and easier since no code has to be written to process the transport format of the data.

To write a network application which processes ASN.1 encoded messages, it is prudent and sometimes essential to have a set of off-line development tools such as an ASN.1 compiler which can generate the encode and decode logic for the specific ASN.1 data types. It is also necessary to combine this with some general language-specific runtime support for ASN.1 encoding and decoding.

The ASN.1 compiler must be directed towards a target language or a set of closely related languages. This manual describes a compiler which is directed towards the declarative language Erlang. In order to use this compiler, familiarity with the language Erlang is essential. Therefore, the runtime support for ASN.1 is also closely related to the language Erlang and consist of a number of functions, which the compiler uses. The types in ASN.1 and how to represent values of those types in Erlang are described in this manual.

The following document is structured so that the first part describes how to use ASN.1 compiler, and then there are descriptions of all the primitive and constructed ASN.1 types and their representation in Erlang,

Assumed Knowledge

It is assumed that the reader is familiar with the ASN.1 notation as documented in the standard definition [[X.680]] which is the primary text. However it may also be helpful, but not necessary, to read the standard definitions [[X.681]] [[X.682]] [[X.683]] [[X.690]] [[X.691]].

Knowledge of Erlang programming is also essential and reading the book [[erlbook2]] is recommended. Part 1 of this is available on the web in PDF[1] format.

[1] URL: `http://www.erlang.org/download/erlang-book-part1.pdf`

Getting Started ASN1

A Guiding Example

The following example demonstrates the basic functionality used to run the Erlang ASN.1 compiler.

First, create a file called `People.asn` containing the following:

```
People DEFINITIONS IMPLICIT TAGS ::=

BEGIN
EXPORTS Person;

Person ::= [PRIVATE 19] SEQUENCE {
        name PrintableString,
        location INTEGER {home(0),field(1),roving(2)},
        age INTEGER OPTIONAL }
END
```

This file (`people.asn`) must be compiled before it can be used. The ASN.1 compiler checks that the syntax is correct and that the text represents proper ASN.1 code before generating an abstract syntax tree which is saved in a database. The code-generator then uses the information in the database in order to generate code.

The generated Erlang files will be placed in the current directory or in the directory specified with the `{outdir,Dir}` option. The following shows how the compiler can be called from the Erlang shell:

```
1>asn1ct:compile("People").
Erlang ASN.1 compiling "People.py"
--{generated,"People.asn1db"}--
--{generated,"People.hrl"}--
--{generated,"People.erl"}--
ok
2>
```

The ASN.1 module People is now accepted and fed into the database, the generated Erlang code is compiled using the Erlang compiler and loaded into the Erlang runtime system.

Assume there is a network application which receives instances of the ASN.1 defined type Person, modifies and sends them back again:

```
receive
   {Port,{data,Bytes}} ->
       case asn1rt:decode('People','Person',Bytes) of
           {ok,P} ->
               {ok,Answer} = asn1rt:encode('People','Person',mk_answer(P)),
               Port ! {self(),{command,Answer}};
           {error,Reason} ->
               exit({error,Reason})
       end
    end,
```

In the example above, a series of bytes is received from an external source and the bytes are then decoded into a valid Erlang term. This was achieved with the call `asn1rt:decode('People','Person',Bytes)` which returned an Erlang value of the ASN.1 type `Person`. Then an answer was constructed and encoded using `asn1rt:encode('People','Person',Answer)` which takes an instance of a defined ASN.1 type and transforms it to a (possibly) nested list of bytes according to the BER or PER encoding-rules.
The encoder and the decoder can also be run from the shell. The following dialogue with the shell illustrates how the functions `asn1rt:encode/3` and `asn1rt:decode/3` are used.

```
3> Rockstar = {'Person',"Some Name",roving,50}.
{'Person',"Some Name",roving,50}
4> {ok,Bytes} = asn1rt:encode('People','Person',Rockstar).
{ok,["\363",[17],[19,"\t","Some Name"],[2,[1],2],[2,[1],50]]}
5> FlatBytes= lists:flatten.Bytes)
[243,17,19,9,83,111,109,101,32,78,97,109,101,2,1,2,2,1,50]
6> {ok,Person} = asn1rt:decode('People','Person',FlatBytes).
{ok,{'Person',"Some Name",roving,50}}
7>
```

Notice that the result from `encode` is a nested list which must be flattened before the call to `decode`. The reason for returning a nested list is that it is faster to produce and the flatten operation is performed automatically when the list is sent via the Erlang port mechanism.

The ASN1 Application User Interface

The ASN.1 application provides two separate user interfaces:

- The module `asn1ct` which provides the compile-time functions (including the compiler).
- The module `asn1rt` which provides th run-time functions.

The reason for the division of the interface into compile-time and run-time is that only run-time modules(`asn1rt*`) need to be loaded in an embedded system.

Compile-time functions

The ASN.1 compiler can be invoked directly from the command-line by means of the `erlc` program. This is convenient when compiling many ASN.1 files from the command-line or when using Makefiles. Here are some examples of how the `erlc` command can be used to invoke the ASN.1 compiler:

```
erlc -bper Person.asn
erlc -bber ../Example.asn
erlc -o ../asnfiles -i ../asnfiles -i /usr/local/standards/asn1 Person.asn
```

The useful options for the ASN.1 compiler are:

`-b[per|ber]` Choice of encoding rules, if omitted ber is the default.

`-o OutDirectory` Where to put the generated files, default is the current directory.

`-i IncludeDir` Where to search for `.asn1db` files with info about types and values imported from other modules. This option can be repeated many times if there are several places to search in. The compiler will always search the current directory first.

For a complete description of `erlc` see the erlc reference manual.

The compiler and other compile-time functions can also be invoked from the Erlang shell, below follows a brief description of the primary functions, for a complete description of each function see the reference manual for asn1ct [page 19].

The compiler is invoked by using `asn1ct:compile/1` with default options, or `asn1ct:compile/2` if explicit options are given. Example:

```
asn1ct:compile("H323-MESSAGES.asn1",[per]).
```

The generic encode and decode functions can be invoked like this:

```
asn1ct:encode('H323-MESSAGES','SomeChoiceType',{call,"octetstring"}).
asn1ct:decode('H323-MESSAGES','SomeChoiceType',Bytes).
```

Run-time functions

A brief description of the major functions is given here. For a complete description of each function see the reference manual for asn1rt [page 22].

The generic run-time encode and decode functions can be invoked as below:

```
asn1rt:encode('H323-MESSAGES','SomeChoiceType',{call,"octetstring"}).
asn1rt:decode('H323-MESSAGES','SomeChoiceType',Bytes).
```

Errors

Errors detected at compile time appear on the screen together with a line number indicating where in the source file the error was detected. If no errors are found, an Erlang ASN.1 module will be created.

The run-time encoders and decoders (in the asn1rt module) does execute within a catch and returns `{ok, Data}` or `{error, {asn1, Description}}` where `Description` is an Erlang term describing the error.

The ASN.1 types

This section describes the ASN.1 types including their functionality, purpose and how values are assigned in Erlang.

ASN.1 has both primitive and constructed types:

- Primitive types
  - BOOLEAN
  - INTEGER
  - REAL
  - NULL
  - ENUMERATED
  - BIT STRING
  - OCTET STRING
  - Character Strings

  - OBJECT IDENTIFIER
  - Object Descriptor
  - The TIME types
- Constructed types
  - SEQUENCE
  - SET
  - CHOICE
  - SET OF and SEQUENCE OF
  - ANY (not supported in version 1.0.3)
  - ANY DEFINED BY (not supported in version 1.0.3)
  - EXTERNAL (not supported in version 1.0.3)

The general and preferred Erlang representation of most ASN.1 types is a value without a typename. It is also possible to use a tuple notation with type and value as this {`Typename,Value`}. The representation of SEQUENCE and SET are exceptions to the rule above, and they are represented as Erlang records. Below follows a description of how values of each type can be represented in Erlang.

BOOLEAN

Booleans in ASN.1 express values that can be either TRUE or FALSE. The meanings assigned to TRUE or FALSE is beyond the scope of this text.
In ASN.1 it is possible to have:

```
Operational ::= BOOLEAN
```

Assigning a value to the type Operational in Erlang is possible by using the following Erlang code:

```
Myvar1 = true,
Myvar2 = {'Operational',false}
```

INTEGER

ASN.1 itself specifies indefinitely large integers, and the Erlang systems with versions 4.3 and higher, support very large integers, in practice indefinitely large integers.

The concept of sub-typing can be applied to integers as well as to other ASN.1 types. The details of sub-typing are not explained here, however refer to [X.680]. A variety of syntaxes are allowed when defining a type as an integer:

```
T1 ::= INTEGER
T2 ::= INTEGER (-2..7)
T3 ::= INTEGER (0..MAX)
T4 ::= INTEGER (0<..MAX)
T5 ::= INTEGER (MIN<..-99)
T6 ::= INTEGER {red(0),blue(1),white(2)}
```

Below is an example of Erlang code which assigns values for the above types:

```
T1value = 0,
I2 = {'T2',6},
I3 = {'T6',blue},
I4 = {'T6',0},
T6value = white
```

The Erlang variables above are now bound to valid instances of ASN.1 defined types. This style of value can be passed directly to the encoder for transformation into a series of bytes.

REAL

In this version reals are not implemented. When they are, the following ASN.1 type is used:

```
R1 ::= REAL
```

Can be assigned a value in Erlang as:

```
R1value1 = 2.14,
R1value2 = {256,10,-2},
V1 = {'R1',2.56},
V2 = {'R1',{256,10,-2}}
```

In the last line note that the tuple {256,10,-2} is the real number 2.56 in a special notation, which will encode faster than simply stating the number as 2.56. The arity three tuple is `{Mantissa,Base,Exponent}` i.e. Mantissa * Base^Exponent.

NULL

Null is suitable in cases where supply and recognition of a value is important but the actual value is not.

```
Notype ::= NULL
```

The NULL type can be assigned in Erlang:

```
N1 = 'NULL',
N2 = {'Notype','NULL'},
```

The actual value is the quoted atom 'NULL'.

ENUMERATED

The enumerated type can be used, when the value we wish to describe, may only take one of a set of predefined values.

```
DaysOfTheWeek ::= { sunday(1), monday(2),tuesday(3),wednesday(4),
                    thursday(5),friday(6),saturday(7) }
```

For example to assign a weekday value in Erlang:

```
Day1 = {'DaysOfTheWeek',saturday},
```

The enumerated type is very similar to an integer type, when defined with a set of predefined values. An enumerated type differs from an integer in that it may only have specified values, whereas an integer can also have any other value.

BIT STRING

The BIT STRING type can be used to model information which is made up of arbitrary length series of bits. It is intended to be used for a selection of flags, not for binary files.

```
Bits1 ::= BIT STRING
Bits2 ::= BIT STRING {foo(0),bar(1),gnu(2),gnome(3),punk(14)}
```

There are three different notations available for representation of BIT STRINGs in Erlang and as input to the encode functions.

```
Bits1Val = [0,1,0,1,1],
B1 = {'Bits1',[0,1,0,1,1]},
B2 = {'Bits1',16#1A},
```

Note that `Bits1Val` ,`B1` and `B2` denote the same value.

```
Bits2Val1 = [gnu,punk],
Bits2Val2 = 2#1110,
B3 = {'Bits2',[bar,gnu,gnome]},
B4 = {'Bits2',[0,1,1,1]}
```

The above `Bits2Val2`, `B3` and `B4` also all denote the same value.

`Bits2Val1` is assigned symbolic values. The assignment means that the bits corresponding to `gnu` and `punk` i.e. bits 2 and 14 are set to 1 and the rest set to 0. The symbolic values appear as a list of values. If a named value appears, which is not specified in the type definition, a run-time error will occur.

BIT STRINGS may also be sub-typed with for example a SIZE specification:

```
Bits3 ::= BIT STRING (SIZE((0..31))
```

This means that no bit higher than 31 can ever be set.

OCTET STRING

The OCTET STRING is the simplest of all ASN.1 types The OCTET STRING only moves or transfers e.g. binary files or other unstrucrered information complying to two rules. Firstly, the bytes consist of octets and secondly, do not require encoding.

It is possible to have the following ASN.1 type definitions:

```
O1 ::= OCTET STRING
O2 ::= OCTET STRING (SIZE(28))
```

With the following example assignments in Erlang:

```
O1Val = [17,13,19,20],
O2Val = "must be exactly 28 chars....",
Str1 = {'O1',[0,0,0,1,1,1,255,254]},
Str2 = {'O2',"some twentyeight asciioctets"},
```

Observe that `Str1` is assigned a series of numbers between 0 and 255 i.e. octets. `Str2` is assigned using the string notation.

Character Strings

ASN.1 supports a wide variety of character sets. The main difference between OCTET STRINGS and the Character strings is that OCTET STRINGS have no imposed semantics on the bytes delivered.

However, when using for instance the IA5String (which closely resembles ASCII) the byte 65 (in decimal notation) *means* the character 'A'.

For example, if a defined type is to be a VideotexString and an octet is received with the unsigned integer value X, then the octet should be interpreted as specified in the standard ITU-T T.100,T.101.

The ASN.1 to Erlang compiler will not determine the correct interpretation of each BER (Basic Encoding Rules) string octet value with different Character strings. Interpretation of octets is the responsibility of the application. Therefore, from the BER string point of view, octets appear to be very similar to character strings and are compiled in the same way.

It should be noted that when PER (Packed Encoding Rules) is used, there is a significant difference between OCTET STRINGS and other strings. The constraints, specified for a type, are especially important for PER but are not taken into account for BER.

Please note that *all* the Character strings are supported and it is possible to use the following ASN.1 type definitions:

```
Digs ::= NumericString (SIZE(1..3))
TextFile ::= IA5String (SIZE(0..64000))
```

and the following Erlang assignments:

```
DigsVal = "456",
D = {'Digs',"123"},
TextFileVal = "abc...xyz...",
T = {'TextFile',[88,76,55,44,99,121 .......... a lot of characters here
```

OBJECT IDENTIFIER

The OBJECT IDENTIFIER is used whenever a unique identity is required. An ASN.1 module, a transfer syntax, etc. is identified with an OBJECT IDENTIFIER. Assume the example below:

```
Oid ::= OBJECT IDENTIFIER
```

Therefore, the example below is a valid Erlang instance of the type 'Oid'.

```
OidVal = {1,2,55},
Module = {'Oid',{0,0,1,2,3}}
```

The OBJECT IDENTIFIER value is simply a tuple with the consecutive values which must be integers.

The first value is limited to the values 0, 1 or 2 and the second value must be in the range 0..39 when the first value is 0 or 1.

The OBJECT IDENTIFIER is a very important type and it is widely used within different standards to uniquely identify various objects. In [ROSE2] there is an easy-to-understand description of the usage of OBJECT IDENTIFIER.

Object Descriptor

Values of this type can be assigned a value as an ordinary string i.e.
"This is the value of an Object descriptor"

The TIME types

Two different time types are defined within ASN.1 namely Generalized Time and UTC (Universal Time Coordinated), both are assigned a value as an ordinary string within double quotes i.e. "19820102070533.8".

SEQUENCE

The structured types of ASN.1 are constructed from other types in a manner similar to the concepts of array and struct in C.
A SEQUENCE in ASN.1 is comparable with a struct in C and a record in Erlang. A SEQUENCE may be defined as:

```
Pdu ::= SEQUENCE {
   a INTEGER,
   b REAL,
   c OBJECT IDENTIFIER,
   d NULL }
```

This is a 4-component structure called 'Pdu'. The major format for representation of `SEQUENCE` in Erlang is the record format. For each `SEQUENCE` and `SET` in an ASN.1 module an Erlang record declaration is generated. For `Pdu` above, a record like this is defined:

```
-record('Pdu',{a, b, c, d}).
```

The record declarations for a module M are placed in a separate M.hrl file.

Values can be assigned in Erlang as shown below:

```
MyPdu = #'Pdu'{a=22,b=77.99,c={0,1,2,3,4},d='NULL'}.
```

The decode functions will return a record as result when decoding a SEQUENCE or SET.

SET

The SET type is very much like the SEQUENCE type except that the tags of all components must be distinct and the order of the components is not significant. Hence, it must be possible to distinguish every component in the 'SET' both when encoding and decoding a value of a type defined to be a SET. The tags of all components must be different from each other in order to be easily recognizable.

A SET may be defined as:

```
Pdu2 ::= SET {
    a INTEGER,
    b BOOLEAN,
    c ENUMERATED {on(0),off(1)} }
```

A SET is represented as an Erlang record. For each SEQUENCE and SET in an ASN.1 module an Erlang record declaration is generated. For Pdu2 above a record is defined like this:

```
-record('Pdu2',{a, b, c}).
```

The record declarations for a module M are placed in a separate M.hrl file.

Values can be assigned in Erlang as demonstrated below:

```
V = #'Pdu2'{a=44,b=false,c=off}.
```

The decode functions will return a record as result when decoding a SET.

The difference between SET and SEQUENCE is that the order of the components (in the BER encoded format) is undefined for SET and defined as the lexical order from the ASN.1 definition for SEQUENCE. The ASN.1 compiler for Erlang will always encode a SET in the lexical order. The decode routines can handle SET components encoded in any order but will always return the result as a record. Since all components of the SET must be distinguishable both in the encoding phase as well as the decoding phase the following type is not allowed in a module with EXPLICIT or IMPLICIT as tag-default :

```
Bad ::= SET {i INTEGER,
             j INTEGER }
```

The ASN.1 to Erlang compiler rejects the above type. We shall not explain the concept of tag further here, we refer to [X.680].

The concept of SET is an unusual construct and one cannot think of one single application where the set type is essential. (Imagine if someone "invented" the shuffled array in 'C') People tend to think that 'SET' sounds nicer and more mathematical than 'SEQUENCE' and hence use it when 'SEQUENCE' would have been more appropriate. It is also most inefficient, since every correct implementation of SET must always be prepared to accept the components in any order.

Notes about extensibility for SEQUENCE and SET

When a SEQUENCE or SET contains an extension marker and extension components like this:

```
SExt ::= SEQUENCE {
           a INTEGER,
           ...,
           b BOOLEAN }
```

Then the SEQUENCE is represented in Erlang as a record like this:

```
-record('SExt',{a,b=asn1_NOEXTVALUE}).
```

During decoding the `b` field of the record will get the decoded value of the `b` component if present and otherwise the value `asn1_NOEXTVALUE`.

CHOICE

The CHOICE type is a space saver and is similar to the concept of a 'union' in the C-language. As with the previous SET-type, the tags of all components of a CHOICE need to be distinct. If AUTOMATIC TAGS are defined for the module (which is preferable) the tags can be omitted completely in the ASN.1 specification of a CHOICE.

Assume:

```
T ::= CHOICE {
x [0] REAL,
y [1] INTEGER,
z [2] OBJECT IDENTIFIER }
```

It is then possible to assign values:

```
TVal = {y,17},
Val1 = {'T',{z,{0,1,2}}},
```

A CHOICE value is always represented as the tuple {`ChoiceAlternative, Val`} where `ChoiceAlternative` is an atom denoting the selected choice alternative.

It is also allowed to have a CHOICE type tagged as follow:

```
C ::= [PRIVATE 111] CHOICE {
C1,
C2 }

C1 ::= CHOICE {
         a [0] INTEGER,
         b [1] BOOLEAN }

C2 ::= CHOICE {
         c [2] INTEGER,
         d [3] OCTET STRING }
```

In this case, the top type C appears to have no tags at all in its components, however, both C1 and C2 are also defined as CHOICE types and they have distinct tags among themselves. Hence, the above type C is both legal and allowed.

Extensible CHOICE When a CHOICE contains an extension marker and the decoder detects an unknown alternative of the CHIOCE the value is represented as:

```
{asn1_ExtAlt, BytesForOpenType}
```

Where `BytesForOpenType` is a list of bytes constituting the encoding of the “unknown” CHOICE alternative.

SET OF and SEQUENCE OF

The SET OF and SEQUENCE OF types correspond to the concept of an array found in several programming languages. The Erlang syntax for both of these types is straight forward. For example:

```
Arr1 ::= SET SIZE (5) OF INTEGER (4..9)
Arr2 ::= SEQUENCE OF OCTET STRING
```

We may have the following in Erlang:

```
Arr1Val = [4,5,6,7,8], A1 = {'Arr1',[4,5,6,7,8]},
Arr2Val = ["abc",[14,34,54],"Octets"],
 A2 = {'Arr2',[[1,2,3,4,5],[255,254,253],"cyborg"]}
```

Please note that the definition of the SET OF type implies that the order of the components is undefined, but in practice there is no difference between SET OF and SEQUENCE OF. The ASN.1 compiler for Erlang does not randomize the order of the SET OF components before encoding.

Embedded named types

The structured types previously described may very well have other named types as their components. The general syntax to assign a value to the component C of a named ASN.1 type T in Erlang is the record syntax `#'T'{'C'=Value}`. Where `Value` may be a value of yet another type T2, whose value again is assigned as `{T2,Value2}`.

For example:

```
B ::= SEQUENCE {
        a Arr1,
        b [0] T }
```

The above example can be assigned like this in Erlang:

```
V2 = #'B'{a=[4,5,6,7,8], b={x,7.77}}.
```

Embedded structured types

It is also possible in ASN.1 to have components that are themselves structured types. For example, it is possible to have:

```
Emb ::= SEQUENCE {
    a SEQUENCE OF OCTET STRING,
    b SET {
       a [0] INTEGER,
       b [1] INTEGER DEFAULT 66},
    c CHOICE {
       a INTEGER,
       b FooType } }

FooType ::= [3] VisibleString
```

The following records are generated because of the type `Emb`:

```
-record('Emb',{a, b, c}).
-record('Emb_b',{a, b = asn1_DEFAULT}). % the embedded SET type
```

Values of the `Emb` type can be assigned like this:

```
V = #'Emb'{a=["qqqq",[1,2,255]],
           b = #'Emb_b'{a=99},
           c ={b,"Can you see this"}}.
```

Recursive types

Types may refer to themselves. Suppose:

```
Rec ::= CHOICE {
     nothing [0] NULL,
     something SEQUENCE {
          a INTEGER,
          b OCTET STRING,
          c Rec }}
```

This type is recursive; that is, it refers to itself. This is allowed in ASN.1 and the ASN.1-to-Erlang compiler supports this recursive type. A value for this type is assigned in Erlang as shown below:

```
V = {'Rec',
      {something,#'Rec_something'{a = 77,
                                  b = "some octets here",
                                  c = {'Rec',{nothing,'NULL'}}}}}.
```

ASN.1 Values

Values can be assigned to ASN.1 type within the ASN.1 code itself, as opposed to the actions taken in the previous chapter where a value was assigned to an ASN.1 type in Erlang. The full value syntax of ASN.1 is supported and [X.680] describes in detail how to assign values in ASN.1. Below is a short example:

```
TT ::= SEQUENCE {
   a INTEGER,
   b SET OF OCTET STRING }

tt TT ::= {a 77,b {"kalle" "kula"}}
```

The value defined here could be used in several ways. Firstly, it could be used as the value in some DEFAULT component:

```
SS ::= SET {
    s [0] OBJECT IDENTIFIER,
    val TT DEFAULT tt }
```

It could also be used from inside an Erlang program. If the above ASN.1 code was defined in ASN.1 module `Mod`, then the ASN.1 value `tt` can be reached from Erlang as a function call to `Mod:tt()`.

Macros

Macros is not supported at all in this ASN.1 compiler. The main reason is that MACROS is no longer part of the ASN.1 standard.

Tags

Every built-in ASN.1 type, except CHOICE and ANY have a universal tag. This is a unique number that clearly identifies the type.
It is essential for all users of ASN.1 to understand all the details about tags.
There are four different types of tags.

universal For types whose meaning is the same in all applications. Such as integers, sequences and so on; that is, all the built in types.

application For application specific types for example, the types in X.400 Message handling service have this sort of tag.

private For your own private types.

context This is used to distinguish otherwise indistinguishable types in a specific context. For example, if we have two components of a CHOICE type that are both `INTEGER` values, there is no way for the decoder to decipher which component was actually chosen, since both components will be tagged as `INTEGER`. When this or similar situations occur, one or both of the components should be given a context specific to resolve the ambiguity.

The tag in the case of the 'Apdu' type [PRIVATE 1] is encoded to a sequence of bytes making it possible for a decoder to look at the (initial) bytes that arrive and determine whether the rest of the bytes must be of the type associated with that particular sequence of bytes. This means that each tag must be uniquely associated with *only* one ASN.1 type.

Immediately following the tag is a sequence of bytes informing the decoder of the length of the instance. This is sometimes referred to as TLV (Tag length value) encoding. Hence, the structure of a BER encoded series of bytes is:

Tag	Len	Value

Table 1.1:

Encoding Rules

When the first recommendation on ASN.1 was released 1988 it was accompanied with the Basic Encoding Rules, BER as the only alternative for encoding. BER is a somewhat verbose protocol. It adopts a so-called TLV (type, length, value) approach to encoding in which every element of the encoding carries some type information, some length information and then the value of that element. Where the element is itself structured, then the Value part of the element is itself a series of embedded TLV components, to whatever depth is necessary. In summary, BER is not a compact encoding but is relatively fast and easy to produce.

A more compact encoding is achieved with the Packed Encoding Rules PER which was introduced together with the revised recommendation in 1994. PER takes a rather different approach from that taken by BER. The first difference is that the tag part in the TLV is omitted from the encodings, and any tags in the notation are completely ignored. The potential ambiguities are resolved as follows:

- A CHOICE is encoded by first encoding a choice index which identifies the chosen alternative by it's position in the notation.
- The SET and SEQUENCE is treated in an identical manner as the elements transmitted in order. When a SET or SEQUENCE has OPTIONAL or DEFAULT elements, the encoding of each of the elements is preceded by a bit map to identify which OPTIONAL or DEFAULT elements are present.

A second difference is that PER takes full account of the sub-typing information while BER completely ignores it. PER uses the sub-typing information to for example omit length fields whenever possible.

There are two variants of PER, *aligned* and *unaligned*. In summary, PER results in compact encodings which require much more computation to produce than BER.

ASN1 Reference Manual

Short Summaries

- Erlang Module asn1ct [page 19] – ASN.1 compiler and compile-time support functions
- Erlang Module asn1rt [page 22] – ASN.1 runtime support functions

asn1ct

The following functions are exported:

- `compile(ASN1module) -> ok | {error,Reason}`
  [page 19] Compile an ASN.1 module and generate a corresponding Erlang module with encode/decode functions for each type defined.
- `compile(ASN1module , Options) -> ok | {error,Reason}`
  [page 19] Compile an ASN.1 module and generate a corresponding Erlang module with encode/decode functions for each type defined.
- `encode(Module,{Type,Value})-> {ok,Bytes}| {error,Reason}`
  [page 20] Encodes `Value` of `Type` defined in the ASN.1 module `Module`.
- `encode(Module,Type,Value)-> {ok,Bytes} | {error,Reason}`
  [page 20] Encodes `Value` of `Type` defined in the ASN.1 module `Module`.
- `decode(Module,Type,Bytes) -> {ok,Value}|{error,Reason}`
  [page 20] Decodes `Type` from `Module` from the list of bytes `Bytes`.
- `validate(Module,{Type,Value}) -> ok | {error,Reason}`
  [page 20] Validates that `Value` conforms to `Type` from `Module`.
- `validate(Module,Type,Value) -> ok | {error,Reason}`
  [page 20] Validates that `Value` conforms to `Type` from `Module`.
- `start(IncludePaths) -> ok | already_started`
  [page 20] Starts the ASN.1 database server with `IncludePaths` as data.
- `stop() -> void`
  [page 20] Stops the ASN.1 database server.
- `value(Module ,Type) -> {ok,Value} | {error,Reason}`
  [page 21] Returns an Erlang term corresponding to the ASN.1 type `Type` in module `Module`.
- `test(Module) -> ok | {error,Reason}`
  [page 21] Performs a test of encode and decode the types in `Module`.

- `test(Module,Type) -> ok | {error,Reason}`
  [page 21] Performs a test of encode and decode the types in `Module`.
- `test(Module,Type,Value) -> ok | {error,Reason}`
  [page 21] Performs a test of encode and decode the types in `Module`.

asn1rt

The following functions are exported:

- `encode(Module,{Type,Value})-> {ok,Bytes}| {error,Reason}`
  [page 22] Encodes `Value` of `Type` defined in the ASN.1 module `Module`.
- `encode(Module,Type,Value)-> {ok,Bytes} | {error,Reason}`
  [page 22] Encodes `Value` of `Type` defined in the ASN.1 module `Module`.
- `decode(Module,Type,Bytes) -> {ok,Value}|{error,Reason}`
  [page 22] Decodes `Type` from `Module` from the list of bytes `Bytes`.
- `validate(Module,{Type,Value}) -> ok | {error,Reason}`
  [page 22] Validates that `Value` conforms to `Type` from `Module`.
- `validate(Module,Type,Value) -> ok | {error,Reason}`
  [page 22] Validates that `Value` conforms to `Type` from `Module`.

asn1ct (Module)

The ASN.1 compiler takes an ASN.1 module as input and genarates a corresponding Erlang module which can encode and decode the datatypes specified. There are also some generic functions which can be used in during development of applications which handles ASN.1 data (encoded as BER or PER).

Exports

```
compile(ASN1module) -> ok | {error,Reason}
compile(ASN1module , Options) -> ok | {error,Reason}
```

Types:

- ASN1module = atom() | string()
- Options = [Option]
- Option = ber|per|noobj|{outdir,Dir}|{i,IncludeDir}
- Reason = term()

Compiles the ASN.1 module `ASN1module` and generates an Erlang module `ASN1module.erl` with encode decode functions for the types defined in `ASN1module`. For each ASN.1 value defined in the module an Erlang function which returns the value in Erlang representation is generated.

If `ASN1module` is a filename without extension first `".asn"` is assumed and then `".py"` (to be compatible with the old ASN.1 compiler). Of course `ASN.1module` can be a full pathname (relative or absolute) including filename with (or without) extension.

`Options` is a list with some of the following available options:

`ber | per` The encoding rule to be used. EncodingRule is `ber` or `per`. If this option is omitted `ber` is the default. The `per` option means the aligned variant, the unaligned variant of PER is not supported in this version of the compiler. The generated Erlang module always gets the same name as the ASN.1 module and as a consequnce of this only one encoding rule per ASN.1 module can be used at runtime.

`{i,IncludeDir}` Adds `IncludeDir` to the search-path for `.asn1db` files. The compiler tries to open a `.asn1db` file when a module imports definitions from another ASN.1 module. Several `{i,IncludeDir}` can be given.

`noobj` Do not compile (i.e do not produce object code) the generated `.erl` file. If this option is omitted the generated Erlang module will be compiled.

`{out_dir,Dir}` Specifies the directory `Dir` where all generated files shall be placed. If omitted the files are placed in the current directory.

The compiler generates the following files:

- `ASN1module.hrl` (if any SET or SEQUENCE is defined)
- `ASN1module.erl` the Erlang module with encode, decode and value functions.
- `ASN1module.asn1db` intermediate format used by the compiler when modules IMPORTS definitions from each other.

```
encode(Module,{Type,Value})-> {ok,Bytes}| {error,Reason}
encode(Module,Type,Value)-> {ok,Bytes} | {error,Reason}
```

Types:

- Module = Type = atom()
- Value = term()
- Bytes = [Int] when integer(Int), Int >= 0, Int =< 255
- Reason = term()

Encodes `Value` of `Type` defined in the ASN.1 module `Module`. Returns a list of bytes if successful. To get as fast execution as possible the encode function only performs rudimentary tests that the input `Value` is a correct instance of `Type`. The length of strings is for example not always checked. Returns {`ok,Bytes`} if successful or {`error,Reason`} if an error occured.

```
decode(Module,Type,Bytes) -> {ok,Value}|{error,Reason}
```

Types:

- Module = Type = atom()
- Value = Reason = term()
- Bytes = [Int] when integer(Int), Int >= 0, Int =< 255

Decodes `Type` from `Module` from the list of bytes `Bytes`. Returns {`ok,Value`} if successful.

```
validate(Module,{Type,Value}) -> ok | {error,Reason}
validate(Module,Type,Value) -> ok | {error,Reason}
```

Types:

- Module = Type = atom()
- Value = term()

Validates that `Value` conforms to `Type` from `Module`. *Not implemented in this version of the ASN.1 application.*

```
start(IncludePaths) -> ok | already_started
```

Types:

- IncludePaths = [IncludeDirectory]
- IncludeDirectory = string()

Starts the ASN.1 database server and initiates it with `IncludePaths`. `IncludePaths` is a list of directories where the data base server should search for `.asn1db` files when a new module should be loaded into the database.

```
stop() -> void
```

Stops the ASN.1 database server. The database server is only used by the compile-time functions.

`value(Module ,Type) -> {ok,Value} | {error,Reason}`

Types:

- Module = Type = atom()
- Value = term()
- Reason = term()

Returns an Erlang term which is an example of a valid Erlang representation of a value of the ASN.1 type `Type`. The value is a random value and subsequent calls to this function will for most types return different values.

`test(Module) -> ok | {error,Reason}`

`test(Module,Type) -> ok | {error,Reason}`

`test(Module,Type,Value) -> ok | {error,Reason}`

Performs a test of encode and decode of all types in `Module`. The generated functions are called by this function. This function is useful during test to secure that the generated encode and decode functions and the general runtime support work as expected.
`test/1` iterates over all types in `Module`.
`test/2` tests type `Type` with a random value.
`test/3 tests type <c>Type` with `Value`.
Schematically the following happens for each type in the module.

```
{ok,Value} = asn1ct:value(Module,Type),
{ok,Bytes} = asn1ct:encode(Module,Type,Value),
{ok,Value} = asn1:decode(Module,Type,Bytes).
```

asn1rt (Module)

This module is the interface module for the ASN.1 runtime support functions. To encode and decode ASN.1 types in runtime the functions in this module should be used.

Exports

`encode(Module,{Type,Value})-> {ok,Bytes}| {error,Reason}`
`encode(Module,Type,Value)-> {ok,Bytes} | {error,Reason}`

Types:

- Module = Type = atom()
- Value = term()
- Bytes = [Int] when integer(Int), Int >= 0, Int =< 255
- Reason = term()

Encodes `Value` of `Type` defined in the ASN.1 module `Module`. Returns a list of bytes if successful. To get as fast execution as possible the encode function only performs rudimentary tests that the input `Value` is a correct instance of `Type`. The length of strings is for example not always checked.

`decode(Module,Type,Bytes) -> {ok,Value}|{error,Reason}`

Types:

- Module = Type = atom()
- Value = Reason = term()
- Bytes = [Int] when integer(Int), Int >= 0, Int =< 255

Decodes `Type` from `Module` from the list of bytes `Bytes`. Returns `{ok,Value}` if successful.

`validate(Module,{Type,Value}) -> ok | {error,Reason}`
`validate(Module,Type,Value) -> ok | {error,Reason}`

Types:

- Module = Type = atom()
- Value = term()

Validates that `Value` conforms to `Type` from `Module`. *Not implemented in this version of the ASN.1 application.*

List of Tables

Index

Modules are typed in *this way*.
Functions are typed in `this way`.