

# **ASN.1 Application**

**version 1.3**

Typeset in L<sup>A</sup>T<sub>E</sub>X from SGML source using the DOCBUILDER 3.2.2 Document System.

# Contents

<b>1</b>	<b>Asn1 User's Guide</b>	<b>1</b>
1.1	Asn1 . . . . .	1
1.1.1	Introduction . . . . .	1
1.1.2	Getting Started with Asn1 . . . . .	2
1.1.3	The Asn1 Application User Interface . . . . .	4
1.1.4	The ASN.1 Types . . . . .	5
1.1.5	ASN.1 Values . . . . .	18
1.1.6	Macros . . . . .	18
1.1.7	ASN.1 Information Objects (X.681) . . . . .	19
1.1.8	Parameterization (X.683) . . . . .	20
1.1.9	Tags . . . . .	21
1.1.10	Encoding Rules . . . . .	21
<b>2</b>	<b>Asn1 Reference Manual</b>	<b>23</b>
2.1	asn1ct . . . . .	25
2.2	asn1rt . . . . .	29
	<b>List of Tables</b>	<b>31</b>
	<b>Bibliography</b>	<b>33</b>
	<b>Index of Modules and Functions</b>	<b>35</b>



# Chapter 1

## Asn1 User's Guide

The *Asn1* application contains modules with compile-time and run-time support for ASN.1.

### 1.1 Asn1

#### 1.1.1 Introduction

##### Features

The *Asn1* application provides:

- An ASN.1 compiler for Erlang, which generates encode and decode functions to be used by Erlang programs sending and receiving ASN.1 specified data.
- Run-time functions used by the generated code.
- Encoding rules supported are *BER*, the specialized BER version *DER* and the aligned variant of *PER*.

##### 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 functional 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.

### Prerequisites

It is assumed that the reader is familiar with the ASN.1 notation as documented in the standard definition [ITU-T X.680 [1]] which is the primary text. It may also be helpful, but not necessary, to read the standard definitions [ITU-T X.681 [2]] [ITU-T X.682 [3]] [ITU-T X.683 [4]] [ITU-T X.690 [5]] [ITU-T X.691 [6]].

A very good book explaining those reference texts is [ASN.1 Communication between Heterogeneous Systems [7]], free to download at ASN.1 book download <sup>1</sup>.

Knowledge of Erlang programming is also essential and reading the book *Concurrent Programming in ERLANG*, [Concurrent Programming in ERLANG [8]], is recommended. Part 1 of this is available on the web in PDF<sup>2</sup> format.

### 1.1.2 Getting Started with Asn1

#### A First 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:

---

<sup>1</sup>URL: <http://www.oss.com/asn1/dubuisson.html>

<sup>2</sup>URL: <http://www.erlang.org/download/erlang-book-part1.pdf>

```

1>asn1ct:compile("People").
Erlang ASN.1 compiling "People.asn"
--{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. Now there is a user interface of `encode/2` and `decode/2` in the module `People`, which is invoked by:

```

'People':encode(<Type name>,<Value>),
or

```

```

'People':decode(<Type name>,<Value>),

```

Alternatively one can use the `asn1rt:encode(<Module name> ,<Type name>,<Value>)` and `asn1rt:decode(<Module name>,<Type name>,<Value>)` calls. However, they are not as efficient as the previous methods since they result in an additional `apply/3` call.

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 'People':decode('Person',Bytes) of
      {ok,P} ->
        {ok,Answer} = 'People':encode('Person',mk_answer(P)),
        Port ! {self(),{command,Answer}};
      {error,Reason} ->
        exit({error,Reason})
    end
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 `'People':decode('Person',Bytes)` which returned an Erlang value of the ASN.1 type `Person`. Then an answer was constructed and encoded using `'People':encode('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.

### 1.1.3 The Asn1 Application User Interface

The Asn1 application provides two separate user interfaces:

- The module `asn1ct` which provides the compile-time functions (including the compiler).
- The module `asn1rt` which provides the run-time functions. However, it is preferable to use the generated `encode/2` and `decode/2` functions in each module, eg. `'Module':encode('Type',Value)`, in favor of the `asn1rt` interface.

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[ber|per|ber_bin|per_bin]` Choice of encoding rules, if omitted `ber` is the default. The `ber_bin` and `per_bin` options implies that the encoding/decoding functions use binaries and the bit syntax, which in most cases gives significant increase of performance. We really recommend that you use these options instead of `ber` and `per` since the use of binaries will be the default in forthcoming versions.
- `-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.
- `+compact_bit_string` Gives the user the option to use a compact format of the BIT STRING type to save memory space, typing space and increase encode/decode performance, for details see BIT STRING [page 8]type section.
- `+der` DER encoding rule. Only when using `-ber` or `-ber_bin` option.

For a complete description of `erlc` see Erts 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 Asn1 Reference Manual [page 25], the `asn1ct` module.

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).
```



Or, preferable like:

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

### Run-time Functions

A brief description of the major functions is given here. For a complete description of each function see the Asn1 Reference Manual [page 29], the `asn1rt` module.

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).
```

Or, preferable like:

```
'H323-MESSAGES':encode('SomeChoiceType',{call,"octetstring"}).  
'H323-MESSAGES':decode('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) do execute within a catch and returns `{ok, Data}` or `{error, {asn1, Description}}` where `Description` is an Erlang term describing the error.

#### 1.1.4 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 [page 6]
  - INTEGER [page 7]
  - REAL [page 7]
  - NULL [page 8]
  - ENUMERATED [page 8]
  - BIT STRING [page 8]
  - OCTET STRING [page 10]
  - Character Strings [page 10]
  - OBJECT IDENTIFIER [page 11]
  - Object Descriptor [page 11]
  - The TIME types [page 12]
- Constructed types

- SEQUENCE [page 12]
- SET [page 14]
- CHOICE [page 15]
- SET OF and SEQUENCE OF [page 16]
- 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)

**Note:**

The general and preferred Erlang representation of most ASN.1 types is a value without a type name.

However, it is also possible to use a tuple notation with type and value as this {Typename, Value}.

**Warning:**

The tuple notation is only kept because of backward compatibility and may be withdrawn in a future release. If the notation is used the Typename element must be spelled correctly, otherwise a run-time error will occur.

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,
```

Thus, in Erlang the atoms `true` and `false` are used to encode a boolean value.

## 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, bits1val refer to [ITU-T X.680 [1]]. 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)}
```

In Erlang one has to assign an integer value to an INTEGER type, alternatively one can use an atom if it exists in the NamedNumberList of the type definition.

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

```
T1value = 0,
T2value = 6,
T6value1 = blue,
T6value2 = 0,
T6value3 = 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},
```

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<sup>Exponent</sup>.

### 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',
```

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

For example to assign a weekday value in Erlang use the same atom as in the Enumerations of the type definition:

```
Day1 = 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.

In ASN.1 BIT STRING definitions may look like:

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

There are four different notations available for representation of BIT STRING values in Erlang and as input to the encode functions.

1. A list of binary digits (0 or 1).
2. A hexadecimal number (or an integer). This format should be avoided, since it is easy to misinterpret a BIT STRING value in this format. This format may be withdrawn in a future release.
3. A list of atoms corresponding to atoms in the NamedBitList in the BIT STRING definition.

4. As {Unused, Binary} where Unused denotes how many trailing zero-bits 0 to 7 that are unused in the least significant byte in Binary. This notation is only available when the ASN.1 files have been compiled with the *+compact\_bit\_string* flag in the option list. In this case it is possible to use all kinds of notation when encoding. But the result when decoding is always in the compact form. The benefit from this notation is a more compact notation when one has large BIT STRINGs. The encode/decode performance is also much better in the case of large BIT STRINGs.

**Note:**

Note that it is advised not to use the integer format of a BIT STRING, see the second point above.

```
Bits1Val1 = [0,1,0,1,1],
Bits1Val2 = 16#1A,
Bits1Val3 = {3,<<0:1,1:1,0:1,1:1,1:1,0:3>>>}
```

Note that Bits1Val1, Bits1Val2 and Bits1Val3 denote the same value.

```
Bits2Val1 = [gnu,punk],
Bits2Val2 = 2#1110,
Bits2Val3 = [bar,gnu,gnome],
Bits2Val4 = [0,1,1,1]
```

The above Bits2Val2, Bits2Val3 and Bits2Val4 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.

The compact notation equivalent to the empty BIT STRING is {0,<<>>}, which in the other notations is [] or 0.

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 unstructured information complying to two rules. Firstly, the bytes consist of octets and secondly, encoding is not required.

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

```
01 ::= OCTET STRING
02 ::= OCTET STRING (SIZE(28))
```

With the following example assignments in Erlang:

```
01Val = [17,13,19,20,0,0,255,254],
02Val = "must be exactly 28 chars....",
```

Observe that 01Val is assigned a series of numbers between 0 and 255 i.e. octets. 02Val 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 in the encoding scheme between OCTET STRINGS and other strings. The constraints specified for a type are especially important for PER, where they affect the encoding.

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:

```
DigsVal1 = "456",
DigsVal2 = "123",
TextFileVal1 = "abc...xyz...",
TextFileVal2 = [88,76,55,44,99,121 ..... a lot of characters here ....]
```

The Erlang representation for “BMPString” and “UniversalString” is either a list of ASCII values or a list of quadruples. The quadruple representation associates to the Unicode standard representation of characters. The ASCII characters are all represented by quadruples beginning with three zeros like {0,0,0,65} for the 'A' character. When decoding a value for these strings the result is a list of quadruples, or integers when the value is an ASCII character. The following example shows how it works:

In a file `PrimStrings.asn1` the type BMP is defined as

`BMP ::= BMPString` then using BER encoding the input/output format will be:

```
1> {ok,Bytes1} = asn1rt:encode('PrimStrings','BMP',[{0,0,53,53},{0,0,45,56}]).
{ok,[[30],[4],"55-8"]}
2> asn1rt:decode('PrimStrings','BMP',lists:flatten(Bytes1)).
{ok,[[{0,0,53,53},{0,0,45,56}]]}
3> {ok,Bytes2} = asn1rt:encode('PrimStrings','BMP',[{0,0,53,53},{0,0,0,65}]).
{ok,[[30],[4],[53,53,0,65]]}
4> asn1rt:decode('PrimStrings','BMP',lists:flatten(Bytes2)).
{ok,[[{0,0,53,53},65]]}
5> {ok,Bytes3} = asn1rt:encode('PrimStrings','BMP',"BMP string").
{ok,[[30],[20],[0,66,0,77,0,80,0,32,0,115,0,116,0,114,0,105,0,110,0,103]]}
6> asn1rt:decode('PrimStrings','BMP',lists:flatten(Bytes3)).
{ok,"BMP string"}
```

## 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'.

```
OidVal1 = {1,2,55},
```

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 [ASN.1 *Communication between Heterogeneous Systems* [7]], 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, Generalized Time and UTC (Universal Time Coordinated), both are assigned a value as an ordinary string within double quotes i.e. "19820102070533.8".

In case of DER encoding the compiler does not check the validity of the time values. The DER requirements upon those strings is regarded as a matter for the application to fulfill.

### 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.hr1 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 a SET.

A SEQUENCE and a SET may contain a component with a DEFAULT key word followed by the actual value that is the default value. In case of BER encoding it is optional to encode the value if it equals the default value. If the application uses the atom `asn1_DEFAULT` as value or if the value is a primitive value that equals the default value the encoding omits the bytes for this value, which is more efficient and it results in fewer bytes to send to the receiving application.

For instance, if the following types exists in a file "File.asn":

```
Seq1 ::= SEQUENCE {  
    a INTEGER DEFAULT 1,  
    b Seq2 DEFAULT {aa TRUE, bb 15}  
}  
  
Seq2 ::= SEQUENCE {  
    aa BOOLEAN,  
    bb INTEGER  
}
```



Some values and the corresponding encoding in an Erlang terminal is shown below:

```
1> asn1ct:compile('File').
Erlang ASN.1 version "1.3.2" compiling "File.asn1"
Compiler Options: []
--{generated,"File.asn1db"}--
--{generated,"File.hrl"}--
--{generated,"File.erl"}--
ok
2> 'File':encode('Seq1',{ 'Seq1',asn1_DEFAULT,asn1_DEFAULT}).
{ok,["0",[0],[[]],[[]]]}
3> lists:flatten(["0",[0],[[]],[[]]]).
[48,0]
4> 'File':encode('Seq1',{ 'Seq1',1,{ 'Seq2',true,15}}).
{ok,["0","\b",[[]],[["\241",[6],[[128],[1],"377"],[129],[1],[15]]]]]}
5> lists:flatten(["0","\b",[[]],[["\241",[6],[[128],[1],"377"],[129],[1],[15]]]]).
[48,8,161,6,128,1,255,129,1,15]
6>
```

The result after command line 3, in the example above, shows that the encoder omits the encoding of default values when they are specified by `asn1_DEFAULT`. Line 5 shows that even primitive values that equals the default value are detected and not encoded. But the constructed value of component `b` in `Seq1` is not recognized as the default value. Checking of default values in BER is not done in case of complex values, because it would be too expensive.

But, the DER encoding format has stronger requirements regarding default values both for SET and SEQUENCE. A more elaborate and time expensive check of default values will take place. The following is an example with the same types and values as above but with der encoding format.

```
1> asn1ct:compile('File',[der]).
Erlang ASN.1 version "1.3.2" compiling "File.asn1"
Compiler Options: [der]
--{generated,"File.asn1db"}--
--{generated,"File.hrl"}--
--{generated,"File.erl"}--
ok
2> 'File':encode('Seq1',{ 'Seq1',asn1_DEFAULT,asn1_DEFAULT}).
{ok,["0",[0],[[]],[[]]]}
3> lists:flatten(["0",[0],[[]],[[]]]).
[48,0]
4> 'File':encode('Seq1',{ 'Seq1',1,{ 'Seq2',true,15}}).
{ok,["0",[0],[[]],[[]]]}
5> lists:flatten(["0",[0],[[]],[[]]]).
[48,0]
6>
```

Line 5 shows that even values of constructed types is checked and if it equals the default value it will not be encoded.

## SET

The SET type is an unusual construct and normally the SEQUENCE type is more appropriate to use. Set is also inefficient compared with SEQUENCE, as the components can be in any order. Hence, it must be possible to distinguish every component in '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 [ITU-T X.680 [1]].

Encoding of a SET with components with DEFAULT values behaves similar as a SEQUENCE, see above [page 12]. The DER encoding format restrictions on DEFAULT values is the same for SET as for SEQUENCE, and is supported by the compiler, see above [page 13].

Moreover, in DER the elements of a SET will be sorted. If a component is an untagged choice the sorting have to take place in run-time. This fact emphasizes the following recommendation if DER encoding format is used.

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. So, if possible use SEQUENCE instead of SET.

## Notes about Extendability 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:

```
TVal1 = {y,17},
TVal2 = {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.

**Extendable CHOICE** When a CHOICE contains an extension marker and the decoder detects an unknown alternative of the CHOICE 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],
Arr2Val = ["abc", [14,34,54], "Octets"],
```

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.

However, in case of a value of the type SET OF, the DER encoding format requires the elements to be sent in ascending order of their encoding, which implies an expensive sorting procedure in run-time. Therefore it is strongly recommended to use SEQUENCE OF instead of SET OF if it is possible.

### 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.

For example:

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

```
Arr1 ::= SET SIZE (5) OF INTEGER (4..9)
```

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

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

### 1.1.5 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 Values, then the ASN.1 value tt can be reached from Erlang as a function call to 'Values':tt() as in the example below.

```
1> Val = 'Values':tt().  
{'TT',77,["kalle","kula"]}  
2> {ok,Bytes} = 'Values':encode('TT',Val).  
{ok,["0",  
    [18],  
    [[[128],[1],"M"],["\241","\r",[[[4],[5],"kalle"],[[4],[4],"kula"]]]]]]}  
3> FlatBytes = lists:flatten(Bytes).  
[48,18,128,1,77,161,13,4,5,107,97,108,108,101,4,4,107,117,108,97]  
4> 'Values':decode('TT',FlatBytes).  
{ok,{'TT',77,["kalle","kula"]}}  
5>
```

The above example shows that a function is generated by the compiler that returns a valid Erlang representation of the value, even though the value is of a complex type.

Furthermore, there is a macro generated for each value in the .hrl file. So, the defined value tt can also be extracted by ?tt in application code.

### 1.1.6 Macros

MACRO is not supported as the type is no longer part of the ASN.1 standard.

### 1.1.7 ASN.1 Information Objects (X.681)

Information Object Classes, Information Objects and Information Object Sets, (in the following called classes, objects and object sets respectively), are defined in the standard definition [ITU-T X.681 [2]]. In the following only a brief explanation is given.

These constructs makes it possible to define open types, i.e. values of that type can be of any ASN.1 type. It is also possible to define relationships between different types and values, since classes can hold types, values, objects, object sets and other classes in its fields. An Information Object Class may be defined in ASN.1 as:

```
GENERAL-PROCEDURE ::= CLASS {
    &Message,
    &Reply          OPTIONAL,
    &Error          OPTIONAL,
    &id             PrintableString UNIQUE
}
WITH SYNTAX {
    NEW MESSAGE    &Message
    [REPLY         &Reply]
    [ERROR         &Error]
    ADDRESS        &id
}
```

An object is an instance of a class and an object set is a set containing objects of one specified class. A definition may look like below.

The object `object1` is an instance of the CLASS GENERAL-PROCEDURE and has one type field and one fixed type value field. The object `object2` also has an OPTIONAL field ERROR, which is a type field.

```
object1 GENERAL-PROCEDURE ::= {
    NEW MESSAGE    PrintableString
    ADDRESS        "home"
}

object2 GENERAL-PROCEDURE ::= {
    NEW MESSAGE INTEGER
    ERROR INTEGER
    ADDRESS "remote"
}
```

The field ADDRESS is a UNIQUE field. Objects in an object set must have unique values in their UNIQUE field, as in GENERAL-PROCEDURES:

```
GENERAL-PROCEDURES GENERAL-PROCEDURE ::= {
    object1 | object2}
```

One can not encode a class, object or object set, only refering to it when defining other ASN.1 entities. Typically one refers to a class and to object sets by table constraints and component relation constraints [ITU-T X.682 [3]] in ASN.1 types, as in:

```
StartMessage ::= SEQUENCE {  
    msgId GENERAL-PROCEDURE.&id ({GENERAL-PROCEDURES}),  
    content GENERAL-PROCEDURE.&Message ({GENERAL-PROCEDURES}{@msgId}),  
}
```

In the type `StartMessage` the constraint following the content field tells that in a value of type `StartMessage` the value in the content field must come from the same object that is chosen by the `msgId` field.

So, the value `#'StartMessage'{msgId="home",content="Any Printable String"}` is legal to encode as a `StartMessage` value, while the value `#'StartMessage'{msgId="remote", content="Some String"}` is illegal since the constraint in `StartMessage` tells that when you have chosen a value from a specific object in the object set `GENERAL-PROCEDURES` in the `msgId` field you have to choose a value from that same object in the content field too. In this second case it should have been any `INTEGER` value.

`StartMessage` can in the content field be encoded with a value of any type that an object in the `GENERAL-PROCEDURES` object set has in its `NEW MESSAGE` field. This field refers to a type field `&Message` in the class. The `msgId` field is always encoded as a `PrintableString`, since the field refers to a fixed type in the class.

### 1.1.8 Parameterization (X.683)

Parameterization, which is defined in the standard [ITU-T X.683 [4]], can be used when defining types, values, value sets, information object classes, information objects or information object sets. A part of a definition can be supplied as a parameter. For instance, if a Type is used in a definition with certain purpose, one want the typename to express the intention. This can be done with parameterization.

When many types (or an other ASN.1 entity) only differs in some minor cases, but the structure of the types are similar, only one general type can be defined and the differences may be supplied through parameters.

One example of use of parameterization is:

```
General{Type} ::= SEQUENCE  
{  
    number    INTEGER,  
    string    Type  
}  
  
T1 ::= General{PrintableString}  
  
T2 ::= General{BIT STRING}
```

An example of a value that can be encoded as type `T1` is `{12,"hello"}`.

Observe that the compiler not generates encode/decode functions for parameterized types, only for the instances of the parameterized types. So, if a file contains the types `General{}`, `T1` and `T2` above, encode/decode functions will only be generated for `T1` and `T2`.



### 1.1.9 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.

Tags are implicitly encoded in the BER encoding as shown below, but are hardly not accounted for in the PER encoding. In PER tags are used for instance to sort the components of a SET.

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 'Adu' 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 as shown in the table below.

Tag	Len	Value
-----	-----	-------

Table 1.1: Structure of a BER encoded series of bytes

### 1.1.10 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.

The DER (Distinguished Encoding Rule) encoding format was included in the standard in 1994. It is a specialized form of BER, which gives the encoder the option to encode some entities differently. For instance, is the value for TRUE any octet with any bit set to one. But, DER does not leave any such choices. The value for TRUE in the DER case is encoded as the octet 11111111. So, the same value encoded by two different DER encoders must result in the same bit stream.

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 its 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 in that the encoded bytes are affected by the constraints. The BER encoded bytes are unaffected by the constraints. PER uses the sub-typing information to for example omit length fields whenever possible.

The run-time functions, sometimes take the constraints into account both for BER and PER. For instance are SIZE constrained strings checked.

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 25] – ASN.1 compiler and compile-time support functions
- Erlang Module **asn1rt** [page 29] – ASN.1 runtime support functions

### asn1ct

The following functions are exported:

- `compile(Asn1module) -> ok | {error,Reason}`  
[page 25] Compile an ASN.1 module and generate encode/decode functions according to the encoding rules BER or PER.
- `compile(Asn1module , Options) -> ok | {error,Reason}`  
[page 25] Compile an ASN.1 module and generate encode/decode functions according to the encoding rules BER or PER.
- `encode(Module,Type,Value)-> {ok,Bytes} | {error,Reason}`  
[page 26] Encode an ASN.1 value.
- `decode(Module,Type,Bytes) -> {ok,Value}|{error,Reason}`  
[page 27] Decode from Bytes into an ASN.1 value.
- `validate(Module,Type,Value) -> ok | {error,Reason}`  
[page 27] Validate an ASN.1 value.
- `start(IncludePaths) -> ok | already_started`  
[page 27] Start the ASN.1 database server.
- `stop() -> void`  
[page 27] Stop the ASN.1 database server.
- `value(Module ,Type) -> {ok,Value} | {error,Reason}`  
[page 27] Create an ASN.1 value for test purposes.
- `test(Module) -> ok | {error,Reason}`  
[page 28] Perform a test of encode and decode for types in an ASN.1 module.
- `test(Module,Type) -> ok | {error,Reason}`  
[page 28] Perform a test of encode and decode for types in an ASN.1 module.
- `test(Module,Type,Value) -> ok | {error,Reason}`  
[page 28] Perform a test of encode and decode for types in an ASN.1 module.

## asn1rt

The following functions are exported:

- `encode(Module,Type,Value)-> {ok,Bytes} | {error,Reason}`  
[page 29] Encode an ASN.1 value.
- `decode(Module,Type,Bytes) -> {ok,Value}|{error,Reason}`  
[page 29] Decode from bytes into an ASN.1 value.
- `validate(Module,Type,Value) -> ok | {error,Reason}`  
[page 29] Validate an ASN.1 value.

# asn1ct

## Erlang Module

The ASN.1 compiler takes an ASN.1 module as input and generates a corresponding Erlang module which can encode and decode the datatypes specified. Alternatively the compiler takes a specification module (see below) specifying all input modules and generates one module with encode/decode functions. There are also some generic functions which can be used in the development of applications which handle 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 | ber\_bin | per\_bin | der | compact\_bit\_string | noobj | {outdir,Dir} | {i,IncludeDir}
- Reason = term()

Compiles the ASN.1 module `Asn1module` and generates an Erlang module `Asn1module.erl` with encode and 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 `".asn1"` is assumed, then `".asn"` and finally `".py"` (to be compatible with the old ASN.1 compiler). Of course `Asn1module` can be a full pathname (relative or absolute) including filename with (or without) extension.

If one wishes to compile a set of ASN.1 modules into one Erlang file with encode/decode functions one has to list all involved files in a configuration file. This configuration file must have a double extension `".set.asn"`, (`".asn"` can alternatively be `".asn1"` or `".py"`). The input files' names must be listed, within quotation marks (`"`), one at each row in the file. If the input files are `File1.asn`, `File2.asn` and `File3.asn` the configuration file shall look like:

```
"File1.asn"
"File2.asn"
"File3.asn"
```

The output files will in this case get their names from the configuration file. If the configuration file has the name `SetOfFiles.set.asn` the name of the output files will be `SetOfFiles.hrl`, `SetOfFiles.hrl` and `SetOfFiles.asn1db`.

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

`ber` | `ber_bin` | `per` | `per_bin` 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 consequence of this only one encoding rule per ASN.1 module can be used at runtime.

The `ber_bin` and `per_bin` options are equivalent with the `ber` and `per` options with the difference that the generated encoding/decoding functions take advantage of the bit syntax, which in most cases increases the performance considerably. The result from encoding is a list (maybe nested) with Erlang terms, including binaries. Note that the erlang virtual machine that will execute the generated code must be of version R7 or higher.

`der` By this option the Distinguished Encoding Rule (DER) is chosen. DER is regarded as a specialized variant of the BER encoding rule, therefore the `der` option only makes sense when the `ber` or `ber_bin` option is used. This option sometimes adds sorting and value checks when encoding, which implies a slower encoding. The decoding routines are the same as for `ber`.

`compact_bit_string` Makes it possible to use a compact notation for values of the BIT STRING type in Erlang. The notation:

```
BitString = {Unused, Binary},
Unused = integer(),
Binary = binary()
```

`Unused` must be a number in the range 0 to 7. It tells how many bits in the least significant byte in `Binary` that is unused. For details see BIT STRING type section in users guide [page 8].

`{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}
```

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 occurred.

`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}`

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

Erlang 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}`

Types:

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

Encodes `Value` of `Type` defined in the ASN.1 module `Module`. Returns a possibly nested list of bytes and or binaries 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 = binary | [Int]` when `integer(Int)`, `Int >= 0`, `Int <= 255` | `binary`

Decodes `Type` from `Module` from the list of bytes `Bytes`. If the module is compiled with `ber_bin` or `per_bin` option `Bytes` must be a binary. Returns `{ok,Value}` if successful.

`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

1.1 Structure of a BER encoded series of bytes . . . . . 21



# Bibliography

- [1] ITU-T Recommendation X.680 (1994) | ISO/IEC 8824-1: 1995, Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.
- [2] ITU-T Recommendation X.681 (1994) | ISO/IEC 8824-2: 1995, Abstract Syntax Notation One (ASN.1): Information Object Specification.
- [3] ITU-T Recommendation X.682 (1994) | ISO/IEC 8824-3: 1995, Abstract Syntax Notation One (ASN.1): Constraint Specification.
- [4] ITU-T Recommendation X.683 (1994) | ISO/IEC 8824-4: 1995, Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 Specifications.
- [5] ITU-T Recommendation X.690 (1994) | ISO/IEC 8825-1: 1995, ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).
- [6] ITU-T Recommendation X.691 (04/95) | ISO/IEC 8825-2: 1995, ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER).
- [7] Oliver Dubuisson, ASN.1 Communication between Heterogeneous Systems, June 2000 ISBN 0-126333361-0.
- [8] J. Armstrong, R. Virding, C. Wikstrom, M. Williams, Concurrent Programming in ERLANG, Prentice Hall, 1996, ISBN 0-13-508301-X.



# Index of Modules and Functions

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

<i>asn1ct</i>	<i>asn1ct</i> , 28
compile/1, 25	
compile/2, 25	test/3
decode/3, 27	<i>asn1ct</i> , 28
encode/3, 26	
start/1, 27	validate/3
stop/0, 27	<i>asn1ct</i> , 27
test/1, 28	<i>asn1rt</i> , 29
test/2, 28	
test/3, 28	value/2
validate/3, 27	<i>asn1ct</i> , 27
value/2, 27	
<i>asn1rt</i>	
decode/3, 29	
encode/3, 29	
validate/3, 29	
compile/1	
<i>asn1ct</i> , 25	
compile/2	
<i>asn1ct</i> , 25	
decode/3	
<i>asn1ct</i> , 27	
<i>asn1rt</i> , 29	
encode/3	
<i>asn1ct</i> , 26	
<i>asn1rt</i> , 29	
start/1	
<i>asn1ct</i> , 27	
stop/0	
<i>asn1ct</i> , 27	
test/1	
<i>asn1ct</i> , 28	
test/2	

