

ASN.1

Copyright © 1997-2026 Ericsson AB. All Rights Reserved.
ASN.1 5.2.2.1
November 25, 2026

Copyright © 1997-2026 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

November 25, 2026

1 Asn1 User's Guide

The ASN.1 application contains modules with compile-time and runtime support for Abstract Syntax Notation One (ASN.1).

1.1 Introduction

The ASN.1 application provides the following:

- 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.
- Runtime functions used by the generated code.
- Support for the following encoding rules:
  - Basic Encoding Rules (BER)
  - Distinguished Encoding Rules (DER), a specialized form of BER that is used in security-conscious applications
  - Packed Encoding Rules (PER), both the aligned and unaligned variant

1.1.1 Scope

This application covers all features of ASN.1 up to the 1997 edition of the specification. In the 2002 edition, new features were introduced. The following features of the 2002 edition are fully or partly supported:

- Decimal notation (for example, `"1.5e3`) for REAL values. The NR1, NR2, and NR3 formats as explained in ISO 6093 are supported.
- The `RELATIVE-OID` type for relative object identifiers is fully supported.
- The subtype constraint (`CONTAINING`/`ENCODED BY`) to constrain the content of an octet string or a bit string is parsed when compiling, but no further action is taken. This constraint is not a PER-visible constraint.
- The subtype constraint by regular expressions (`PATTERN`) for character string types is parsed when compiling, but no further action is taken. This constraint is not a PER-visible constraint.
- Multiple-line comments as in C, `/* ... */`, are supported.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language, concepts of OTP, and is familiar with the ASN.1 notation. The ASN.1 notation is documented in the standard definition X.680, which is the primary text. It can also be helpful, but not necessary, to read the standard definitions X.681, X.682, X.683, X.690, and X.691.

A good book explaining those reference texts is Dubuisson: ASN.1 - Communication Between Heterogeneous Systems, is free to download at **http://www.oss.com/asn1/dubuisson.html**.

1.2 ASN.1

1.2.1 Introduction

ASN.1 is a formal language for describing data structures to be exchanged between distributed computer systems. The purpose of ASN.1 is to have a platform and programming language independent notation 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 any type of communication channel. This way, two applications written in different programming languages running on different computers, and with different internal representation of data, can exchange instances of structured data types.

1.3 Getting Started

1.3.1 Example

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

Create a file named `People.asn` containing the following:

```
People DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  Person ::= SEQUENCE {
    name PrintableString,
    location INTEGER {home(0),field(1),roving(2)},
    age INTEGER OPTIONAL
  }
END
```

This file 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. The code-generator then uses the abstract syntax tree to generate code.

The generated Erlang files are placed in the current directory or in the directory specified with option `{outdir,Dir}`.

The following shows how the compiler can be called from the Erlang shell:

```
1> asn1ct:compile("People", [ber]).
ok
2>
```

Option `verbose` can be added to get information about the generated files:

```
2> asn1ct:compile("People", [ber,verbose]).
Erlang ASN.1 compiling "People.asn"
--{generated,"People.asn1db"}--
--{generated,"People.hrl"}--
--{generated,"People.erl"}--
ok
3>
```

ASN.1 module `People` is now accepted and the abstract syntax tree is saved in file `People.asn1db`. The generated Erlang code is compiled using the Erlang compiler and loaded into the Erlang runtime system. There is now an API for `encode/2` and `decode/2` in module `People`, which is called like:

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

or

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

Assume that there is a network application that 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,
```

In this example, 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 binary according to the BER or PER encoding rules.

The encoder and decoder can also be run from the shell:

```
2> Rockstar = {'Person',"Some Name",roving,50}.
{'Person',"Some Name",roving,50}
3> {ok,Bin} = 'People':encode('Person',Rockstar).
{ok,<<243,17,19,9,83,111,109,101,32,78,97,109,101,2,1,2,
      2,1,50>>}
4> {ok,Person} = 'People':decode('Person',Bin).
{ok,{'Person',"Some Name",roving,50}}
5>
```

Module Dependencies

It is common that ASN.1 modules import defined types, values, and other entities from another ASN.1 module.

Earlier versions of the ASN.1 compiler required that modules that were imported from had to be compiled before the module that imported. This caused problems when ASN.1 modules had circular dependencies.

Referenced modules are now parsed when the compiler finds an entity that is imported. No code is generated for the referenced module. However, the compiled modules rely on that the referenced modules are also compiled.

1.3.2 ASN.1 Application User Interface

The `ASN.1` application provides the following two separate user interfaces:

- The module `asn1ct`, which provides the compile-time functions (including the compiler)
- The module `asn1rt_nif`, which provides the runtime functions for the ASN.1 decoder for the BER back end

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

Compile-Time Functions

The ASN.1 compiler can be started directly from the command line by the `erlc` program. This is convenient when compiling many ASN.1 files from the command line or when using Makefiles. Some examples of how the `erlc` command can be used to start the ASN.1 compiler:

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

Useful options for the ASN.1 compiler:

`-b[ber | per | uper | jer]`

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 and ASN.1 source specs to resolve references to other modules. This option can be repeated many times if there are several places to search in. The compiler searches the current directory first.

`+der`

DER encoding rule. Only when using option `-bber`.

`+jer`

Functions `jer_encode/2` and `jer_decode/2` for JSON encoding rules are generated together with functions for `ber` or `per`. Only to be used when the main encoding option is `-bber`, `-bper` or `-buper`

`+maps`

Use maps instead of records to represent the `SEQUENCE` and `SET` types. No `.hrl` files will be generated. See the Section Map representation for SEQUENCE and SET for more information.

`+asn1config`

This functionality works together with option `ber`. It enables the specialized decodes, see Section Specialized Decode.

`+undec_rest`

A buffer that holds a message being decoded can also have trailing bytes. If those trailing bytes are important, they can be returned along with the decoded value by compiling the ASN.1 specification with option `+undec_rest`. The return value from the decoder is `{ok,Value,Rest}` where `Rest` is a binary containing the trailing bytes.

`+'Any Erlc Option'`

Any option can be added to the Erlang compiler when compiling the generated Erlang files. Any option unrecognized by the ASN.1 compiler is passed to the Erlang compiler.

For a complete description of `erlc`, see ERTS Reference Manual.

The compiler and other compile-time functions can also be started from the Erlang shell. Here follows a brief description of the primary functions. For a complete description of each function, see module `asn1ct` in the ASN.1 Reference Manual.

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

Example:

```
asn1ct:compile("H323-MESSAGES.asn1").
```

This equals:

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

If PER encoding is wanted:

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

The generic encode and decode functions can be called as follows:

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

Runtime Functions

When an ASN.1 specification is compiled with option `ber`, the `asn1rt_nif` module and the NIF library in `asn1/priv_dir` are needed at runtime.

By calling function `info/0` in a generated module, you get information about which compiler options were used.

Errors

Errors detected at compile-time are displayed on the screen together with line numbers indicating where in the source file the respective error was detected. If no errors are found, an Erlang ASN.1 module is created.

The runtime encoders and decoders execute within a catch and return `{ok, Data}` or `{error, {asn1, Description}}` where `Description` is an Erlang term describing the error.

Currently, `Description` looks like this: `{ErrorDescription, StackTrace}`. Applications should not depend on the exact contents of `Description` as it could change in the future.

1.3.3 Multi-File Compilation

There are various reasons for using multi-file compilation:

- To choose the name for the generated module, for example, because you need to compile the same specs for different encoding rules.
- You want only one resulting module.

Specify which ASN.1 specs to compile in a module with extension `.set.asn`. Choose a module name and provide the names of the ASN.1 specs. For example, if you have the specs `File1.asn`, `File2.asn`, and `File3.asn`, your module `MyModule.set.asn` looks as follows:

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

If you compile with the following, the result is one merged module `MyModule.erl` with the generated code from the three ASN.1 specs:

```
~> erlc MyModule.set.asn
```

1.3.4 Remark about Tags

Tags used to be important for all users of ASN.1, because it was necessary to add tags manually to certain constructs in order for the ASN.1 specification to be valid. Example of an old-style specification:

```
Tags DEFINITIONS ::=
BEGIN
  Afters ::= CHOICE { cheese [0] IA5String,
                      dessert [1] IA5String }
END
```

Without the tags (the numbers in square brackets) the ASN.1 compiler refused to compile the file.

In 1994 the global tagging mode `AUTOMATIC TAGS` was introduced. By putting `AUTOMATIC TAGS` in the module header, the ASN.1 compiler automatically adds tags when needed. The following is the same specification in `AUTOMATIC TAGS` mode:

```
Tags DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  Afters ::= CHOICE { cheese IA5String,
                      dessert IA5String }
END
```

Tags are not mentioned any more in this User's Guide.

1.3.5 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	**Constructed Types**
BOOLEAN	SEQUENCE
INTEGER	SET
REAL	CHOICE
NULL	SET OF and SEQUENCE OF
ENUMERATED	ANY
BIT STRING	ANY DEFINED BY
OCTET STRING	EXTERNAL
Character Strings	EMBEDDED PDV
OBJECT IDENTIFIER	CHARACTER STRING
Object Descriptor	
TIME Types	

Table 3.1: Supported ASN.1 Types

Note:

The values of each ASN.1 type have their own representation in Erlang, as described in the following sections. Users must provide these values for encoding according to the representation, as shown in the following example:

```
Operational ::= BOOLEAN --ASN.1 definition
```

In Erlang code it can look as follows:

```
Val = true,
{ok,Bytes} = MyModule:encode('Operational', Val),
```

BOOLEAN

Booleans in ASN.1 express values that can be either TRUE or FALSE. The meanings assigned to TRUE and FALSE are outside the scope of this text.

In ASN.1 it is possible to have:

```
Operational ::= BOOLEAN
```

Assigning a value to 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. Erlang systems with version 4.3 and higher support very large integers, in practice indefinitely large integers.

The concept of subtyping can be applied to integers and to other ASN.1 types. The details of subtyping are not explained here; for more information, see X.680. Various 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)}
```

The Erlang representation of an ASN.1 `INTEGER` is an integer or an atom if a `Named Number List` (see `T6` in the previous list) is specified.

The following is an example of Erlang code that assigns values for the types in the previous list:

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

These Erlang variables 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.

The decoder returns an atom if the value corresponds to a symbol in the `Named Number List`.

REAL

The following ASN.1 type is used for real numbers:

```
R1 ::= REAL
```

It is assigned a value in Erlang as follows:

```
R1value1 = "2.14",
R1value2 = {256,10,-2},
```

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

NULL

The type `NULL` is suitable where supply and recognition of a value is important but the actual value is not.

```
Notype ::= NULL
```

This type is assigned in Erlang as follows:

```
N1 = 'NULL',
```

The actual value is the quoted atom `'NULL'`.

ENUMERATED

The type `ENUMERATED` can be used when the value you want to describe can only take one of a set of predefined values. Example:

```
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 similar to an integer type, when defined with a set of predefined values. The difference is that an enumerated type can only have specified values, whereas an integer can have any value.

BIT STRING

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

In ASN.1, `BIT STRING` definitions can look as follows:

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

The following two notations are available for representation of `BIT STRING` values in Erlang and as input to the encode functions:

- A bitstring. By default, a `BIT STRING` with no symbolic names is decoded to an Erlang bitstring.
- A list of atoms corresponding to atoms in the `NamedBitList` in the `BIT STRING` definition. A `BIT STRING` with symbolic names is always decoded to the format shown in the following example:

```
Bits1Val1 = <<0:1,1:1,0:1,1:1,1:1>>,
Bits2Val1 = [gnu,punk],
Bits2Val2 = <<2#1110:4>>,
Bits2Val3 = [bar,gnu,gnome],
```

`Bits2Val2` and `Bits2Val3` denote the same value.

`Bits2Val1` is assigned symbolic values. The assignment means that the bits corresponding to `gnu` and `punk`, that is, bits 2 and 14 are set to 1, and the rest are set to 0. The symbolic values are shown as a list of values. If a named value, which is not specified in the type definition, is shown, a runtime error occurs.

`BIT STRING`s can also be subtyped with, for example, a `SIZE` specification:

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

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

Deprecated Representations for BIT STRING

In addition to the representations described earlier, the following deprecated representations are available if the specification has been compiled with option `legacy_erlang_types`:

- Aa a list of binary digits (0 or 1). This format is accepted as input to the encode functions, and a `BIT STRING` is decoded to this format if option **legacy_bit_string** is given.
- As `{Unused,Binary}` where `Unused` denotes how many trailing zero-bits 0-7 that are unused in the least significant byte in `Binary`. This format is accepted as input to the encode functions, and a `BIT STRING` is decoded to this format if `compact_bit_string` has been given.
- As a hexadecimal number (or an integer). Avoid this as it is easy to misinterpret a `BIT STRING` value in this format.

OCTET STRING

`OCTET STRING` is the simplest of all ASN.1 types. `OCTET STRING` only moves or transfers, for example, binary files or other unstructured information complying with two rules: the bytes consist of octets and encoding is not required.

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 = <<17,13,19,20,0,0,255,254>>,
O2Val = <<"must be exactly 28 chars....">>,
```

By default, an `OCTET STRING` is always represented as an Erlang binary. If the specification has been compiled with option `legacy_erlang_types`, the encode functions accept both lists and binaries, and the decode functions decode an `OCTET STRING` to a list.

Character Strings

ASN.1 supports a wide variety of character sets. The main difference between an `OCTET STRING` and a character string is that the `OCTET STRING` has no imposed semantics on the bytes delivered.

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

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

The ASN.1 to Erlang compiler does not determine the correct interpretation of each BER string octet value with different character strings. The application is responsible for interpretation of octets. Therefore, from the BER string point of view, octets are very similar to character strings and are compiled in the same way.

When PER is used, there is a significant difference in the encoding scheme between `OCTET STRING`s and other strings. The constraints specified for a type are especially important for PER, where they affect the encoding.

Examples:

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

The corresponding 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 character 'A'. 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. Assume the following specification is in file `PrimStrings.asn1`:

```
PrimStrings DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
   BMP ::= BMPString
END
```

Encoding and decoding some strings:

```
1> asn1ct:compile('PrimStrings', [ber]).
ok
2> {ok,Bytes1} = 'PrimStrings':encode('BMP', [{0,0,53,53},{0,0,45,56}]).
{ok,<<30,4,53,54,45,56>>}
3> 'PrimStrings':decode('BMP', Bytes1).
{ok,[{0,0,53,53},{0,0,45,56}]}
4> {ok,Bytes2} = 'PrimStrings':encode('BMP', [{0,0,53,53},{0,0,0,65}]).
{ok,<<30,4,53,53,0,65>>}
5> 'PrimStrings':decode('BMP', Bytes2).
{ok,[{0,0,53,53},65]}
6> {ok,Bytes3} = 'PrimStrings':encode('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>>}
7> 'PrimStrings':decode('BMP', Bytes3).
{ok,"BMP string"}
```

Type UTF8String is represented as a UTF-8 encoded binary in Erlang. Such binaries can be created directly using the binary syntax or by converting from a list of Unicode code points using function `unicode:characters_to_binary/1`.

The following shows examples of how UTF-8 encoded binaries can be created and manipulated:

```
1> Gs = "### ######### ####".
[1052,1086,1081,32,1084,1072,1083,1077,1085,1100,1082,1080,
 1081,32,1043,1085,1086,1084]
2> Gbin = unicode:characters_to_binary(Gs).
<<208,156,208,190,208,185,32,208,188,208,176,208,187,208,
  181,208,189,209,140,208,186,208,184,208,185,32,208,147,
  208,...>>
3> Gbin = <<"### ######### ####"/utf8>>.
<<208,156,208,190,208,185,32,208,188,208,176,208,187,208,
  181,208,189,209,140,208,186,208,184,208,185,32,208,147,
  208,...>>
4> Gs = unicode:characters_to_list(Gbin).
[1052,1086,1081,32,1084,1072,1083,1077,1085,1100,1082,1080,
 1081,32,1043,1085,1086,1084]
```

For details, see the unicode module in STDLIB.

In the following example, this ASN.1 specification is used:

```
UTF DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
   UTF ::= UTF8String
END
```

Encoding and decoding a string with Unicode characters:

```
5> asn1ct:compile('UTF', [ber]).
ok
6> {ok,Bytes1} = 'UTF':encode('UTF', <<"####"/utf8>>).
{ok,<<12,8,208,147,208,189,208,190,208,188>>}
7> {ok,Bin1} = 'UTF':decode('UTF', Bytes1).
{ok,<<208,147,208,189,208,190,208,188>>}
8> io:format("~ts\n", [Bin1]).
####
ok
9> unicode:characters_to_list(Bin1).
[1043,1085,1086,1084]
```

OBJECT IDENTIFIER

The type `OBJECT IDENTIFIER` is used whenever a unique identity is required. An ASN.1 module, a transfer syntax, and so on, is identified with an `OBJECT IDENTIFIER`. Assume the following example:

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

Therefore, the following example is a valid Erlang instance of 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. The second value must be in the range 0..39 when the first value is 0 or 1.

The `OBJECT IDENTIFIER` is an important type and it is widely used within different standards to identify various objects uniquely. Dubuisson: ASN.1 - Communication Between Heterogeneous Systems includes an easy-to-understand description of the use of `OBJECT IDENTIFIER`.

Object Descriptor

Values of this type can be assigned a value as an ordinary string as follows:

```
     "This is the value of an Object descriptor"
```

TIME Types

Two time types are defined within ASN.1: Generalized Time and Universal Time Coordinated (UTC). Both are assigned a value as an ordinary string within double quotes, for example, "19820102070533.8".

For DER encoding, the compiler does not check the validity of the time values. The DER requirements upon those strings are 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` can be defined as follows:

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

This is a 4-component structure called `Pdu`. By default, a `SEQUENCE` is represented by a record in Erlang. It can also be represented as a map; see Map representation for SEQUENCE and SET. For each `SEQUENCE` and `SET` in an ASN.1 module an Erlang record declaration is generated. For `Pdu`, a record like the following 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 follows:

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

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

A `SEQUENCE` and a `SET` can contain a component with a `DEFAULT` keyword followed by the actual value, which is the default value. The `DEFAULT` keyword means that the application doing the encoding can omit encoding of the value, which results in fewer bytes to send to the receiving application.

An application can use the atom `asn1_DEFAULT` to indicate that the encoding is to be omitted for that position in the `SEQUENCE`.

Depending on the encoding rules, the encoder can also compare the given value to the default value and automatically omit the encoding if the values are equal. How much effort the encoder makes to compare the values depends on the encoding rules. The DER encoding rules forbid encoding a value equal to the default value, so it has a more thorough and time-consuming comparison than the encoders for the other encoding rules.

In the following example, this ASN.1 specification is used:

```
File DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
Seq1 ::= SEQUENCE {
    a INTEGER DEFAULT 1,
    b Seq2 DEFAULT {aa TRUE, bb 15}
}

Seq2 ::= SEQUENCE {
    aa BOOLEAN,
    bb INTEGER
}

Seq3 ::= SEQUENCE {
    bs BIT STRING {a(0), b(1), c(2)} DEFAULT {a, c}
}
END
```

Example where the BER encoder is able to omit encoding of the default values:

```
1> asn1ct:compile('File', [ber]).
ok
2> 'File':encode('Seq1', {'Seq1',asn1_DEFAULT,asn1_DEFAULT}).
{ok,<<48,0>>}
3> 'File':encode('Seq1', {'Seq1',1,{'Seq2',true,15}}).
{ok,<<48,0>>}
```

Example with a named `BIT STRING` where the BER encoder does not omit the encoding:

```
4> 'File':encode('Seq3', {'Seq3',asn1_DEFAULT).
{ok,<<48,0>>}
5> 'File':encode('Seq3', {'Seq3',<<16#101:3>>).
{ok,<<48,4,128,2,5,160>>}
```

The DER encoder omits the encoding for the same `BIT STRING`:

```
6> asn1ct:compile('File', [ber,der]).
ok
7> 'File':encode('Seq3', {'Seq3',asn1_DEFAULT).
{ok,<<48,0>>}
8> 'File':encode('Seq3', {'Seq3',<<16#101:3>>).
{ok,<<48,0>>}
```

SET

In Erlang, the `SET` type is used exactly as `SEQUENCE`. Notice that if BER or DER encoding rules are used, decoding a `SET` is slower than decoding a `SEQUENCE` because the components must be sorted.

Extensibility for SEQUENCE and SET

When a `SEQUENCE` or `SET` contains an extension marker and extension components as the following, the type can get more components in newer versions of the ASN.1 spec:

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

In this case it has got a new component b. Thus, incoming messages that are decoded can have more or fever components than this one.

The component b is treated as an original component when encoding a message. In this case, as it is not an optional element, it must be encoded.

During decoding, the b field of the record gets the decoded value of the b component, if present, otherwise the value asn1_NOVALUE.

Map representation for SEQUENCE and SET

If the ASN.1 module has been compiled with option maps, the types SEQUENCE and SET are represented as maps.

In the following example, this ASN.1 specification is used:

```
File DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
Seq1 ::= SEQUENCE {
    a INTEGER DEFAULT 42,
    b BOOLEAN OPTIONAL,
    c IA5String
}
END
```

Optional fields are to be omitted from the map if they have no value:

```
1> asn1ct:compile('File', [per,maps]).
ok
2> {ok,E} = 'File':encode('Seq1', #{a=>0,c=>"string"}).
{ok,<<128,1,0,6,115,116,114,105,110,103>>}
```

When decoding, optional fields will be omitted from the map:

```
3> 'File':decode('Seq1', E).
{ok,#{a => 0,c => "string"}}
```

Default values can be omitted from the map:

```
4> {ok,E2} = 'File':encode('Seq1', #{c=>"string"}).
{ok,<<0,6,115,116,114,105,110,103>>}
5> 'File':decode('Seq1', E2).
{ok,#{a => 42,c => "string"}}
```

Note:

It is not allowed to use the atoms asn1_VALUE and asn1_DEFAULT with maps.

CHOICE

The type CHOICE is a space saver and is similar to the concept of a 'union' in C.

Assume the following:

```
SomeModuleName DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
T ::= CHOICE {
        x REAL,
        y INTEGER,
        z OBJECT IDENTIFIER }
END
```

It is then possible to assign values as follows:

```
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.

Extensible CHOICE

When a `CHOICE` contains an extension marker and the decoder detects an unknown alternative of the `CHOICE`, the value is represented as follows:

```
{asn1_ExtAlt, BytesForOpenType}
```

Here `BytesForOpenType` is a list of bytes constituting the encoding of the "unknown" `CHOICE` alternative.

SET OF and SEQUENCE OF

The types `SET OF` and `SEQUENCE OF` correspond to the concept of an array in several programming languages. The Erlang syntax for both types is straightforward, for example:

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

In Erlang the following can apply:

```
Arr1Val = [4,5,6,7,8],
Arr2Val = ["abc",[14,34,54],"Octets"],
```

Notice that the definition of type `SET OF` 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, for a value of 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 runtime. Therefore it is recommended to use `SEQUENCE OF` instead of `SET OF` if possible.

ANY and ANY DEFINED BY

The types `ANY` and `ANY DEFINED BY` have been removed from the standard since 1994. It is recommended not to use these types any more. They can, however, exist in some old ASN.1 modules. The idea with this type was to leave a "hole" in a definition where it was possible to put unspecified data of any kind, even non-ASN.1 data.

A value of this type is encoded as an `open type`.

Instead of `ANY` and `ANY DEFINED BY`, it is recommended to use `information object class`, `table constraints`, and `parameterization`. In particular the construct `TYPE-IDENTIFIER.@Type` accomplish the same as the deprecated `ANY`.

See also Information object.

EXTERNAL, EMBEDDED PDV, and CHARACTER STRING

The types `EXTERNAL`, `EMBEDDED PDV`, and `CHARACTER STRING` are used in presentation layer negotiation. They are encoded according to their associated type, see X.680.

The type `EXTERNAL` had a slightly different associated type before 1994. X.691 states that encoding must follow the older associated type. So, generated encode/decode functions convert values of the newer format to the older format before encoding. This implies that it is allowed to use `EXTERNAL` type values of either format for encoding. Decoded values are always returned in the newer format.

Embedded Named Types

The structured types previously described can have other named types as their components. The general syntax to assign a value to component `C` of a named ASN.1 type `T` in Erlang is the record syntax `#'T'{'C'=Value}`. Here `Value` can be a value of yet another type `T2`, for example:

```
EmbeddedExample DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
B ::= SEQUENCE {
        a Arr1,
        b T }

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

T ::= CHOICE {
        x REAL,
        y INTEGER,
        z OBJECT IDENTIFIER }
        END
```

`SEQUENCE b` can be encoded as follows in Erlang:

```
1> 'EmbeddedExample':encode('B', {'B',[4,5,6,7,8],{x,"7.77"}}).
{ok,<<5,56,0,8,3,55,55,55,46,69,45,50>>}
```

1.3.6 Naming of Records in .hrl Files

When the option `maps` is given, no `.hrl` files will be generated. The rest of this section describes the behavior of the compiler when `maps` is not used.

When an ASN.1 specification is compiled, all defined types of type `SET` or `SEQUENCE` result in a corresponding record in the generated `.hrl` file. This is because the values for `SET` and `SEQUENCE` are represented as records by default.

Some special cases of this functionality are presented in the next section.

Embedded Structured Types

In ASN.1 it is also possible to have components that are themselves structured types. For example, it is possible to have the following:

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

FooType ::= [3] VisibleString
```

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

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

Values of type `Emb` can be assigned as follows:

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

For an embedded type of type `SEQUENCE`/`SET` in a `SEQUENCE`/`SET`, the record name is extended with an underscore and the component name. If the embedded structure is deeper with the `SEQUENCE`, `SET`, or `CHOICE` types in the line, each component name/alternative name is added to the record name.

Example:

```
Seq ::= SEQUENCE{
    a CHOICE{
        b SEQUENCE {
           c  INTEGER
        }
    }
}
```

This results in the following record:

```
-record('Seq_a_b',{c}).
```

If the structured type has a component with an embedded `SEQUENCE OF`/`SET OF` which embedded type in turn is a `SEQUENCE`/`SET`, it gives a record with the `SEQUENCE OF`/`SET OF` addition as in the following example:

```
Seq ::= SEQUENCE {
    a SEQUENCE OF SEQUENCE {
           b
               }
    c SET OF SEQUENCE {
           d
               }
}
```

This results in the following records:

```
-record('Seq_a_SEQOF'{b}).
-record('Seq_c_SETOF'{d}).
```

A parameterized type is to be considered as an embedded type. Each time such a type is referenced, an instance of it is defined. Thus, in the following example a record with name `'Seq_b'` is generated in the `.hrl` file and is used to hold values:

```
Seq ::= SEQUENCE {
    b PType{INTEGER}
}

PType{T} ::= SEQUENCE{
    id T
}
```

Recursive Types

Types that refer to themselves are called recursive types. Example:

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

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 follows:

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

1.3.7 ASN.1 Values

Values can be assigned to an ASN.1 type within the ASN.1 code itself, as opposed to the actions in the previous section 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. A short example:

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

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

The value defined here can be used in several ways. It can, for example, be used as the value in some `DEFAULT` component:

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

It can also be used from inside an Erlang program. If this ASN.1 code is defined in ASN.1 module `Values`, the ASN.1 value `tt` can be reached from Erlang as a function call to `'Values':tt()` as in the following example:

```
1> Val = 'Values':tt().
{'TT',77,["kalle","kula"]}
2> {ok,Bytes} = 'Values':encode('TT',Val).
{ok,<<48,18,128,1,77,161,13,4,5,107,97,108,108,101,4,4,
      107,117,108,97>>}
4> 'Values':decode('TT',Bytes).
{ok,{'TT',77,["kalle","kula"]}}
5>
```

This example shows that a function is generated by the compiler that returns a valid Erlang representation of the value, although the value is of a complex type.

Furthermore, if the option `maps` is not used, a macro is generated for each value in the `.hrl` file. So, the defined value `tt` can also be extracted by `?tt` in application code.

1.3.8 Macros

The type `MACRO` is not supported. It is no longer part of the ASN.1 standard.

1.3.9 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 X.681. Only a brief explanation is given here.

These constructs makes it possible to define open types, that is, values of that type can be of any ASN.1 type. Also, relationships can be defined between different types and values, as classes can hold types, values, objects, object sets, and other classes in their fields. A class can be defined in ASN.1 as follows:

```
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. An object set is a set containing objects of a specified class. A definition can look as follows:

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

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

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` has also an optional field `ERROR`, which is a type field. 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}
```

You cannot encode a class, object, or object set, only refer to it when defining other ASN.1 entities. Typically you refer to a class as well as to object sets by table constraints and component relation constraints (X.682) in ASN.1 types, as in the following:

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

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

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

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

In practice, object sets are usually declared to be extensible so that more objects can be added to the set later. Extensibility is indicated as follows:

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

When decoding a type that uses an extensible set constraint, it is always possible that the value in field UNIQUE is unknown (that is, the type has been encoded with a later version of the ASN.1 specification). The unencoded data is then returned wrapped in a tuple as follows:

```
{asn1_OPENTYPE,Binary}
```

Here Binary is an Erlang binary that contains the encoded data. (If option legacy_erlang_types has been given, only the binary is returned.)

1.3.10 Parameterization (X.683)

Parameterization, which is defined in X.683, can be used when defining types, values, value sets, classes, objects, or object sets. A part of a definition can be supplied as a parameter. For example, if a Type is used in a definition with a certain purpose, you want the type name to express the intention. This can be done with parameterization.

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

Example of use of parameterization:

```
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"}`.

Notice that the compiler does not generate encode/decode functions for parameterized types, only for the instances of the parameterized types. Therefore, if a file contains the types `General{}`, `T1`, and `T2` as in the previous example, encode/decode functions are only generated for `T1` and `T2`.

1.4 Specialized Decodes

When performance is of highest priority and you are interested in a limited part of the ASN.1 encoded message before deciding what to do with the rest of it, an option is to decode only this small part. The situation can be a server that has to decide the addressee of a message. The addressee can be interested in the entire message, but the server can be a bottleneck that you want to spare any unnecessary load.

Instead of making two **complete decodes** (the normal case of decode), one in the server and one in the addressee, it is only necessary to make one **specialized decode**(in the server) and another complete decode(in the addressee). This section describes the following two specialized decodes, which support to solve this and similar problems:

- **Exclusive decode**
- **Selected decode**

This functionality is only provided when using `BER` (option `ber`).

1.4.1 Exclusive Decode

The basic idea with exclusive decode is to specify which parts of the message you want to exclude from being decoded. These parts remain encoded and are returned in the value structure as binaries. They can be decoded in turn by passing them to a certain `decode_part/2` function. The performance gain is high for large messages. You can do an exclusive decode and later one or more decodes of the parts, or a second complete decode instead of two or more complete decodes.

Procedure

To perform an exclusive decode:

- **Step 1:** Decide the name of the function for the exclusive decode.
- **Step 2:** Include the following instructions in a configuration file:
  - The name of the exclusive decode function
  - The name of the ASN.1 specification
  - A notation that tells which parts of the message structure to be excluded from decode
- **Step 3** Compile with the additional option `asn1config`. The compiler searches for a configuration file with the same name as the ASN.1 specification but with extension `.asn1config`. This configuration file is not the same as used for compilation of a set of files. See Section Writing an Exclusive Decode Instruction.

User Interface

The runtime user interface for exclusive decode consists of the following two functions:

- A function for an exclusive decode, whose name the user decides in the configuration file
- The compiler generates a `decode_part/2` function when exclusive decode is chosen. This function decodes the parts that were left undecoded during the exclusive decode.

Both functions are described in the following.

If the exclusive decode function has, for example, the name `decode_exclusive` and an ASN.1 encoded message `Bin` is to be exclusive decoded, the call is as follows:

```
{ok,Excl_Message} = 'MyModule':decode_exclusive(Bin)
```

The result `Excl_Message` has the same structure as a complete decode would have, except for the parts of the top type that were not decoded. The undecoded parts are on their places in the structure on format `{Type_Key,Undecoded_Value}`.

Each undecoded part that is to be decoded must be fed into function `decode_part/2` as follows:

```
{ok,Part_Message} = 'MyModule':decode_part(Type_Key,Undecoded_Value)
```

Writing an Exclusive Decode Instruction

This instruction is written in the configuration file in the following format:

```
Exclusive_Decode_Instruction = {exclusive_decode,{Module_Name,Decode_Instructions}}.

Module_Name = atom()

Decode_Instructions = [Decode_Instruction]+

Decode_Instruction = {Exclusive_Decode_Function_Name,Type_List}

Exclusive_Decode_Function_Name = atom()

Type_List = [Top_Type,Element_List]

Element_List = [Element]+

Element = {Name,parts} |
          {Name,undecoded} |
          {Name,Element_List}

Top_Type = atom()

Name = atom()
```

The instruction must be a valid Erlang term ended by a dot.

In `Type_List` the "path" from the top type to each undecoded subcomponents is described. The top type of the path is an atom, the name of it. The action on each component/type that follows is described by one of `{Name,parts}`, `{Name,undecoded}`, `{Name,Element_List}`.

The use and effect of the actions are as follows:

- `{Name,undecoded}` - Tells that the element is left undecoded during the exclusive decode. The type of `Name` can be any ASN.1 type. The value of element `Name` is returned as a tuple (as mentioned in the previous section) in the value structure of the top type.
- `{Name,parts}` - The type of `Name` can be one of `SEQUENCE OF` or `SET OF`. The action implies that the different components of `Name` are left undecoded. The value of `Name` is returned as a tuple (as mentioned in the previous section) where the second element is a list of binaries. This is because the representation of a

SEQUENCE OF or a SET OF in Erlang is a list of its internal type. Any of the elements in this list or the entire list can be decoded by function decode_part.

- {Name,Element_List} - This action is used when one or more of the subtypes of Name is exclusive decoded.

Name in these actions can be a component name of a SEQUENCE OF or a SET OF, or a name of an alternative in a CHOICE.

Example

In this examples, the definitions from the following ASN.1 specification are used:

```
GUI DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

Action ::= SEQUENCE
 {
   number  INTEGER DEFAULT 15,
   handle  [0] Handle DEFAULT {number  12, on  TRUE}
 }

Key ::= [11] EXPLICIT Button
Handle ::= [12] Key
Button ::= SEQUENCE
 {
   number  INTEGER,
   on  BOOLEAN
 }

Window ::= CHOICE
 {
   vsn INTEGER,
   status E
 }

Status ::= SEQUENCE
 {
   state INTEGER,
   buttonList SEQUENCE OF Button,
   enabled BOOLEAN OPTIONAL,
   actions CHOICE {
     possibleActions SEQUENCE OF Action,
     noOfActions INTEGER
   }
 }


END
```

If Button is a top type and it is needed to exclude component number from decode, Type_List in the instruction in the configuration file is ['Button',[{number,undecoded}]]. If you call the decode function decode_Button_exclusive, Decode_Instruction is {decode_Button_exclusive, ['Button',[{number,undecoded}]]}.

Another top type is Window whose subcomponent actions in type Status and the parts of component buttonList are to be left undecoded. For this type, the function is named decode__Window_exclusive. The complete Exclusive_Decode_Instruction configuration is as follows:

```
{exclusive_decode,{'GUI',
 [{decode_Window_exclusive,['Window',[{status,[{buttonList,parts},{actions,undecoded}]}]]},
  {decode_Button_exclusive,['Button',[{number,undecoded}]]}]}}.
```

The following figure shows the bytes of a `Window:status` message. The components `buttonList` and `actions` are excluded from decode. Only `state` and `enabled` are decoded when `decode__Window_exclusive` is called.

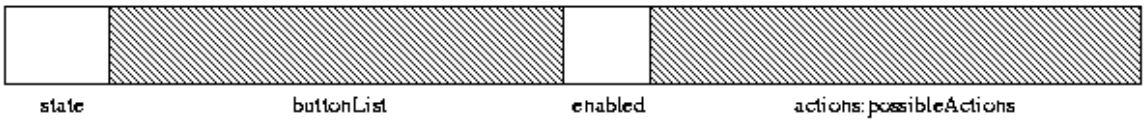

Figure 4.1: Bytes of a Window:status Message

Compiling `GUI.asn` including the configuration file is done as follows:

```
unix> erlc -bber +asn1config GUI.asn

erlang> asn1ct:compile('GUI', [ber,asn1config]).
```

The module can be used as follows:

```
1> Button_Msg = {'Button',123,true}.
{'Button',123,true}
2> {ok,Button_Bytes} = 'GUI':encode('Button',Button_Msg).
{ok,[<<48>>,
     [6],
     [<<128>>,
      [1],
      123],
     [<<129>>,
      [1],
      255]]}
3> {ok,Exclusive_Msg_Button} = 'GUI':decode_Button_exclusive(list_to_binary(Button_Bytes)).
{ok,{'Button',{'Button_number',<<28,1,123>>},
         true}}
4> 'GUI':decode_part('Button_number',<<128,1,123>>).
{ok,123}
5> Window_Msg =
{'Window',{status,{'Status',35,
              [{'Button',3,true},
               {'Button',4,false},
               {'Button',5,true},
               {'Button',6,true},
               {'Button',7,false},
               {'Button',8,true},
               {'Button',9,true},
               {'Button',10,false},
               {'Button',11,true},
               {'Button',12,true},
               {'Button',13,false},
               {'Button',14,true}],
              false,
              {possibleActions,[{'Action',16,{'Button',17,true}}]}}}}.
{'Window',{status,{'Status',35,
              [{'Button',3,true},
               {'Button',4,false},
               {'Button',5,true},
               {'Button',6,true},
               {'Button',7,false},
               {'Button',8,true},
               {'Button',9,true},
               {'Button',10,false},
               {'Button',11,true},
               {'Button',12,true},
               {'Button',13,false},
               {'Button',14,true}],
              false,
              {possibleActions,[{'Action',16,{'Button',17,true}}]}}}}
6> {ok,Window_Bytes}='GUI':encode('Window',Window_Msg).
{ok,[<<161>>,
     [127],
     [<<128>>, ...


8> {ok,{status,{'Status',Int,{Type_Key_SeqOf,Val_SEQOF},
BoolOpt,{Type_Key_Choice,Val_Choice}}}}=
'GUI':decode_Window_status_exclusive(list_to_binary(Window_Bytes)).
{ok,{status,{'Status',35,
        {'Status_buttonList',[<<48,6,128,1,3,129,1,255>>,
                              <<48,6,128,1,4,129,1,0>>,
                              <<48,6,128,1,5,129,1,255>>,
                              <<48,6,128,1,6,129,1,255>>,
                              <<48,6,128,1,7,129,1,0>>,
                              <<48,6,128,1,8,129,1,255>>,
                              <<48,6,128,1,9,129,1,255>>,
```

```
                              <<48,6,128,1,10,129,1,0>>,
                              <<48,6,128,1,11,129,1,255>>,
                              <<48,6,128,1,12,129,1,255>>,
                              <<48,6,128,1,13,129,1,0>>,
                              <<48,6,128,1,14,129,1,255>>]},
        false,
        {'Status_actions',
<<163,21,160,19,48,17,2,1,16,160,12,172,10,171,8,48,6,128,1,...>>}}}}
10> 'GUI':decode_part(Type_Key_SeqOf,Val_SEQOF).
{ok,[{'Button',3,true},
     {'Button',4,false},
     {'Button',5,true},
     {'Button',6,true},
     {'Button',7,false},
     {'Button',8,true},
     {'Button',9,true},
     {'Button',10,false},
     {'Button',11,true},
     {'Button',12,true},
     {'Button',13,false},
     {'Button',14,true}]}
11> 'GUI':decode_part(Type_Key_SeqOf,hd(Val_SEQOF)).
{ok,{'Button',3,true}}
12> 'GUI':decode_part(Type_Key_Choice,Val_Choice).
{ok,{possibleActions,[{'Action',16,{'Button',17,true}}]}}
```

1.4.2 Selective Decode

This specialized decode decodes a subtype of a constructed value and is the fastest method to extract a subvalue. This decode is typically used when you want to inspect, for example, a version number, to be able to decide what to do with the entire value. The result is returned as `{ok,Value}` or `{error,Reason}`.

Procedure

To perform a selective decode:

- **Step 1:** Include the following instructions in the configuration file:
  - The name of the user function
  - The name of the ASN.1 specification
  - A notation that tells which part of the type to be decoded
- **Step 2:** Compile with the additional option `asn1config`. The compiler searches for a configuration file with the same name as the ASN.1 specification, but with extension `.asn1config`. In the same file you can also provide configuration specifications for exclusive decode. The generated Erlang module has the usual functionality for encode/decode preserved and the specialized decode functionality added.

User Interface

The only new user interface function is the one provided by the user in the configuration file. The function is started by the `ModuleName:FunctionName` notation.

For example, if the configuration file includes the specification `{selective_decode, {'ModuleName',[{selected_decode_Window,TypeList}]}}` do the selective decode by `{ok,Result}='ModuleName':selected_decode_Window(EncodedBinary).`

Writing a Selective Decode Instruction

One or more selective decode functions can be described in a configuration file. Use the following notation:

```
Selective_Decode_Instruction = {selective_decode,{Module_Name,Decode_Instructions}}.

Module_Name = atom()

Decode_Instructions = [Decode_Instruction]+

Decode_Instruction = {Selective_Decode_Function_Name,Type_List}

Selective_Decode_Function_Name = atom()

Type_List = [Top_Type|Element_List]

Element_List = Name|List_Selector

Name = atom()

List_Selector = [integer()]
```

The instruction must be a valid Erlang term ended by a dot.

- `Module_Name` is the same as the name of the ASN.1 specification, but without the extension.
- `Decode_Instruction` is a tuple with your chosen function name and the components from the top type that leads to the single type you want to decode. Ensure to choose a name of your function that is not the same as any of the generated functions.
- The first element of `Type_List` is the top type of the encoded message. In `Element_List`, it is followed by each of the component names that leads to selected type.
- Each name in `Element_List` must be a constructed type except the last name, which can be any type.
- `List_Selector` makes it possible to choose one of the encoded components in a `SEQUENCE OF` or a `SET OF`. It is also possible to go further in that component and pick a subtype of that to decode. So, in the `Type_List`: `['Window',status,buttonList,[1],number]`, component `buttonList` must be of type `SEQUENCE OF` or `SET OF`.

In the example, component `number` of the first of the encoded elements in the `SEQUENCE OF` `buttonList` is selected. This applies on the ASN.1 specification in Section Writing an Exclusive Decode Instruction.

Another Example

In this example, the same ASN.1 specification as in Section Writing an Exclusive Decode Instruction is used. The following is a valid selective decode instruction:

```
{selective_decode,
    {'GUI',
        [{selected_decode_Window1,
            ['Window',status,buttonList,
             [1],
             number]},
 {selected_decode_Action,
     ['Action',handle,number]},
 {selected_decode_Window2,
     ['Window',
      status,
      actions,
      possibleActions,
      [1],
      handle,number]}]}}.
```

The first instruction, `{selected_decode_Window1,['Window',status,buttonList,[1],number]}` is described in the previous section.

The second instruction, `{selected_decode_Action,['Action',handle,number]}`, takes component `number` in the `handle` component of type `Action`. If the value is `ValAction = {'Action',17,{'Button',4711,false}}`, the internal value 4711 is to be picked by `selected_decode_Action`. In an Erlang terminal it looks as follows:

```
ValAction = {'Action',17,{'Button',4711,false}}.
{'Action',17,{'Button',4711,false}}
7> {ok,Bytes}='GUI':encode('Action',ValAction).
...
8> BinBytes = list_to_binary(Bytes).
<<48,18,2,1,17,160,13,172,11,171,9,48,7,128,2,18,103,129,1,0>>
9> 'GUI':selected_decode_Action(BinBytes).
{ok,4711}
10>
```

The third instruction, `['Window',status,actions,possibleActions,[1],handle,number]`, works as follows:

- **Step 1:** Starts with type `Window`.
- **Step 2:** Takes component `status` of `Window` that is of type `Status`.
- **Step 3:** Takes **actions** of type `Status`.
- **Step 4:** Takes `possibleActions` of the internally defined `CHOICE` type.
- **Step 5:** Goes into the first component of `SEQUENCE OF` by `[1]`. That component is of type `Action`.
- **Step 6:** Takes component `handle`.
- **Step 7:** Takes component `number` of type `Button`.

The following figure shows which components are in `TypeList ['Window',status,actions,possibleActions,[1],handle,number]`:

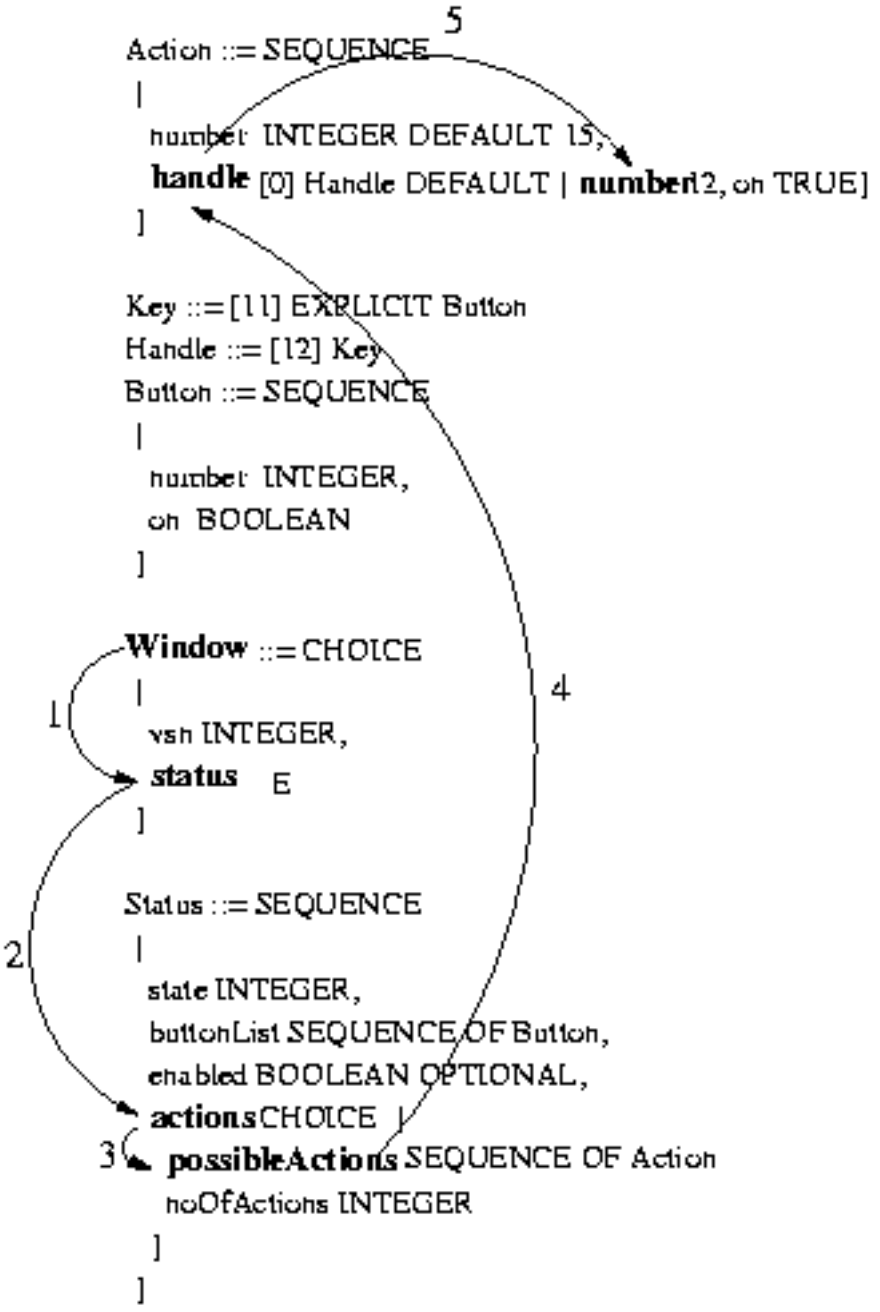

Figure 4.2: Elements Specified in Configuration File for Selective Decode of a Subvalue in a Window Message

In the following figure, only the marked element is decoded by `selected_decode_Window2`:

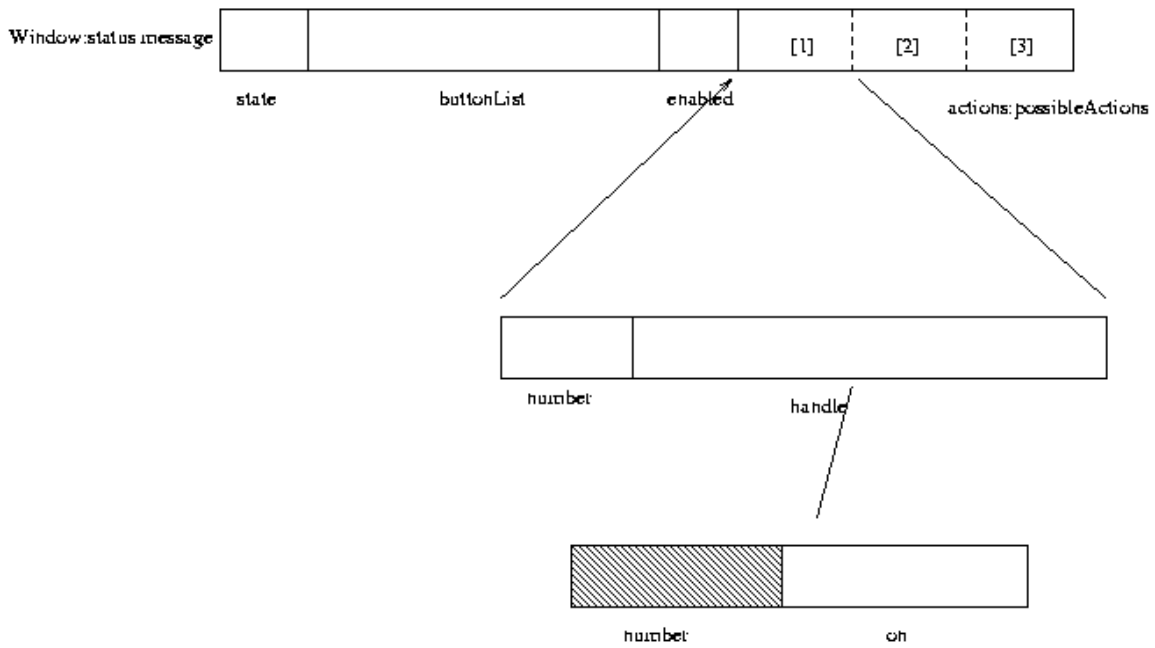

Figure 4.3: Bytes of a Window:status Message

With the following example, you can examine that both `selected_decode_Window2` and `selected_decode_Window1` decodes the intended subvalue of value `Val`:

```
1> Val = {'Window',{status,{'Status',12,
                    [{'Button',13,true},
                     {'Button',14,false},
                     {'Button',15,true},
                     {'Button',16,false}],
                    true,
                    {possibleActions,[{'Action',17,{'Button',18,false}},
                                      {'Action',19,{'Button',20,true}},
                                      {'Action',21,{'Button',22,false}}]}}}}
2> {ok,Bytes}='GUI':encode('Window',Val).
...
3> Bin = list_to_binary(Bytes).
<<161,101,128,1,12,161,32,48,6,128,1,13,129,1,255,48,6,128,1,14,129,1,0,48,6,128,1,15,129,...>>
4> 'GUI':selected_decode_Window1(Bin).
{ok,13}
5> 'GUI':selected_decode_Window2(Bin).
{ok,18}
```

Notice that the value fed into the selective decode functions must be a binary.

1.4.3 Performance

To give an indication on the possible performance gain using the specialized decodes, some measures have been performed. The relative figures in the outcome between selective, exclusive, and complete decode (the normal case) depend on the structure of the type, the size of the message, and on what level the selective and exclusive decodes are specified.

ASN.1 Specifications, Messages, and Configuration

The specifications GUI and **MEDIA-GATEWAY-CONTROL** were used in the test.

For the `GUI` specification the configuration was as follows:

```
{selective_decode,
  {'GUI',
    [{selected_decode_Window1,
         ['Window',
          status,buttonList,
          [1],
          number]},
     {selected_decode_Window2,
         ['Window',
          status,
          actions,
          possibleActions,
          [1],
          handle,number]}]}}.
     {exclusive_decode,
         {'GUI',
            [{decode_Window_status_exclusive,
                ['Window',
                 [{status,
                     [{buttonList,parts},
                      {actions,undecoded}]}]]}]}}.
```

The `MEDIA-GATEWAY-CONTROL` configuration was as follows:

```
{exclusive_decode,
  {'MEDIA-GATEWAY-CONTROL',
    [{decode_MegacoMessage_exclusive,
        ['MegacoMessage',
         [{authHeader,undecoded},
          {mess,
             [{mId,undecoded},
              {messageBody,undecoded}]}]]}]}}.
{selective_decode,
  {'MEDIA-GATEWAY-CONTROL',
    [{decode_MegacoMessage_selective,
         ['MegacoMessage',mess,version]}]}}.
```

The corresponding values were as follows:

```
{'Window',{status,{'Status',12,
              [{'Button',13,true},
               {'Button',14,false},
               {'Button',15,true},
               {'Button',16,false},
               {'Button',13,true},
               {'Button',14,false},
               {'Button',15,true},
               {'Button',16,false},
               {'Button',13,true},
               {'Button',14,false},
               {'Button',15,true},
               {'Button',16,false}],
              true,
              {possibleActions,
                 [{'Action',17,{'Button',18,false}},
                  {'Action',19,{'Button',20,true}},
                  {'Action',21,{'Button',22,false}},
                  {'Action',17,{'Button',18,false}},
                  {'Action',19,{'Button',20,true}},
                  {'Action',21,{'Button',22,false}},
                  {'Action',17,{'Button',18,false}},
                  {'Action',19,{'Button',20,true}},
                  {'Action',21,{'Button',22,false}},
                  {'Action',17,{'Button',18,false}},
                  {'Action',19,{'Button',20,true}},
                  {'Action',21,{'Button',22,false}},
                  {'Action',17,{'Button',18,false}},
                  {'Action',19,{'Button',20,true}},
                  {'Action',21,{'Button',22,false}}]}}}}


{'MegacoMessage',asn1_NOVALUE,
  {'Message',1,
    {ip4Address,
      {'IP4Address',[125,125,125,111],55555}},
  {transactions,
    [{transactionReply,
      {'TransactionReply',50007,asn1_NOVALUE,
       {actionReplies,
        [{'ActionReply',0,asn1_NOVALUE,asn1_NOVALUE,
          [{auditValueReply,{auditResult,{'AuditResult',
            {'TerminationID',[],[255,255,255]},
             [{mediaDescriptor,
               {'MediaDescriptor',asn1_NOVALUE,
                {multiStream,
                 [{'StreamDescriptor',1,
                   {'StreamParms',
                    {'LocalControlDescriptor',
                     sendRecv,
                     asn1_NOVALUE,
                     asn1_NOVALUE,
                     [{'PropertyParm',
                       [0,11,0,7],
                       [[52,48]],
                       asn1_NOVALUE}]},
                    {'LocalRemoteDescriptor',
                     [[{'PropertyParm',
                        [0,0,176,1],
                        [[48]],
                        asn1_NOVALUE},
```

```
                    {'PropertyParm',
                      [0,0,176,8],
                      [[73,78,32,73,80,52,32,49,50,53,46,49,
                        50,53,46,49,50,53,46,49,49,49]],
                      asn1_NOVALUE},
                    {'PropertyParm',
                      [0,0,176,15],
                      [[97,117,100,105,111,32,49,49,49,49,32,
                        82,84,80,47,65,86,80,32,32,52]],
                      asn1_NOVALUE},
                    {'PropertyParm',
                      [0,0,176,12],
                      [[112,116,105,109,101,58,51,48]],
                      asn1_NOVALUE}]]},
                  {'LocalRemoteDescriptor',
                   [[{'PropertyParm',
                      [0,0,176,1],
                      [[48]],
                      asn1_NOVALUE},
                    {'PropertyParm',
                      [0,0,176,8],
                      [[73,78,32,73,80,52,32,49,50,52,46,49,50,
                        52,46,49,50,52,46,50,50,50]],
                      asn1_NOVALUE},
                    {'PropertyParm',
                      [0,0,176,15],
                      [[97,117,100,105,111,32,50,50,50,50,32,82,
                        84,80,47,65,86,80,32,32,52]],
                      asn1_NOVALUE},
                    {'PropertyParm',
                      [0,0,176,12],
                      [[112,116,105,109,101,58,51,48]],
                      asn1_NOVALUE}]]}}]}}},
             {packagesDescriptor,
              [{'PackagesItem',[0,11],1},
               {'PackagesItem',[0,11],1}]},
             {statisticsDescriptor,
              [{'StatisticsParameter',[0,12,0,4],[[49,50,48,48]]},
               {'StatisticsParameter',[0,11,0,2],[[54,50,51,48,48]]},
               {'StatisticsParameter',[0,12,0,5],[[55,48,48]]},
               {'StatisticsParameter',[0,11,0,3],[[52,53,49,48,48]]},
               {'StatisticsParameter',[0,12,0,6],[[48,46,50]]},
               {'StatisticsParameter',[0,12,0,7],[[50,48]]},
               {'StatisticsParameter',[0,12,0,8],[[52,48]]}]}}}}}]}}}]}}}
```

The size of the encoded values was 458 bytes for `GUI` and 464 bytes for `MEDIA-GATEWAY-CONTROL`.

Results

The ASN.1 specifications in the test were compiled with options `ber_bin`, `optimize`, `driver` and `asn1config`. Omitting option `driver` gives higher values for `decode` and `decode_part`. These tests have not been rerun using NIFs, but are expected to perform about 5% better than the linked-in driver.

The test program runs 10000 decodes on the value, resulting in an output with the elapsed time in microseconds for the total number of decodes.

Function	**Time** (microseconds)	**Decode Type**	**ASN.1 Specification**	**% of Time versus Complete Decode**
`decode_MegacoMessage_selective/1`	37045	`selective`	`MEDIA-GATEWAY-CONTROL`	**8.3**

decode_MegacoMessage_exclusive/1	621107	Exclusive	MEDIA-GATEWAY-CONTROL	**13.8**
decode/2	4507457	Complete	MEDIA-GATEWAY-CONTROL	**100**
selected_decode_Window1/1	419585	Selective	GUI	**7.6**
selected_decode_Window2/1	890666	Selective	GUI	**15.1**
decode_Window_status_exclusive/1	1251878	Exclusive	GUI	**21.3**
decode/2	5889197	Complete	GUI	**100**

Table 4.1: Results of Complete, Exclusive, and Selective Decode

It is also of interest to know the relation is between a complete decode, an exclusive decode followed by `decode_part` of the excluded parts, and a selective decode followed by a complete decode. Some situations can be compared to this simulation, for example, inspect a subvalue and later inspect the entire value. The following table shows figures from this test. The number of loops and the time unit are the same as in the previous test.

Actions	**Function**	**Time** (microseconds)	**ASN.1 Specification**	**% of Time vs. Complete Decode**
Complete	decode/2	4507457	MEDIA-GATEWAY-CONTROL	**100**
Selective and Complete	decode_-MegacoMessage_-selective/1	4881502	MEDIA-GATEWAY-CONTROL	**108.3**
Exclusive and decode_part	decode_-MegacoMessage_-exclusive/1	5481034	MEDIA-GATEWAY-CONTROL	**112.3**
Complete	decode/2	5889197	GUI	**100**
Selective and Complete	selected_-decode_-Window1/1	6337636	GUI	**107.6**
Selective and Complete	selected_-decode_-Window2/1	6795319	GUI	**115.4**
Exclusive and decode_part	decode_-Window_-status_-exclusive/1	6249200	GUI	**106.1**

Table 4.2: Results of Complete, Exclusive + decode_part, and Selective + complete decodes

Other ASN.1 types and values can differ much from these figures. It is therefore important that you, in every case where you intend to use either of these decodes, perform some tests that show if you will benefit your purpose.

Final Remarks

- The gain of using selective and exclusive decode instead of a complete decode is greater the bigger the value and the less deep in the structure you have to decode.
- Use selective decode instead of exclusive decode if you are interested in only a single subvalue.
- Exclusive decode followed by `decode_part` decodes is attractive if the parts are sent to different servers for decoding, or if you in some cases are not interested in all parts.
- The fastest selective decode is when the decoded type is a primitive type and not so deep in the structure of the top type. `selected_decode_Window2` decodes a high constructed value, which explains why this operation is relatively slow.
- It can vary from case to case which combination of selective/complete decode or exclusive/part decode is the fastest.

2 Reference Manual

The `ASN.1` application contains modules with compile-time and runtime support for ASN.1.

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 specified data types. Alternatively, the compiler takes a specification module specifying all input modules, and generates a module with encode/decode functions. In addition, some generic functions can be used during development of applications that handles ASN.1 data (encoded as BER or PER).

Note:

By default in OTP 17, the representation of the `BIT STRING` and `OCTET STRING` types as Erlang terms were changed. `BIT STRING` values are now Erlang bit strings and `OCTET STRING` values are binaries. Also, an undecoded open type is now wrapped in an `asn1_OPENTYPE` tuple. For details, see BIT STRING, OCTET STRING, and ASN.1 Information Objects in the User's Guide.

To revert to the old representation of the types, use option `legacy_erlang_types`.

Note:

In OTP R16, the options were simplified. The back end is chosen using one of the options `ber`, `per`, `uper` or `jer`. Options `optimize`, `nif`, and `driver` options are no longer necessary (and the ASN.1 compiler generates a warning if they are used). Options `ber_bin`, `per_bin`, and `uper_bin` options still work, but generates a warning.

Another change in OTP R16 is that the generated function `encode/2` always returns a binary. Function `encode/2` for the BER back end used to return an iolist.

Exports

```
compile(Asn1module) -> ok | {error, Reason}
compile(Asn1module, Options) -> ok | {error, Reason}
```

Types:

```
Asn1module = atom() | string()
Options = [Option| OldOption]
Option = ber | per | uper | jer | der | compact_bit_string |
legacy_bit_string | legacy_erlang_types | noobj | {n2n, EnumTypeName} |
{outdir, Dir} | {i, IncludeDir} | asn1config | undec_rest | no_ok_wrapper
| {macro_name_prefix, Prefix} | {record_name_prefix, Prefix} | verbose |
warnings_as_errors | deterministic
OldOption = ber | per
Reason = term()
Prefix = string()
```

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 that 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). `Asn1module` can be a full pathname (relative or absolute) including filename with (or without) extension.

If it is needed to compile a set of `ASN.1` modules into an Erlang file with encode/decode functions, ensure 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"`). List the input file names 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 must look as follows:

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

The output files in this case get their names from the configuration file. If the configuration file is named `SetOfFiles.set.asn`, the names of the output files are `SetOfFiles.hrl, SetOfFiles.erl, and SetOfFiles.asn1db`.

Sometimes in a system of `ASN.1` modules, different default tag modes, for example, `AUTOMATIC`, `IMPLICIT`, or `EXPLICIT`. The multi-file compilation resolves the default tagging as if the modules were compiled separately.

Name collisions is another unwanted effect that can occur in multi file-compilation. The compiler solves this problem in one of two ways:

- If the definitions are identical, the output module keeps only one definition with the original name.
- If the definitions have the same name and differs in the definition, they are renamed. The new names are the definition name and the original module name concatenated.

If a name collision occurs, the compiler reports a `"NOTICE: ..."` message that tells if a definition was renamed, and the new name that must be used to encode/decode data.

`Options` is a list with options specific for the `ASN.1` compiler and options that are applied to the Erlang compiler. The latter are not recognized as `ASN.1` specific. The available options are as follows:

`ber | per | uper | jer`

The encoding rule to be used. The supported encoding rules are Basic Encoding Rules (`ber`), Packed Encoding Rules (`per`) aligned, PER unaligned (`uper`) and JSON Encoding Rules (`jer`). The `jer` option can be used by itself to generate a module that only supports encoding/decoding to JER or it can be used as a supplementary option to ber, per and uper. In the latter case a module with for both the main encoding rules and JER will be generated. The exported functions for JER will then be `jer_encode(Type, Value)` and `jer_decode(Type, Bytes)`.

The `jer` encoding rules (ITU-T X.697) are experimental in OTP 22. There is support for a subset of the X.697 standard, for example there is no support for:

- JER encoding instructions
- the REAL type

Also note that when using the `jer` encoding rules the generated module will get a dependency to an external json component. The generated code is currently tested together with:

- `jsx` which currently is the default.
- `jsone` can be chosen instead of `jsx` by providing the option `{d,jsone}`.

If the encoding rule option is omitted, `ber` is the default.

The generated Erlang module always gets the same name as the `ASN.1` module. Therefore, only one encoding rule per `ASN.1` module can be used at runtime.

`der`

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

`maps`

This option changes the representation of the types `SEQUENCE` and `SET` to use maps (instead of records). This option also suppresses the generation of `.hrl` files.

For details, see Section Map representation for SEQUENCE and SET in the User's Guide.

`compact_bit_string`

The `BIT STRING` type is decoded to "compact notation". **This option is not recommended for new code.** This option cannot be combined with the option `maps`.

For details, see Section BIT STRING in the User's Guide.

This option implies option `legacy_erlang_types`.

`legacy_bit_string`

The `BIT STRING` type is decoded to the legacy format, that is, a list of zeroes and ones. **This option is not recommended for new code.** This option cannot be combined with the option `maps`.

For details, see Section BIT STRING in the User's Guide

This option implies option `legacy_erlang_types`.

`legacy_erlang_types`

Use the same Erlang types to represent `BIT STRING` and `OCTET STRING` as in OTP R16.

For details, see Section BIT STRING and Section OCTET STRING in the User's Guide.

This option is not recommended for new code. This option cannot be combined with the option `maps`.

`{n2n, EnumTypeName}`

Tells the compiler to generate functions for conversion between names (as atoms) and numbers and conversely for the specified `EnumTypeName`. There can be multiple occurrences of this option to specify several type names. The type names must be declared as `ENUMERATIONS` in the ASN.1 specification.

If `EnumTypeName` does not exist in the ASN.1 specification, the compilation stops with an error code.

The generated conversion functions are named `name2num_EnumTypeName/1` and `num2name_EnumTypeName/1`.

`noobj`

Do not compile (that is, do not produce object code) the generated `.erl` file. If this option is omitted, the generated Erlang module is compiled.

`{i, IncludeDir}`

Adds `IncludeDir` to the search-path for `.asn1db` and `ASN.1` source files. The compiler tries to open an `.asn1db` file when a module imports definitions from another `ASN.1` module. If no `.asn1db` file is found, the `ASN.1` source file is parsed. Several `{i, IncludeDir}` can be given.

`{outdir, Dir}`

Specifies directory `Dir` where all generated files are to be placed. If this option is omitted, the files are placed in the current directory.

`asn1config`

When using one of the specialized decodes, exclusive or selective decode, instructions must be given in a configuration file. Option `asn1config` enables specialized decodes and takes the configuration file in concern. The configuration file has the same name as the ASN.1 specification, but with extension `.asn1config`.

For instructions for exclusive decode, see Section Exclusive Decode in the User's Guide.

For instructions for selective decode, see Section Selective Decode in the User's Guide.

`undec_rest`

A buffer that holds a message, being decoded it can also have some following bytes. Those following bytes can now be returned together with the decoded value. If an ASN.1 specification is compiled with this option, a tuple `{ok, Value, Rest}` is returned. `Rest` can be a list or a binary. Earlier versions of the compiler ignored those following bytes.

`no_ok_wrapper`

With this option, the generated `encode/2` and `decode/2` functions do not wrap a successful return value in an `{ok,...}` tuple. If any error occurs, an exception will be raised.

`{macro_name_prefix, Prefix}`

All macro names generated by the compiler are prefixed with `Prefix`. This is useful when multiple protocols that contain macros with identical names are included in a single module.

`{record_name_prefix, Prefix}`

All record names generated by the compiler are prefixed with `Prefix`. This is useful when multiple protocols that contain records with identical names are included in a single module.

`verbose`

Causes more verbose information from the compiler describing what it is doing.

`warnings_as_errors`

Causes warnings to be treated as errors.

`deterministic`

Causes all non-deterministic options to be stripped from the -asn1_info() attribute.

Any more option that is applied is passed to the final step when the generated `.erl` file is compiled.

The compiler generates the following files:

- `Asn1module.hrl` (if any `SET` or `SEQUENCE` is defined)
- `Asn1module.erl` - Erlang module with encode, decode, and value functions
- `Asn1module.asn1db` - Intermediate format used by the compiler when modules `IMPORT` definitions from each other.

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

Types:

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

Returns an Erlang term that 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.

> **Note:**
>
> Currently, the `value` function has many limitations. Essentially, it will mostly work for old specifications based on the 1997 standard for ASN.1, but not for most modern-style applications. Another limitation is that the `value` function may not work if options that change code generations strategies such as the options `macro_name_prefix` and `record_name_prefix` have been used.

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

Types:

```
Module = Type = atom()
Value = term()
Options = [{i, IncludeDir}]
Reason = term()
```

Performs a test of encode and decode of 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 as well as the general runtime support work as expected.

> **Note:**
>
> Currently, the `test` functions have many limitations. Essentially, they will mostly work for old specifications based on the 1997 standard for ASN.1, but not for most modern-style applications. Another limitation is that the `test` functions may not work if options that change code generations strategies such as the options `macro_name_prefix` and `record_name_prefix` have been used.

- `test/1` iterates over all types in `Module`.
- `test/2` tests type `Type` with a random value.
- `test/3` tests type `Type` with `Value`.

Schematically, the following occurs for each type in the module:

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

The `test` functions use the `*.asn1db` files for all included modules. If they are located in a different directory than the current working directory, use the `include` option to add paths. This is only needed when automatically generating values. For static values using `Value` no options are needed.