ASN.1

ASN.1 1.6.16
March 21 2011

March 21 2011

1 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.
- The supported encoding rules are:
  - 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.

Overview

ASN.1 (Abstract Syntax Notation 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 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 [] which is the primary text. It may also be helpful, but not necessary, to read the standard definitions [] [] [] [] [].

A very good book explaining those reference texts is [], free to download at **http://www.oss.com/asn1/dubuisson.html** .

Capability

This application covers all features of ASN.1 up to the 1997 edition of the specification. In the 2002 edition of ASN.1 a number of new features where introduced of which some are supported while others are not. For example the ECN (Encoding Control Notation) and XML notation are still unsupported. Though, the other features of the 2002 edition are fully or partly supported as shown below:

- Decimal notation (e.g., "1.5e3") for REAL values. The NR1, NR2 and NR3 formats as explained in ISO6093 are supported.
- The RELATIVE-OID type for relative object identifiers are 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.

It should also be added here that the encoding formats supported are *BER, DER, PER aligned basic* variant and *PER unaligned basic* variant.

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. The code-generator then uses the abstract syntax tree in order to generate code.

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

```
1>asn1ct:compile("People",[ber_bin]).
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 the abstract syntax tree is saved in the `People.asn1db` file, 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,
```

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.

```
2> Rockstar = {'Person',"Some Name",roving,50}.
{'Person',"Some Name",roving,50}
3> {ok,Bytes} = asn1rt:encode('People','Person',Rockstar).
{ok,[<<243>>,
     [17],
     [19,9,"Some Name"],
     [2,1,[2]],
     [2,1,"2"]]}
4> Bin = list_to_binary(Bytes).
<<243,17,19,9,83,111,109,101,32,78,97,109,101,2,1,2,2,1,50>>
5> {ok,Person} = asn1rt:decode('People','Person',Bin).
{ok,{'Person',"Some Name",roving,50}}
6>
```

Notice that the result from `encode` is a nested list which must be turned into a binary before the call to `decode`. A binary is necessary as input to decode since the module was compiled with the `ber_bin` option The reason for returning a nested list is that it is faster to produce and the `list_to_binary` operation is performed automatically when the list is sent via the Erlang port mechanism.

Module dependencies

It is common that asn1 modules import defined types, values and other entities from another asn1 module.

Earlier versions of the asn1 compiler required that modules that were imported from had to be compiled before the module that imported. This caused problems when asn1 modules had circular dependencies.

Now are referenced modules parsed when the compiler finds an entity that is imported. There will not be any code generated for the referenced module. However, the compiled module rely on that the referenced modules also will be compiled.

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, ie. 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 Person.asn
erlc -bper_bin Person.asn
erlc -bber_bin +optimize ../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 | uper_bin]`

Choice of encoding rules, if omitted `ber` is the default. The `ber_bin` and `per_bin` options allows for optimizations and are therefore recommended instead of the `ber` and `per` options.

`-o OutDirectory`

Where to put the generated files, default is the current directory.

`-I IncludeDir`

Where to search for `.asn1db` files and asn1 source specs in order to resolve references to 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* type section.

`+der`

DER encoding rule. Only when using `-ber` or `-ber_bin` option.

`+optimize`

This flag has effect only when used together with one of `per_bin` or `ber_bin` flags. It gives time optimized code in the generated modules and it uses another runtime module. In the `per_bin` case a linked-in driver is used. The result from an encode is a binary.

When this flag is used you cannot use the old format `{TypeName,Value}` when you encode values. Since it is an unnecessary construct it has been removed in favor of performance. It is neither admitted to construct SEQUENCE or SET component values with the format `{ComponentName,Value}` since it also is unnecessary. The only case were it is necessary is in a CHOICE, were you have to pass values to the right component by specifying `{ComponentName,Value}`. See also about *{Typename,Value}* below and in the sections for each type.

`+driver`

Together with the flags `ber_bin` and `optimize` you choose to use a linked in driver for considerable faster decode.

`+asn1config`

This functionality works together with the flags `ber_bin` and `optimize`. You enables the specialized decodes, see the *Specialized Decode* chapter.

`+undec_rest`

A buffer that holds a message, being decoded may also have some following bytes. Now it is possible to get those following bytes returned together with the decoded value. If an asn1 spec is compiled with this option a tuple `{ok,Value,Rest}` is returned. `Rest` may be a list or a binary. Earlier versions of the compiler ignored those following bytes.

`{inline,OutputName}`

Compiling with this option gives one output module containing all asn1 run-time functionality. The asn1 specs are provided in a target module `Module.set.asn` as described in the *reference manual*. The name of the resulting module containing generated encode/decode functions and inlined run-time functions will be `OutputName.erl`. The merging/inlining of code is done by the `igor` module of `syntax_tools`. By default the functions generated from the first asn1 spec in the `.set.asn` are exported, unless a `{export,[atom()]}` or `{export_all,true}` option are provided. The list of atoms are names of choosen asn1 specs from the `.set.asn` file. See further examples of usage *below*

`+'Any Erlc Option'`

You may add any option to the Erlang compiler when compiling the generated Erlang files. Any option unrecognised by the asn1 compiler will be passed to the Erlang compiler.

For a complete description of `erlc` see Erts Reference Manual.

For preferred option use see *Preferred Option Use* section.

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*, 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").
```

which equals:

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

If one wants PER encoding with optimizations:

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

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).
```

Preferred Option Use

It may not be obvious which compile options best fit a situation. This section describes the format of the result of encode and decode. It also gives some performance statistics when using certain options. Finally there is a recommendation which option combinations should be used.

The default option is `ber`. It is the same backend as `ber_bin` except that the result of encode is transformed to a flat list. Below is a table that gives the different formats of input and output of encode and decode using the *allowed combinations* of coding and optimization options: (EAVF stands for how ASN1 values are represented in Erlang which is described in the *ASN1 Types chapter*)

Encoding Rule	*Compile options, allowed combinations*	*encode input*	*encode output*	*decode input*	*decode output*
BER	[ber] (default)	EAVF	flat list	flat list / binary	EAVF
BER	[ber_bin]	EAVF	iolist	binary	EAVF
BER	*[ber_bin, optimize]*	EAVF	iolist	binary	EAVF
BER	*[ber_bin, optimize, driver]*	EAVF	iolist	iolist / binary	EAVF
PER aligned variant	[per]	EAVF	flat list	flat list	EAVF
PER aligned variant	*[per_bin]*	EAVF	iolist / binary	binary	EAVF
PER aligned variant	*[per_bin, optimize]*	EAVF	binary	binary	EAVF
PER unaligned variant	*[uper_bin]*	EAVF	binary	binary	EAVF
DER	[(ber), der]	EAVF	flat list	flat list / binary	EAVF

DER	[ber_bin, der]	EAVF	iolist	binary	EAVF
DER	*[ber_bin, optimize, der]*	EAVF	iolist	binary	EAVF
DER	*[ber_bin, optimize, driver, der]*	EAVF	iolist	binary	EAVF

Table 1.1: The output / input formats for different combinations of compile options.

Encode / decode speed comparison in one user case for the above alternatives (except `DER`) is showed in the table below. The `DER` alternatives are slower than their corresponding `BER` alternative.

compile options	*encode time*	*decode time*
[ber]	120	162
[ber_bin]	124	154
[ber_bin, optimize]	50	78
[ber_bin, optimize, driver]	50	62
[per]	141	133
[per_bin]	125	123
[per_bin, optimize]	77	72
[uper_bin]	97	104

Table 1.2: One example of difference in speed for the compile option alternatives.

The sole compile options `ber`, `ber_bin` and `per` are kept for backwards compatibility and should not be used in new code.

You are strongly recommended to use the appropriate alternative of the bold typed options. The `optimize` and `driver` options does not affect the encode or decode result, just the time spent in run-time. When `ber_bin` and `driver` or `per_bin, optimize` and `driver` is combined the C-code driver is used in chosen parts of encode / decode procedure.

Compile options, allowed combinations	*use of linked-in driver*
[ber]	no
[ber_bin]	no
[ber_bin, optimize]	no
[ber_bin, optimize, driver]	yes

[per]	no
[per_bin]	no
[per_bin, optimize]	yes
[uper_bin]	no
[(ber), der]	no
[ber_bin, der]	no
[ber_bin, optimize, der]	no
[ber_bin, optimize, driver, der]	yes

Table 1.3: When the ASN1 linked-in driver is used.

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*, 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).
```

The asn1 linked-in driver is enabled in two occasions: encoding of asn1 values when the asn1 spec is compiled with `per_bin` and `optimize` or decode of encoded asn1 values when the asn1 spec is compiled with `ber_bin`, `optimize` and `driver`. In those cases the driver will be loaded automatically at the first call to `encode`/`decode`. If one doesn't want the performance overhead of the driver being loaded at the first call it is possible to load the driver separately by `asn1rt:load_driver()`.

By invoking the function `info/0` in a generated module, one gets information about which compiler options were used.

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 as default.

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 Multi File Compilation

There are various reasons for using a multi file compilation:

- You want to choose name for the generated module by any reason. Maybe you need to compile the same specs for different encoding/decoding standards.
- You want only one resulting module.
- If it is crucial to have a minimal system. Using `{inline,OutputModule}` includes all necessary run-time functions of the asn1 application, but skips those modules not used.
- Upgrading issues: Even if you upgrade your Erlang system you may want to continue running the old asn1 run-time functionality.
- Performance issues: If you have an asn1 system with a lot of cross references you may gain in performance. Measurements must be done for each case.

You may choose either the plain multi file compilation that just merges the chosen asn1 specs or the `{inline,OutputModule}` that also includes the used asn1 run-time functionality.

For both cases you need to specify which asn1 specs you will compile in a module that must have the extension `.set.asn`. You chose name of the module and provide the names of the asn1 specs. For instance, if you have the specs `File1.asn`, `File2.asn` and `File3.asn` your module `MyModule.set.asn` will look like:

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

If you compile with:

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

the result will be one merged module `MyModule.erl` with the generated code from the three asn1 specs. But if you compile with:

```
~> erlc +"{inline,'OutputModule'}" MyModule.set.asn
```

the result will be a module `OutputModule.erl` that contains all encode/decode functions for the three asn1 specs and all used functions from the asn1 run-time modules, in this case `asn1rt_ber_bin`. In the former case all encode/decode functions are exported but in the latter only the encode/decode functions of the first spec in the `.set.asn`, i.e. those from `File1.asn`.

1.1.5 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	*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	
The TIME types	

Table 1.4: The supported ASN.1 types

> ### Note:
>
> Values of each ASN.1 type has its own representation in Erlang described in the following subsections. Users shall provide these values for encoding according to the representation, as in the example below.

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

In Erlang code it may look like:

```
Val = true,
{ok,Bytes}=asn1rt:encode(MyModule,'Operational',Val),
```

For historical reasons it is also possible to assign ASN.1 values in Erlang using a tuple notation with type and value as this

```
Val = {'Operational',true}
```

> ### Warning:
>
> The tuple notation, `{Typename, Value}` 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.
>
> If the ASN.1 module is compiled with the flags `per_bin` or `ber_bin` and `optimize` it is not allowed to use the tuple notation. That possibility has been removed due to performance reasons. Neither is it allowed to use the `{ComponentName,Value}` notation in case of a SEQUENCE or SET type.

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, for further info see []. 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)}
```

The Erlang representation of an ASN.1 INTEGER is an integer or an atom if a so called `Named Number List` (see T6 above) is specified.

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.

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

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^Exponent.

NULL

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

```
Notype ::= NULL
```

The NULL type can be assigned in Erlang:

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

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.

- A list of binary digits (0 or 1).
- 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.
- A list of atoms corresponding to atoms in the `NamedBitList` in the BIT STRING definition.
- 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:

```
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....",
```

Observe that `O1Val` is assigned a series of numbers between 0 and 255 i.e. octets. `O2Val` 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 (`ber_bin` option)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',list_to_binary(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',list_to_binary(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',list_to_binary(Bytes3)).
{ok,"BMP string"}
```

The UTF8String is represented in Erlang as a list of integers, where each integer represents the unicode value of one character. When a value shall be encoded one first has to transform it to a UTF8 encoded binary, then it can be encoded by asn1. When decoding the result is a UTF8 encoded binary, which may be transformed to an integer list. The transformation functions, `utf8_binary_to_list` and `utf8_list_to_binary`, are in the `asn1rt` module. In the example below we assume an asn1 definition `UTF ::= UTF8String` in a module `UTF.asn`:

```
1> asn1ct:compile('UTF',[ber_bin]).
Erlang ASN.1 version "1.4.3.3" compiling "UTF.asn"
Compiler Options: [ber_bin]
--{generated,"UTF.asn1db"}--
--{generated,"UTF.erl"}--
ok
2> UTF8Val1 = "hello".
"hello"
3> {ok,UTF8bin1} = asn1rt:utf8_list_to_binary(UTF8Val1).
{ok,<<104,101,108,108,111>>}
4> {ok,B}='UTF':encode('UTF',UTF8bin1).
{ok,[12,
     5,
     <<104,101,108,108,111>>]}
5> Bin = list_to_binary(B).
<<12,5,104,101,108,108,111>>
6> {ok,UTF8bin1}='UTF':decode('UTF',Bin).
{ok,<<104,101,108,108,111>>}
7> asn1rt:utf8_binary_to_list(UTF8bin1).
{ok,"hello"}
8> UTF8Val2 = [16#00,16#100,16#ffff,16#ffffff].
[0,256,65535,16777215]
9> {ok,UTF8bin2} = asn1rt:utf8_list_to_binary(UTF8Val2).
{ok,<<0,196,128,239,191,191,248,191,191,191,191>>}
10> {ok,B2} = 'UTF':encode('UTF',UTF8bin2).
{ok,[12,
     11,
     <<0,196,128,239,191,191,248,191,191,191,191>>]}
11> Bin2 = list_to_binary(B2).
<<12,11,0,196,128,239,191,191,248,191,191,191,191>>
12> {ok,UTF8bin2} = 'UTF':decode('UTF',Bin2).
{ok,<<0,196,128,239,191,191,248,191,191,191,191>>}
13> asn1rt:utf8_binary_to_list(UTF8bin2).
```

```
{ok,[0,256,65535,16777215]}
14>
```

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 [], 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.hrl` file.

Values can be assigned in Erlang as shown below:

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

> **Note:**
>
> In very early versions of the asn1 compiler it was also possible to specify the values of the components in a SEQUENCE or a SET as a list of tuples `{ComponentName,Value}`. This is no longer supported.

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 fever 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 specific 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 to 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 [].

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

Moreover, in DER the elements of a SET will be sorted. If a component is an un-tagged 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 Extend-ability for SEQUENCE and SET

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

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

It means that the type may get more components in newer versions of the ASN.1 spec. In this case it has got a new component `b`. Thus, incoming messages that will be decoded may have more or fever components than this one.

The component `b` will be 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 will get the decoded value of the `b` component if present and otherwise the value `asn1_NOVALUE`.

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.

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 may, however, exist in some old ASN.1 modules. The idea with this type was to leave a "hole" in a definition where one could 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`/`ANY DEFINED BY` one should 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

These types are used in presentation layer negotiation. They are encoded according to their associated type, see [].

The `EXTERNAL` type had a slightly different associated type before 1994. [] states that encoding shall follow the older associate type. Therefore does 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 on the newer format.

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}}.
```

1.1.6 Naming of Records in .hrl Files

When an asn1 specification is compiled all defined types of type SET or SEQUENCE will result in a corresponding record in the generated hrl file. This is because the values for SET/SEQUENCE as mentioned in sections above are represented as records.

Though there are some special cases of this functionality that are presented below.

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"}}.
```

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 SEQUENCE, SET or CHOICE types in the line, each component-/alternative-name will be added to the record-name.

For example:

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

will result 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 will give a record with the SEQOF/SETOF addition as in the following example:

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

This results in the records:

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

A parameterized type should be considered as an embedded type. Each time a such 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 used to hold values.

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

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

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'}}}.
```

1.1.7 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.

1.1.8 Macros

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

1.1.9 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 []. 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 referring 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 [] 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.

1.1.10 Parameterization (X.683)

Parameterization, which is defined in the standard [], 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 type-name 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.

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

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

Tag	Len	Value

Table 1.5: Structure of a BER encoded series of bytes

1.1.12 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 not encoded. 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 elements of a SEQUENCE are transmitted in textual order. OPTIONAL or DEFAULT elements are preceded by a bit map to identify which elements are present. After sorting the elements of a SET in the "canonical tag order" as defined in X.680 8.6 they are treated as a SEQUENCE regarding OPTIONAL and DEFAULT elements. A SET is transferred in the sorted order.

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.

1.2 Specialized Decodes

When performance is of highest priority and one is interested in a limited part of the ASN.1 encoded message, before one decide what to do with the rest of it, one may want to decode only this small part. The situation may be a server that has to decide to which addressee it will send a message. The addressee may be interested in the entire message, but the server may be a bottleneck that one 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). The following specialized decodes *exclusive decode* and *selected decode* support to solve this and similar problems.

So far this functionality is only provided when using the optimized BER_BIN version, that is when compiling with the options `ber_bin` and `optimize`. It does also work using the `driver` option. We have no intent to make this available on the default BER version, but maybe in the PER_BIN version (`per_bin`).

1.2.1 Exclusive Decode

The basic idea with exclusive decode is that you 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 may be decoded in turn by passing them to a certain `decode_part/2` function. The performance gain is high when the message is large and you can do an exclusive decode and later on one or several decodes of the parts or a second complete decode instead of two or more complete decodes.

How To Make It Work

In order to make exclusive decode work you have to do the following:

- First,decide the name of the function for the exclusive decode.
- Second, write instructions that must consist of the name of the exclusive decode function, the name of the ASN.1 specification and a notation that tells which parts of the message structure will be excluded from decode. These instructions shall be included in a configuration file.
- Third, compile with the additional option `asn1config`. The compiler searches for a configuration file with the same name as the ASN.1 spec but with the 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 run-time user interface for exclusive decode consists of two different functions. First, the function for an exclusive decode, whose name the user decides in the configuration file. Second, 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 below.

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

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

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

Each undecoded part that shall be decoded must be fed into the `decode_part/2` function,like:

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

Writing an Exclusive Decode Instruction

This instruction is written in the configuration file on the 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()
```

Observe that the instruction must be a valid Erlang term ended by a dot.

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

The use and effect of the actions are:

- `{Name,undecoded}` Tells that the element will be left undecoded during the exclusive decode. The type of Name may be any ASN.1 type. The value of element Name will be returned as a tuple,as mentioned *above*, in the value structure of the top type.
- `{Name,parts}` The type of Name may be one of SEQUENCE OF or SET OF. The action implies that the different components of Name will be left undecoded. The value of Name will be returned as a tuple, as *above* , where the second element is a list of binaries. That is because the representation of a SEQUENCE OF/ SET OF in Erlang is a list of its internal type. Any of the elements of this list or the entire list can be decoded by the `decode_part` function.
- `{Name,Element_List}`This action is used when one or more of the sub-types of Name will be exclusive decoded.

Name in the actions above may be a component name of a SEQUENCE or a SET or a name of an alternative in a CHOICE.

Example

In the examples below we use the definitions from the following ASN.1 spec:

```
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 we want to exclude component `number` from decode the Type_List in the instruction in the configuration file will be `['Button',[{number,undecoded}]]`. If we call the decode function `decode_Button_exclusive` the Decode_Instruction will be `{decode_Button_exclusive, ['Button',[{number,undecoded}]]}`.

We also have another top type `Window` whose sub component actions in type `Status` and the parts of component `buttonList` shall be left undecoded. For this type we name the function `decode__Window_exclusive`. The whole 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}]]}]}}.
```

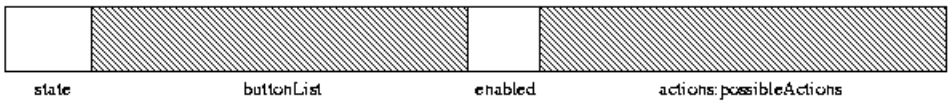

Figure 2.1: Figure symbolizes the bytes of a Window:status message. The components buttonList and actions are excluded from decode. Only state and enabled are decoded when decode__Window_exclusive is called.

Compiling GUI.asn including the configuration file is done like:

```
unix> erlc -bber_bin +optimize +asn1config GUI.asn

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

The module can be used like:

```
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}}]}}
```

1.2.2 Selective Decode

This specialized decode decodes one single subtype of a constructed value. It is the fastest method to extract one sub value. The typical use of this decode is when one want to inspect, for instance 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}`.

How To Make It Work

The following steps are necessary:

- Write instructions in the configuration file. Including the name of a user function, the name of the ASN.1 specification and a notation that tells which part of the type will be decoded.
- Compile with the additional option `asn1config`. The compiler searches for a configuration file with the same name as the ASN.1 spec but with the extension .asn1config. In the same file you can provide configuration specs for exclusive decode as well. 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. You can invoke that function by the `ModuleName:FunctionName` notation.

So, if you have the following spec `{selective_decode,{'ModuleName',[{selected_decode_Window,TypeList}]}}` in the con-fig file, you do the selective decode by `{ok,Result}='ModuleName':selected_decode_Window(EncodedBinary).`

Writing a Selective Decode Instruction

It is possible to describe one or many selective decode functions in a configuration file, you have to 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()]
```

Observe that the instruction must be a valid Erlang term ended by a dot.

The `Module_Name` is the same as the name of the ASN.1 spec, but without the extension. A `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. Notice that you have to choose a name of your function that will not be the same as any of the generated functions. The first element of the `Type_List` is the top type of the encoded message. In the `Element_List` it is followed by each of the component names that leads to selected type. Each of the names in the `Element_List` must be constructed types except the last name, which can be any type.

The List_Selector makes it possible to choose one of the encoded components in a SEQUENCE OF/ SET OF. It is also possible to go further in that component and pick a sub type of that to decode. So in the `Type_List`: `['Window',status,buttonList,[1],number]` the component `buttonList` has to be a SEQUENCE OF or SET OF type. In this example component `number` of the first of the encoded elements in the SEQUENCE OF `buttonList` is selected. This apply on the ASN.1 spec *above*.

Another Example

In this example we use the same ASN.1 spec as *above*. A valid selective decode instruction is:

```
{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 `Decode_Instruction`, `{selected_decode_Window1,['Window',status,buttonList,[1],number]}` is commented in the previous section. The instruction `{selected_decode_Action,['Action',handle,number]}` picks the component `number` in the `handle` component of the type `Action`. If we have the value `ValAction = {'Action',17,{'Button',4711,false}}` the internal value 4711 should be picked by `selected_decode_Action`. In an Erlang terminal it looks like:

```
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]`, which is a little more complicated,

- starts with type *Window*.
- Picks component *status* of `Window` that is of type `Status`.
- Then takes component *actions* of type `Status`.
- Then *possibleActions* of the internal defined CHOICE type.
- Thereafter it goes into the first component of the SEQUENCE OF by *[1]*. That component is of type `Action`.
- The instruction next picks component *handle*.
- And finally component *number* of the type `Button`.

The following figures shows which components are in the TypeList `['Window',status,actions,possibleActions,[1],handle,number]`. And which part of a message that will be decoded by selected_decode_Window2.

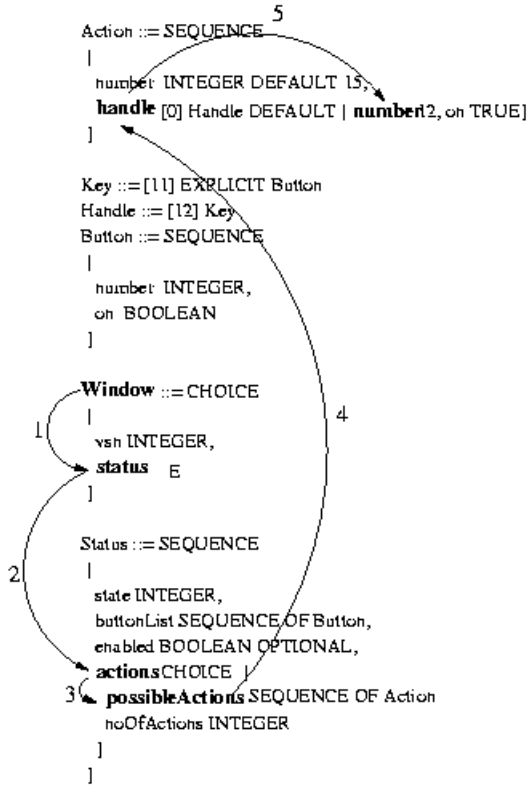

Figure 2.2: The elements specified in the config file for selective decode of a sub-value in a Window message

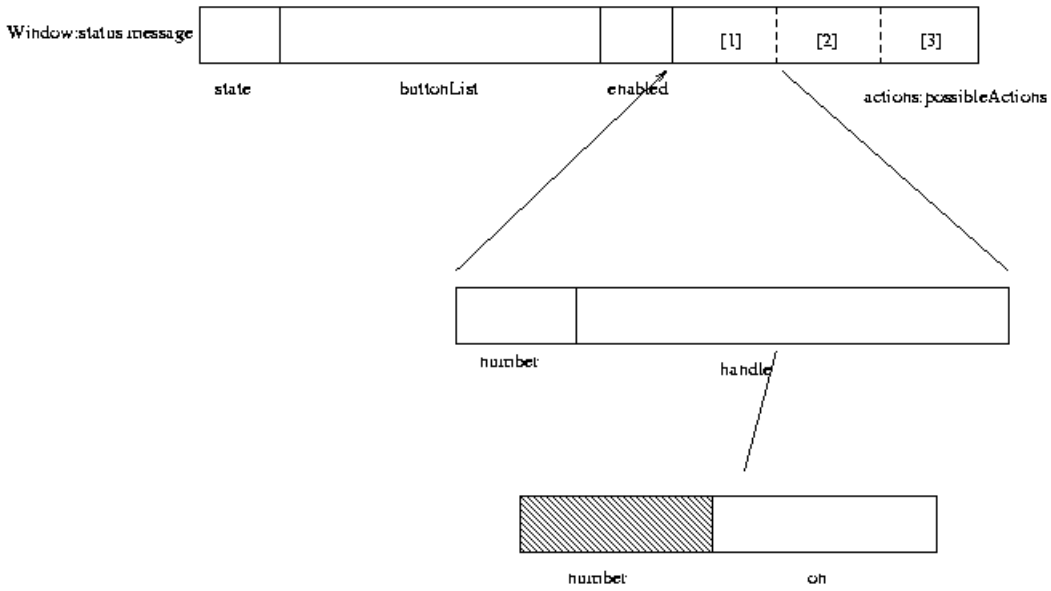

Figure 2.3: Figure symbolizes the bytes of a Window:status message. Only the marked element is decoded when selected_decode_Window2 is called.

With the following example you can examine that both `selected_decode_Window2` and `selected_decode_Window1` decodes the intended sub-value of the 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}
```

Observe that the value feed into the selective decode functions must be a binary.

1.2.3 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) depends 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 specs *GUI* and **MEDIA-GATEWAY-CONTROL** was used in the test.

For the GUI spec the configuration looked like:

```
{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:

```
{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:

```
{'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 specs in the test are compiled with the options `ber_bin, optimize, driver` and `asn1config`. If the `driver` option had been omitted there should have been higher values for `decode` and `decode_part`.

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

Function	*Time*(microseconds)	*Kind of Decode*	*ASN.1 spec*	*% of time vs. complete decode*
decode_MegacoMessage_selective/1	374045	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	449586	selective	GUI	*7.6*
selected_decode_window2/1	890686	selective	GUI	*15.1*
decode_Window_struct_exclusive/1	1251878	exclusive	GUI	*21.3*
decode/2	5889197	complete	GUI	*100*

Table 2.1: Results of complete, exclusive and selective decode

Another interesting question is what 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 may be compared to this simulation, e.g. inspect a sub-value and later on look at the entire value. The following table shows figures from this test. The number of loops and time unit is the same as in the previous test.

Actions	*Function*	*Time*(microseconds)	*ASN.1 spec*	*% 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 2.2: Results of complete, exclusive + decode_part and selective + complete decodes

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

Comments

Generally speaking the gain of selective and exclusive decode in advance of complete decode is greater the bigger value and the less deep in the structure you have to decode. One should also prefer selective decode instead of exclusive decode if you are interested in just one single sub-value.

Another observation is that the exclusive decode followed by decode_part decodes is very attractive if the parts will be sent to different servers for decoding or if one in some cases not is interested in all parts.

The fastest selective decode are when the decoded type is a primitive type and not so deep in the structure of the top type. The `selected_decode_Window2` decodes a big constructed value, which explains why this operation is relatively slow.

It may vary from case to case which combination of selective/complete decode or exclusive/part decode is the fastest.

2 Reference Manual

The *Asn1* application contains modules with compile-time and run-time 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 data-types specified. Alternatively the compiler takes a specification module (se below) specifying all input modules and generates one module with encode/decode functions. There are also some generic functions which can be used in during development of applications which handles ASN.1 data (encoded as BER or PER).

Exports

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

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

Types:

Asn1module = atom() | string()

Options = [Option| OldOption]

Option = ber_bin | per_bin | uper_bin | der | compact_bit_string | noobj | {n2n,EnumTypeName} | {outdir,Dir} | {i,IncludeDir} | optimize | driver | asn1config | undec_rest | {inline,OutputName} | inline | {macro_name_prefix, Prefix} | {record_name_prefix, Prefix} | verbose

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 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 Asn1 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.erl and SetOfFiles.asn1db`.

Sometimes in a system of ASN.1 modules there are different default tag modes, e.g. AUTOMATIC, IMPLICIT or EXPLICIT. The multi file compilation resolves the default tagging as if the modules were compiled separately.

Another unwanted effect that may occur in multi file compilation is name collisions. The compiler solves this problem in two ways: If the definitions are identical then the output module keeps only one definition with the original name.

But if definitions only have same name and differs in the definition, then they will be renamed. The new names will be the definition name and the original module name concatenated.

If any name collision have occurred 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 asn1 compiler and options that are applied to the Erlang compiler. The latter are those that not is recognized as asn1 specific. For *preferred option use* see *Preferred Option Use section in users guide*. Available options are:

`ber | ber_bin | per | per_bin | uper_bin`

The encoding rule to be used. The supported encoding rules are BER (Basic Encoding Rules), PER aligned (Packed Encoding Rules) and PER unaligned. If the encoding rule option is omitted `ber` is the default. The `per_bin` option means the aligned variant. To use the unaligned variant the `uper_bin` option has to be used.

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 `OldOptions` `ber` and `per` 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 binary or an iolist.

`der`

By this option the Distinguished Encoding Rules (DER) is chosen. DER is regarded as a specialized variant of the BER encoding rule, therefore the `der` option only makes sense together with the `ber` or `ber_bin` option. 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* .

`{n2n,EnumTypeName}`

Tells the compiler to generate functions for conversion between names (as atoms) and numbers and vice versa for the EnumTypeName specified. There can be multiple occurrences of this option in order to specify several type names. The type names must be declared as ENUMERATIONS in the ASN.1 spec. If the EnumTypeName does not exist in the ASN.1 spec the compilation will stop with an error code. The generated conversion functions are named `name2num_EnumTypeName/1` and `num2name_EnumTypeName/1`.

`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.

`{i,IncludeDir}`

Adds `IncludeDir` to the search-path for `.asn1db` and asn1 source files. The compiler tries to open a `.asn1db` file when a module imports definitions from another ASN.1 module. If no `.asn1db` file is found the asn1 source file is parsed. Several `{i,IncludeDir}` can be given.

`{outdir,Dir}`

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

`optimize`

This option is only valid together with one of the `per_bin` or `ber_bin` option. It gives time optimized code generated and it uses another runtime module and in the `per_bin` case a linked-in driver. The result in the `per_bin` case from an encode when compiled with this option will be a binary.

`driver`

Option valid together with `ber_bin` and `optimize` options. It enables the use of a linked-in driver that gives considerable faster decode. In `ber_bin` the driver is enabled only by explicit use of the option `driver`.

`asn1config`

When one of the specialized decodes, exclusive or selective decode, is wanted one has to give instructions in a configuration file. The option `asn1config` enables specialized decodes and takes the configuration file, which has the same name as the ASN.1 spec but with extension `.asn1config`, in concern.

The instructions for exclusive decode must follow the *instruction and grammar in the User's Guide*.

You can also find the instructions for selective decode in the *User's Guide*.

`undec_rest`

A buffer that holds a message, being decoded may also have some following bytes. Now it is possible to get those following bytes returned together with the decoded value. If an asn1 spec is compiled with this option a tuple `{ok,Value,Rest}` is returned. `Rest` may be a list or a binary. Earlier versions of the compiler ignored those following bytes.

`{inline,OutputName}`

Compiling with this option gives one output module containing all asn1 run-time functionality. The asn1 specs are provided in a target module Module.set.asn as described *above*. The name of the resulting module containing generated encode/decode functions and in-lined run-time functions will be `OutputName.erl`. The merging/in-lining of code is done by the `igor` module of `syntax_tools`. By default the functions generated from the first asn1 spec in the `.set.asn` are exported, unless a `{export,[atom()]}` or `{export_all,true}` option are provided. The list of atoms are names of chosen asn1 specs from the `.set.asn` file.

`inline`

It is also possible to use the sole argument `inline`. It is as `{inline,OutputName}`, but the output file gets the default name of the source `.set.asn` file.

`{macro_name_prefix, Prefix}`

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

`verbose`

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

Any additional option that is applied will be 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` 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.*

`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} = asn1ct: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

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

Types:

Reason = term()

Starts the asn1 server that loads the drivers.

The server schedules a driver that is not blocked by another caller. The driver is used by the asn1 application if specs are compiled with options `[per_bin, optimize]` or `[ber_bin, optimize, driver]`. The server will be started automatically at encode/decode if it isn't done explicitly. If encode/decode with driver is used in test or industrial code it is a performance gain to start it explicitly to avoid the one time load in run-time.

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

Types:

Reason = term()

Stops the asn1 server and unloads the drivers.

`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 or binary `Bytes`. If the module is compiled with `ber_bin` or `per_bin` option `Bytes` must be a binary. Returns `{ok,Value}` if successful.

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

Types:

Module = Type = atom()

Value = term()

BinOrList = Bytes | binary()

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. If `Module` was compiled with the options `per_bin` and `optimize` the result is a binary. 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.

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

Types:

Module = atom()

Info = list()

Reason = term()

`info/1` returns the version of the asn1 compiler that was used to compile the module. It also returns the compiler options that was used.

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

Types:

Reason = term()

This function loads the linked-in driver before the first call to encode. If this function is not called the driver will be loaded automatically at the first call to encode. If one doesn't want the performance cost of a driver load when the application is running, this function makes it possible to load the driver in an initialization.

The driver is only used when encoding/decoding ASN.1 files that were compiled with the options `per_bin` and `optimize`.

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

Types:

Reason = term()

This function unloads the linked-in driver. When the driver has been loaded it remains in the environment until it is unloaded. Normally the driver should remain loaded, it is crucial for the performance of ASN.1 encoding.

The driver is only used when ASN.1 modules have been compiled with the flags `per_bin` and `optimize`.

`utf8_binary_to_list(UTF8Binary) -> {ok,UnicodeList} | {error,Reason}`

Types:

UTF8Binary = binary()

UnicodeList = [integer()]

Reason = term()

`utf8_binary_to_list/1` Transforms a UTF8 encoded binary to a list of integers, where each integer represents one character as its unicode value. The function fails if the binary is not a properly encoded UTF8 string.

`utf8_list_to_binary(UnicodeList) -> {ok,UTF8Binary} | {error,Reason}`

Types:

UnicodeList = [integer()]

UTF8Binary = binary()

Reason = term()

`utf8_list_to_binary/1` Transforms a list of integers, where each integer represents one character as its unicode value, to a UTF8 encoded binary.

`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.*