

# ASN.1 tagging principles

Unambiguous descriptions  
Decoding performances

Version 0.0.4

<http://www.powerasn.com>

## ASN.1 tag definition

A tag is defined by a {class, value} pair where:

- 'class' takes one of the following:
  - APPLICATION : global (top-level) types
  - CONTEXT : used by default
  - PRIVATE : rarely used
  - UNIVERSAL : reserved to ASN.1 types
- 'value' is a positive number

## Ambiguity issue

- Basic ASN.1 notation may be ambiguous
- Example:

```
Ambiguous ::= SEQUENCE {
    val1    INTEGER    OPTIONAL,
    val2    INTEGER    OPTIONAL
}
```



Leads to a decoding ambiguity...

## UNIVERSAL tags for ASN.1 types

UNIVERSAL tagging is reserved to ASN.1 types

Examples:

- |               |   |                |
|---------------|---|----------------|
| • BOOLEAN     | = | [UNIVERSAL 1]  |
| • INTEGER     | = | [UNIVERSAL 2]  |
| • SEQUENCE    | } | [UNIVERSAL 16] |
| • SEQUENCE OF |   |                |
| • SET         | } | [UNIVERSAL 17] |
| • SET OF      |   |                |
| • ...         |   |                |

Tagging may:

- Override an existing type (or tag!)



IMPLICIT tagging

- Add its value prior to the overridden type or tag



EXPLICIT tagging (default mode)

This recursive definition shows that tags can be applied on top of already tagged elements

MyModule DEFINITIONS

IMPLICIT TAGS

-- Default tagging mode

-- Other values: EXPLICIT, AUTOMATIC

BEGIN

...

Version ::= [0] INTEGER

-- Equivalent to [0] IMPLICIT INTEGER

...

END

- EXPLICIT tagging:
  - Default tagging mode
  - EXPLICIT tag is added in front of the existing tag
- IMPLICIT tagging:
  - IMPLICIT tag replaces an existing tag
- AUTOMATIC tagging:
  - Convenient method to avoid manually tagging elements
  - Defined by ASN.1 97 notation

Algoid ::= [0] OBJECT IDENTIFIER

AppInt ::= [APPLICATION 1] INTEGER

MySet ::= [CONTEXT 10] SET OF INTEGER

PrivInt ::= [PRIVATE 0] IMPLICIT INTEGER

Note that [X] and [CONTEXT X] are equivalent

- Primary use of tags
- Avoid ambiguous descriptions
- Example:

```
Unambiguous ::= SEQUENCE {
  val1    [0]  INTEGER      OPTIONAL,
  val2    INTEGER      OPTIONAL
}
```

Tagging distinguishes val1 and val2 encodings

```
Alt ::= CHOICE {
  first    VisibleString
  second   INTEGER
  third    VisibleString
}
```

→ first and third elements are ambiguous...

... One of them must be tagged:

```
third    [0]  VisibleString    for example
```

```
IntSet ::= SET OF [0] IMPLICIT INTEGER
```

Equivalent to:

```
IntSet ::= SET OF Int
```

```
Int    ::= [0] IMPLICIT INTEGER
```


- Tags values must be sequential from 0
- Tagging class must not be **UNIVERSAL**

```
InvalidSeq ::= SEQUENCE {
  version  INTEGER,
  name     [UNIVERSAL 0] PrintableString
}
```

→ is not a valid declaration!

- OpenType and CHOICE types are always explicitly tagged (even when not mentioned in an IMPLICIT TAGS module header)
- CHOICE inner elements tags must be distinct

Generic ::= [0] IMPLICIT OpenType

 is not a valid declaration (EXPLICIT tagging only)

```
SimpleAmbiguousSequence ::= SEQUENCE {
    sometimes    INTEGER    OPTIONAL,
    always       INTEGER
}
```

ASN.1 offers multiple (recursive) tagging:


TaggedInt ::= [1] EXPLICIT [0] IMPLICIT INTEGER

Equivalent to:

TaggedInt ::= [1] EXPLICIT Int

Int ::= [0] IMPLICIT INTEGER

What is the problem?

 When sometimes is not present (and always is initialized), the encoding is successfully generated (a single INTEGER value)...

... but many decoders will decode this value into sometimes and raise an error as always is not present!

 Use tags!

Non-ambiguous declaration (example):

```
SimpleAmbiguousSequence ::= SEQUENCE {
    sometimes      INTEGER      OPTIONAL,
    always         [0]  INTEGER
}
```

What is the problem?

➡ When first is not present and second holds a single INTEGER value, the encoding is successfully generated...

... but the decoder cannot determine whether first or second is encoded!

➡ Use tags again!

```
HiddenSequence ::= SEQUENCE {
    first      Seq      OPTIONAL,
    second     SeqOfInt
}

Seq ::= SEQUENCE {
    val      INTEGER
}
```

SeqOfInt ::= SEQUENCE OF INTEGER

Non-ambiguous declaration (example):

```
HiddenSequence ::= SEQUENCE {
    first      Seq      OPTIONAL,
    second     [0]  SeqOf
}
```

```
TrickySequence ::= SEQUENCE {
    first      Seq          OPTIONAL,
    second    SeqOfBool
}
```

```
Seq ::= SEQUENCE {
    val      INTEGER
}
```

```
SeqOfBool ::= SEQUENCE OF BOOLEAN
```

Well-formed declaration:

```
TrickySequence ::= SEQUENCE {
    first      Seq          OPTIONAL,
    second    [0] SeqOfBool
}
```

What is the problem?

→ first and second hold different inner elements types (INTEGER and BOOLEAN)...

... but when first is absent, decoders have to go inside the SEQUENCE OF and fail to decode the first INTEGER value to know that the encoding does not enclose first...

... and many decoders may even fail to decode!

Tags usage is often necessary to:

- Avoid ambiguous encodings
- Optimize decodings

Tagging is strongly linked with encoding rules:

- BER, CER and DER need non-ambiguous tagging
- XER only refers to elements names and ASN.1 types and PER does not encode tags

→ Refer to encoding rules presentation...