ASN.1: encoding

Gusztáv ADAMIS
BME TMIT
adamis@tmit.bme.hu
György RÉTHY, János Zoltán SZABÓ
Test Competence Center, Ericsson Hungary
Sunday, February 26, 2017

Contents

PER

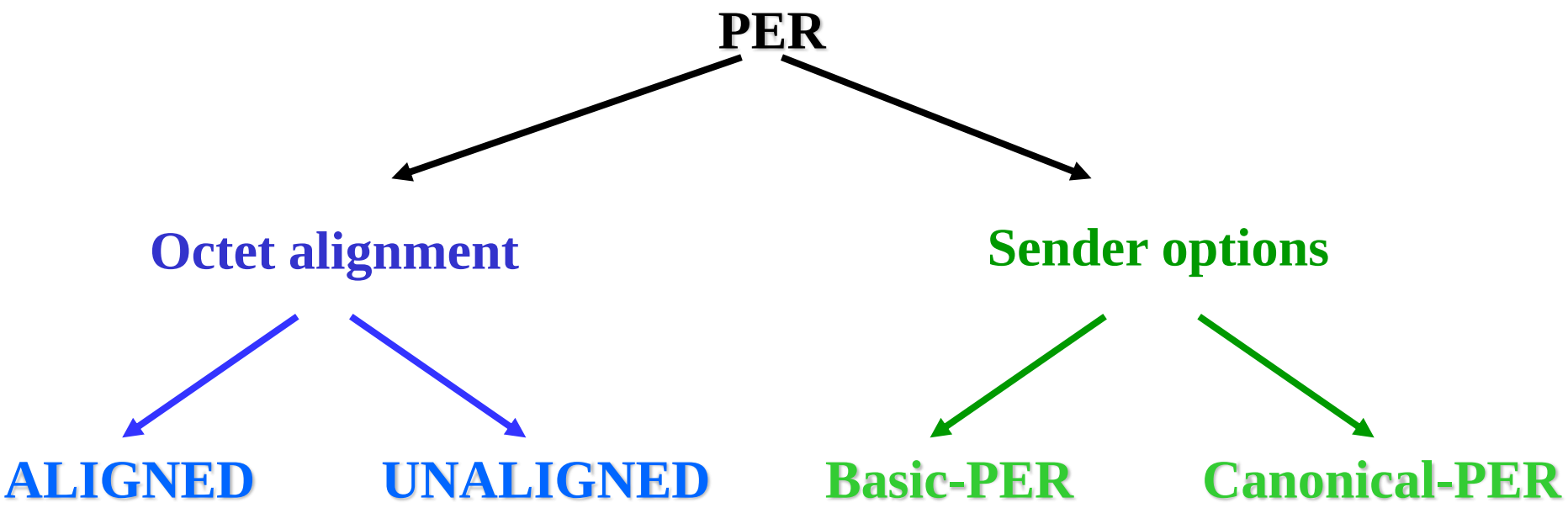
- Basic principles of PER
- PER-visible type constraints (sub-typing)
- Encoding of different types

PER: Basic principles

- **BE AS COMPACT AS REASONABLY POSSIBLE**
  - **bit-oriented where possible**
  - **neither the type nor tag is coded**
  - **use of type constraints where possible to decrease size of message**
  - **always the shortest possible format shall be used**

PER:

Types of PER



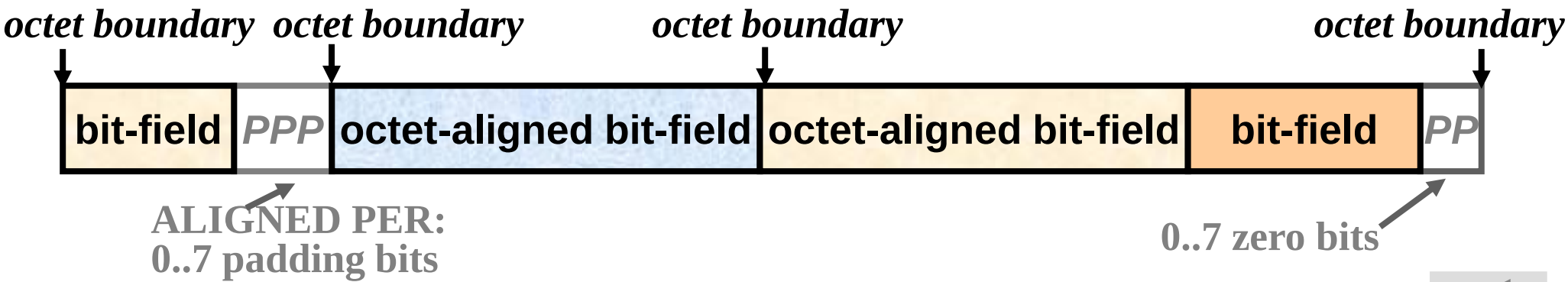
PER:

WHAT THE ENCODING OF A VALUE DEPENDS ON?

- The type
- PER-visible sub-typing
- PER-visible extension marker(s)
- OPTIONAL and/or DEFAULT element(s) in the type definition
- Tags of the components of complex types (SET, CHOICE)
- Whether a component is an open type
- If the value of an extensible type is within the root or not
- The value itself

PER: Production of the complete encoding

- All inner values are encoded and a field-list created
- Concatenate all fields of the field-list
  - without any padding bits for UNALIGNED PER
  - adding 0..7 padding bits before any octet-aligned bit fields for ALIGNED PER
- Append 0..7 zero bits at the end of the whole production to produce a multiple of 8 bits
- If the result is an empty bit string, replace it with one “0” octet

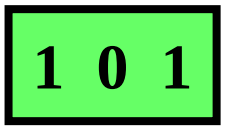


PER: Encoding of open types

- In general (the same way the complete PDU processed)
  - The actual type(s) occupying the field is/are encoded into bit fields
  - Bit fields are concatenated, with padding bits, where needed
  - The production is padded to multiple number of 8 bits
  - The whole composite octet string is wrapped by a general length determinant, which ALWAYS counts in octets

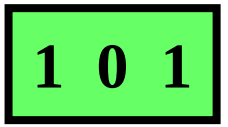
PER: *Uses subtype constraint at coding*

element1 INTEGER(0..7) ::= 5



Value	Code point
0	000
1	001
...	

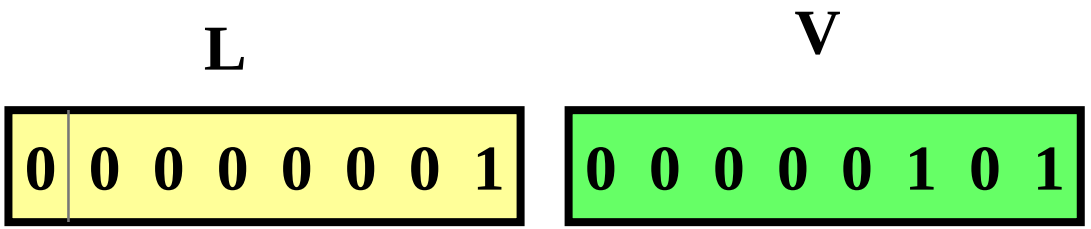
element2 INTEGER(15..22) ::= 20



Value	Code point
15	000
16	001
...	

element3 INTEGER (MIN..7) ::= 5

element4 INTEGER ::= 5



PER: PER visibility of subtypes - 1

- **Single value, value list constraint: is visible for INTEGER only**
  - **INTEGER (1|2)** **=> coded as constrained type**
  - **IA5String (“abc” | “abcd”), BIT STRING (‘00’|’11’)**
    => coded as unconstrained type

SUMMARY

PER: PER visibility of subtypes - 1

- **Value range constraint: NOT visible for REAL and *not* known-multiplier character string types :**

  INTEGER (0..255) **=> coded as constrained type**

  VideotexString (FROM (“a” .. ”z”)) **=> coded as unconstrained type**

- **Known-multiplier character string types:**
  - **IA5String**
  - **PrintableString**
  - **VisibleString**
  - **NumericString**
  - **UniversalString**
  - **BMPString**

PER: PER visibility of subtypes - 2

- **Size constraint** is **NOT** visible for not known-multiplier character string types

  SEQUENCE (SIZE (0..63)) OF MyType => coded as constrained type

  IA5String (SIZE (0..63)) => coded as constrained type

  VideotexString (SIZE (0..63) => coded as unconstrained type

- **Permitted Alphabet** is visible **ONLY** for non-extensible known-multiplier character string types

  IA5String (FROM (“abc”)) => coded as constrained type

  IA5String (FROM (“abc”), ...) => coded as unconstrained type

  VideotexString (FROM (“abc”)) => coded as unconstrained type

- **Type constraint** is **NEVER** PER-visible

SUMMARY

PER: General length determinant (if length is NOT constrained!!)

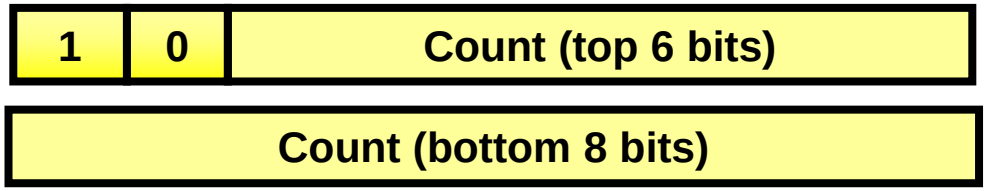
0 – 127:

single octet length encoding

0...127

128 – 16383:

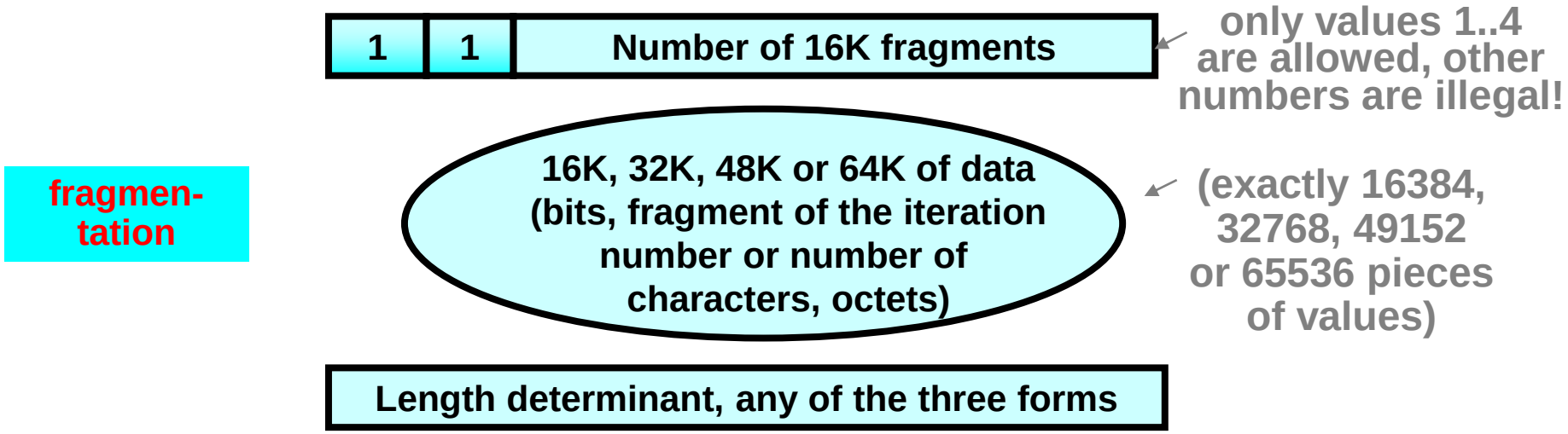
two octets length encoding



128...16383
(16K-1)

PER: General length determinant (if length is NOT constrained!!)

- **From 16384:**



- **If still not enough:**

PER: General length determinant: where to use

- Always used for:
  - semi-constrained types
  - unconstrained types
  - in extensible types for values out of the root range
- Counts
  - BITs for BIT STRING
  - CHARACTERs for known-multiplier character string types
  - ITERATIONs for SEQUENCE OF and SET OF
  - OCTETs in all other cases
- In aligned PER: always octet aligned!

PER Restricted Length

- Length encoded in minimal possible length as unsigned integer
- IA5String(SIZE(6400)) – no length (1 combination!!)
- If length can have 2..256 different values
  - IA5String(SIZE(3..6)) – length 2 bit (4 combinations)
  - IA5String(SIZE(40000..40254) – length 8 bit (255 combinations)
- If length 257..64k – always 2 octets
  - IA5String(0..32000) – length 2 octets

PER: NULL, BOOLEAN

- NULL: just nothing
  (to be formal: encoded by a zero length field)

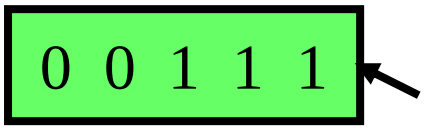
- BOOLEAN: just one bit
  (as normally would be expected for a single digital number)

PER: INTEGER -1

- *Constrained non-extensible* subtype, value range ≤ 255 (!)
  - Finite upper and lower bounds and the constraint is PER-visible
  - Number of bits are defined based on the *RANGE* (upper bound - lower bound + 1)
  - The *OFFSET* from the lower bound is encoded!

```
sample18  INTEGER (OutRange EXCEPT InRange)     ::=  15
          OutRange ::=  INTEGER  ( 8 .. 39 )
          InRange  ::= INTEGER  ( 16 .. 21)
                    (RANGE=32)

sample19 INTEGER ( 0 | 7 | 31)                  ::= 7
                    (RANGE=32)
```



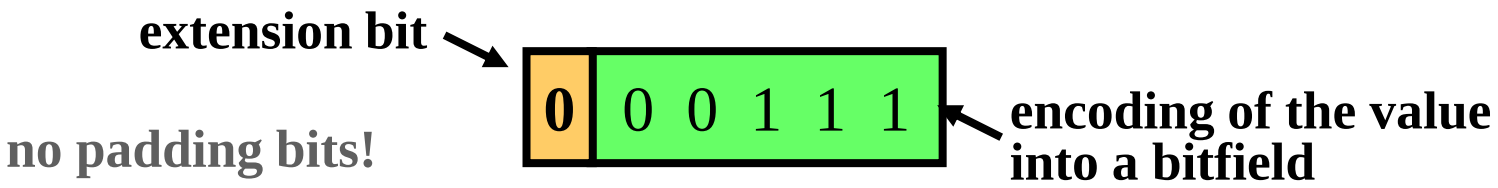
encoding of the value into a bitfield (for both examples)

PER: INTEGER -1/b

- value range ≤ 255 (!)

- Constrained *extensible* subtype or *extended* constrained subtype and the actual value is *within the root*

sample20 INTEGER (0 .. 31, …, 63) ::= 7

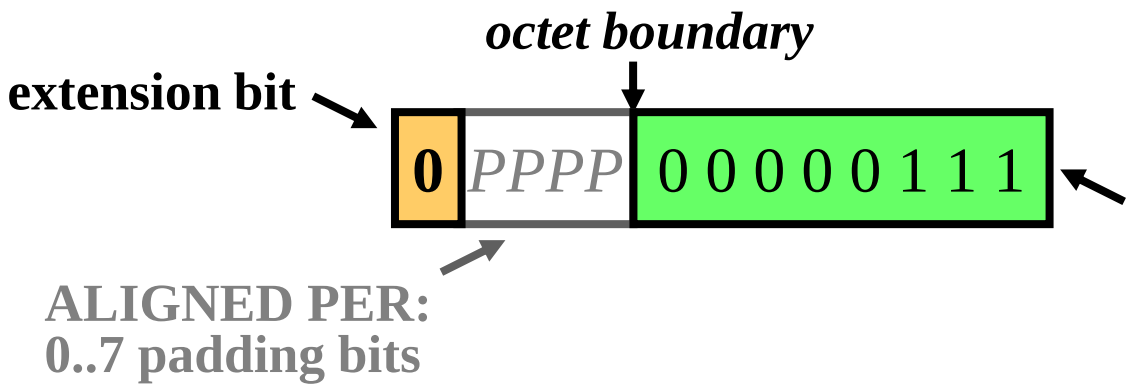


PER: INTEGER -2

- **Constrained non-extensible subtype (no extension bit), constrained extensible subtype or constrained extended subtype and the actual value is within the root**

value range (v.r.) ≥ 256 (!)

sample21 INTEGER (0 .. 255, …, 1023) ::= 7



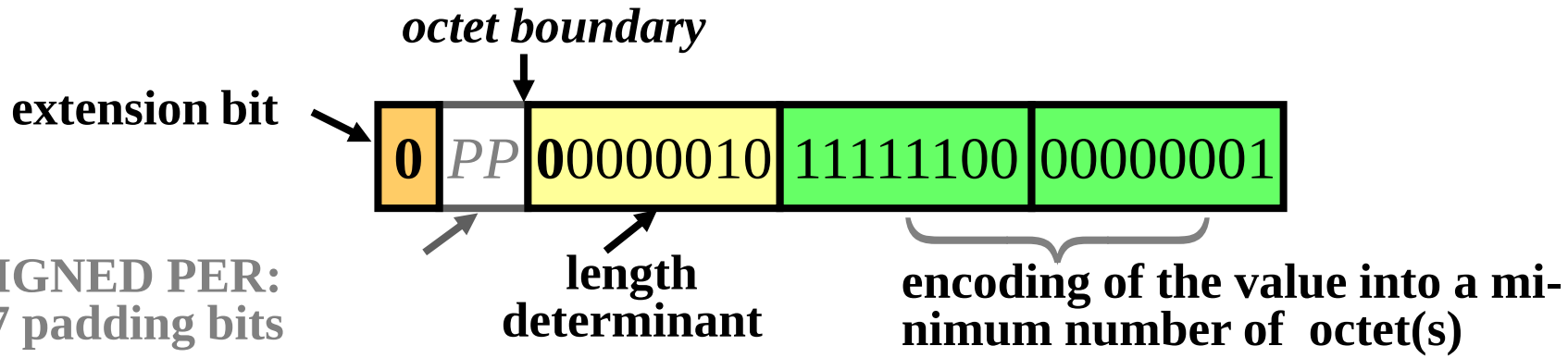
encoding of the value into a minimum number of *octet(s)*

v.r. = 256 -> 1 octet
256 < v.r. ≤ 64K -> 2 octets
etc.

PER: INTEGER -3

- Unconstrained non-extensible type
  - no finite LOWER bound
  - length is in octets and absolute value is encoded -> min. possible length is mandatory
  - value is encoded as signed number -> 2's complement
- Unconstrained extensible subtype or extended constrained subtype and the actual value is within the root

sample23 INTEGER (MIN .. 2047, …, 4095) ::= -1023

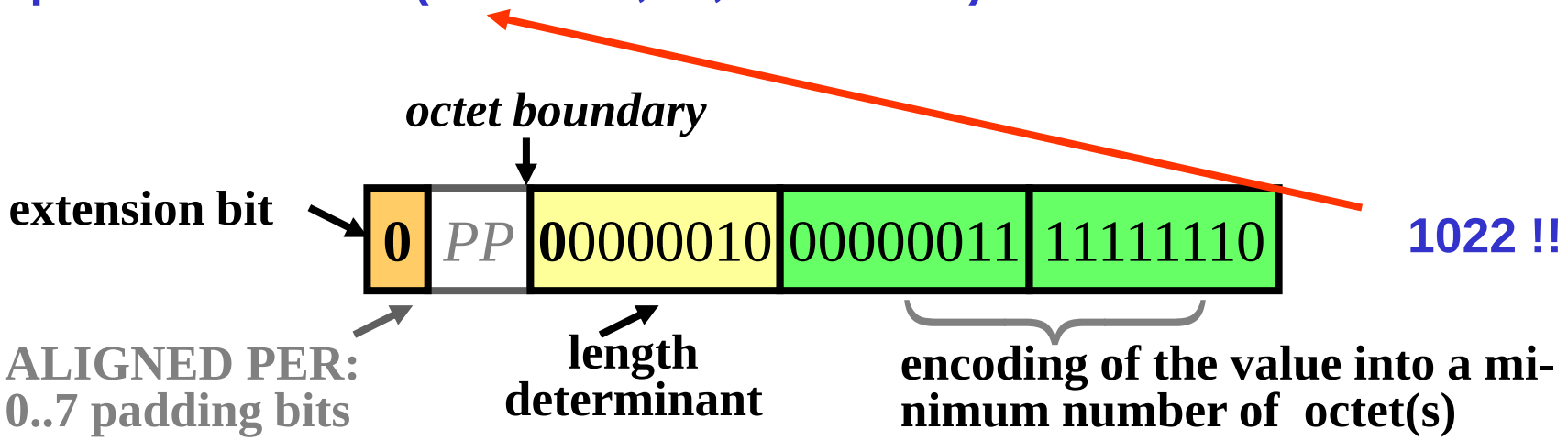


PER: INTEGER -4

- Semi-constrained non-extensible type
  - finite LOWER bound, but no definite upper bound
  - coding format like the unconstrained type (padding-length-value) BUT
  - in the value the *OFFSET* from the lower bound is encoded

- Extended constrained subtype and the actual value is within the root

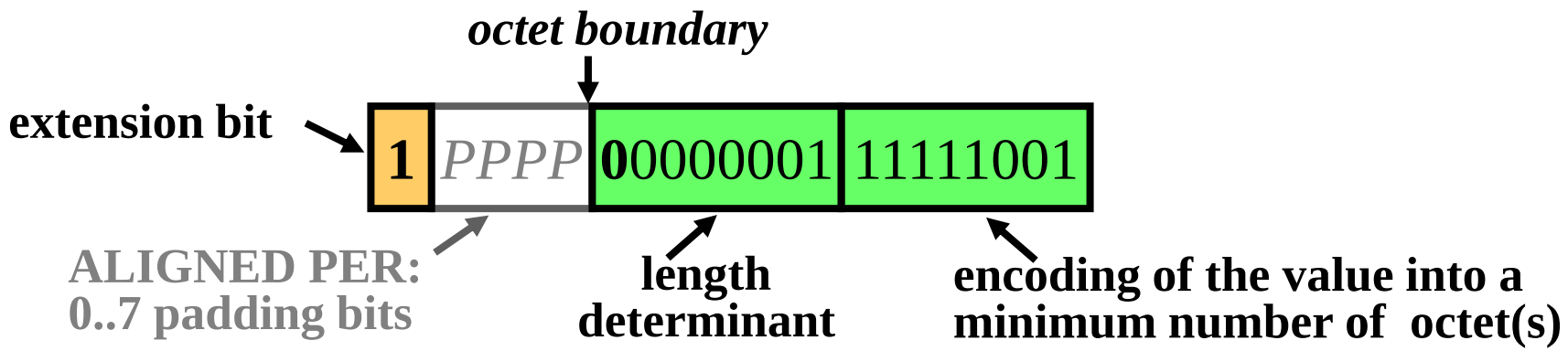
sample25 INTEGER (1 .. MAX, …, -4095 .. -1) ::= 1023



PER: INTEGER -5

- ALL extended subtypes when the actual value is within the extended range
  => extension bit = 1, otherwise the encoding is identical to the unconstrained type
  – always the signed absolute value is encoded in 2's complement!

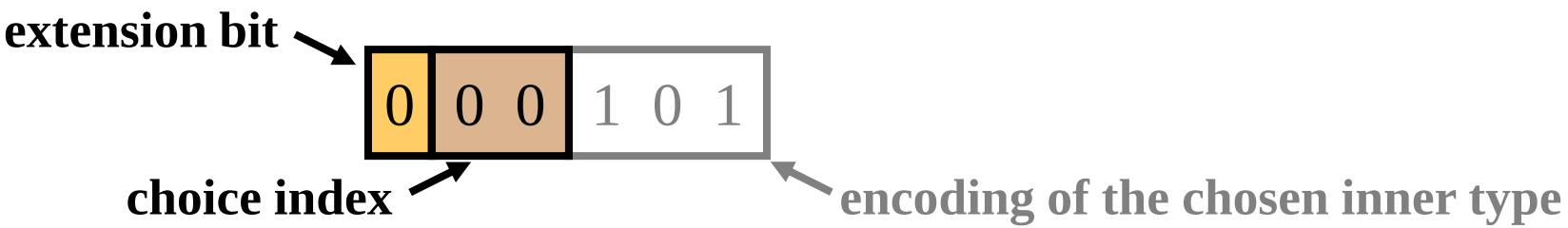
sample27 INTEGER (0 .. 255, …, -1023..-1) ::= -7



PER: CHOICE -1

- Non-extensible type
  - First step: inner choices are arranged in ascending tag order (remember: INTEGER -> (0,2), OCTET STRING ->(0,4), context-spec. tag [0]-> (3,0)
  - CHOICE s are virtually re-numbered from 0 up
  - Then the choice index is encoded just as a constrained integer!
- Extensible type without extended values or the chosen option is within the root

```
sample28 CHOICE {    string          OCTET STRING,
                     value1  [0]     INTEGER (0..15) ,
                     value2          INTEGER (0..7)
                     ... ,
                     flag            BOOLEAN, ~~~   }   ::= value2 : 5
```

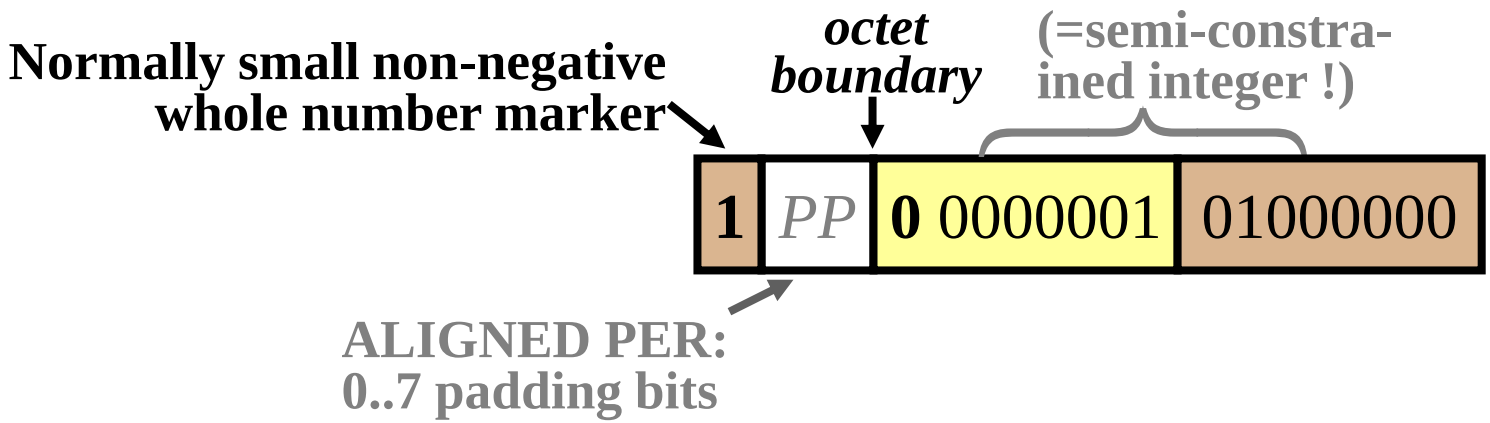


PER: Normally small non-negative whole number

This procedure is used when encoding a non-negative whole number that is expected to be small, but whose size is potentially unlimited due to the presence of an extension marker. An example is a choice index.

0 <= n <= 63:

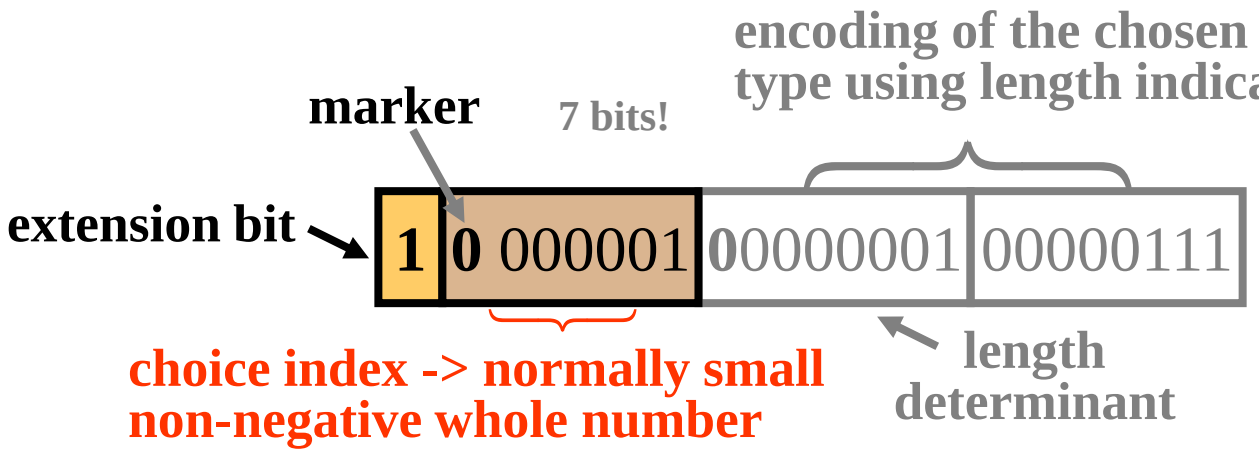
n >= 64: starts by 1 (represented in 1 bit),
encoded as a semi-constrained (0..maxlength)
non-negative INTEGER (length + value)



PER: CHOICE -2

- Extensible type, the number of added choices is ≤ 63 and the chosen option is out of the root
  - for extended values field numbering is re-started from 0
  - choice index = “normally small non-negative whole number”

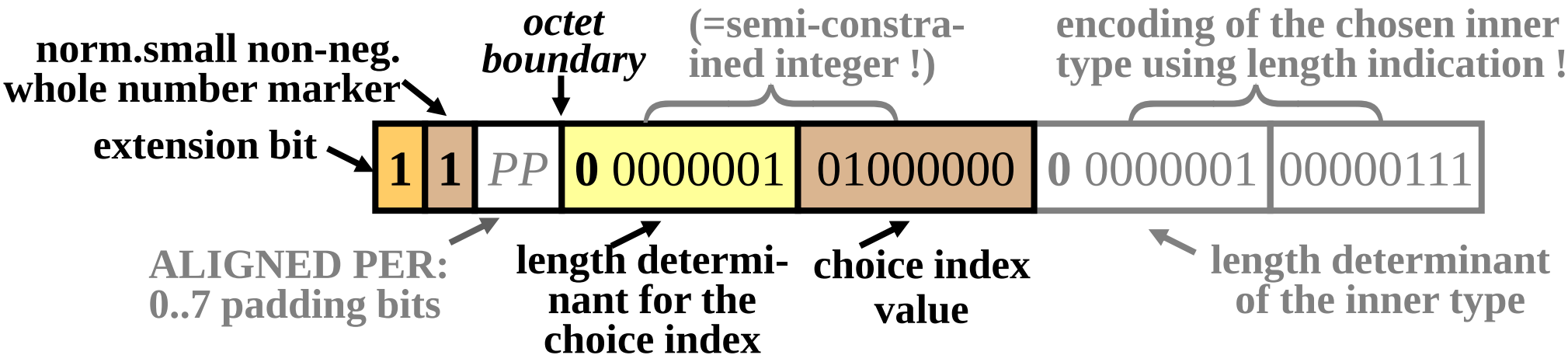
```
sample29 CHOICE {    string          OCTET STRING,
                     value1          INTEGER (0..7) ,
                     value2 [0]      INTEGER (0..15),
                     ...,
                     flag            BOOLEAN,
                     value3 [1]      INTEGER (0..15) }    ::=  value3 : 7
```



PER: CHOICE -3

- Extensible type, the number of added choices is > 63 and the chosen option is out of the root

```
sample30 CHOICE {     string          OCTET STRING,
                      value1          INTEGER (0..7) ,
                      value2 [0]      INTEGER (0..15),
                      ...,
                      flag            BOOLEAN,
                      ~~~
<64th added choice>   value3 [1]      INTEGER (0..15) }     ::=  value3 : 7
```

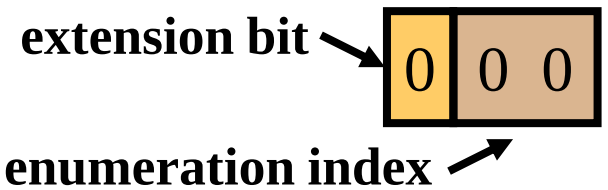


PER: ENUMERATED

- Always constrained from the lower and upper bounds!
- Enumeration index =>Encoded exactly as the choice index
  - Enumerations are re-arranged according to assigned values in increasing order and virtually re-numbered from 0 up
  - Then just encoded as a constrained integer
  - In the sample below the order and encoded values will be: second ->0, first -> 1, third ->2

- for extensible types: extension bit is added; if value is in the root: e.bit = 0, coding = as with the non-extensible type; if value is out of the root: e.bit = 1, coding = normally small non-negative whole numbers

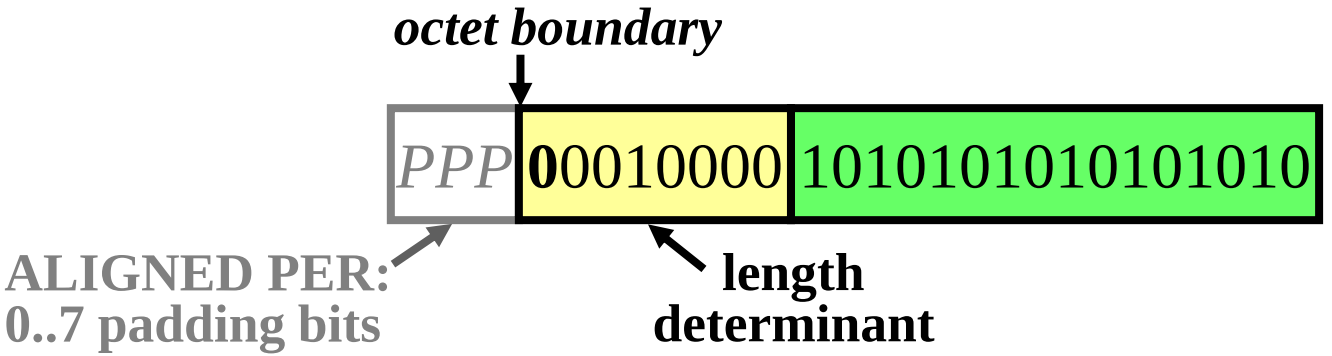
sample31 ENUMERATED { first (2), second (-5), third (9) , … } ::= second



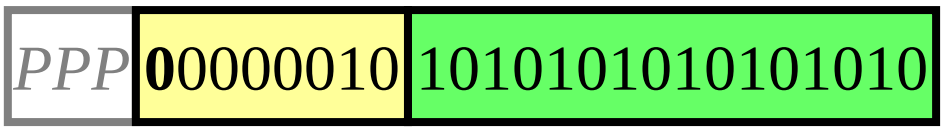
PER: BIT STRING & OCTET STRING - 1

- **Unconstrained type**
  - general length encoding is used (which is always octet aligned)
  - length determinant indicates:
    no. of bits for BIT STRING &
    no. of octets for OCTET STRING

sample37 BIT STRING ::= 'AAAA'H



sample38 OCTET STRING ::= 'AAAA'H

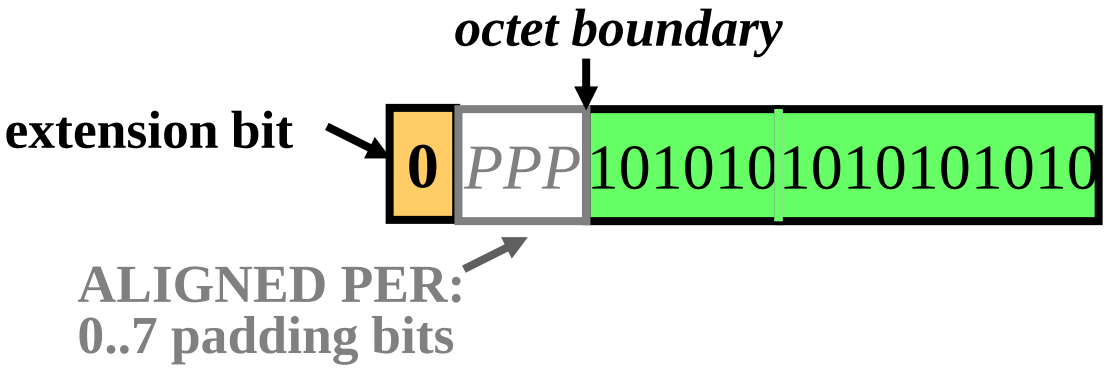


PER: BIT STRING & OCTET STRING - 2

- If length is constrained to a single value, non-extensible length constraint
  - No length is encoded if length ≤ 64 K;
    if length > 64 K, constraint is not PER-visible (->general length encoding)
  - ALIGNED PER: single size strings < 16 bits are unaligned,
    ≥ 16 bits are octet aligned
- Length is constrained to a single value, extensible length constraint

sample32 BIT STRING (SIZE(16), ...) ::= '1010101010101010'B

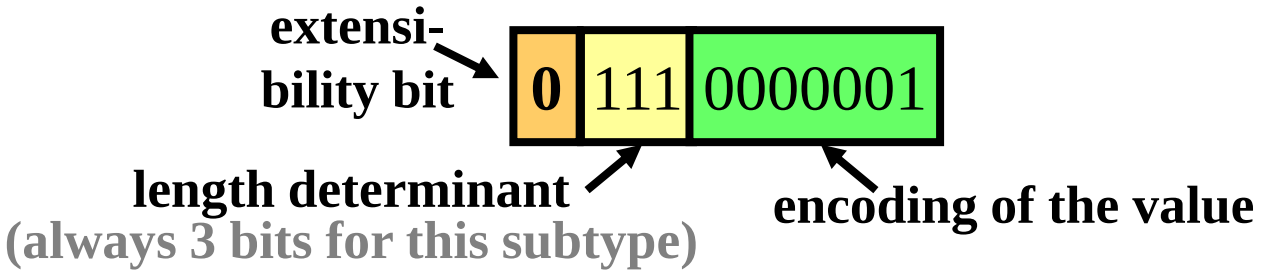
sample33 OCTET STRING (SIZE(2), ...) ::= 'AAAA'H



PER: BIT STRING & OCTET STRING - 3

- Length is constrained to a range & non-extensible
  - Upper bound ≤ 64 K: length is encoded as a ***constrained integer***; (unaligned length field up to and including 255, octet aligned <for ALIGNED PER> octet field above 255, no "length of length") -> bits for BIT SRING, octets for OCTET STRING
  - Upper bound > 64K -> constraint is not PER visible (length is encoded using the appropriate general length form)
- Length is extensible or extended & within the root
  - extension bit = 0 is added, coding is as with the non-extensible type

sample34 BIT STRING (SIZE(0..7), ...) ::= 000001'B

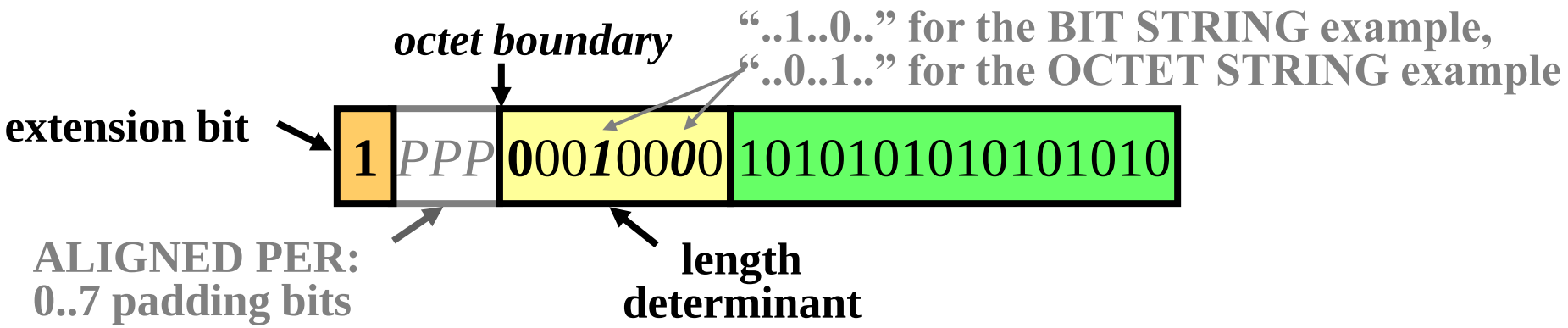


PER: BIT STRING & OCTET STRING - 4

- Length is constrained, actual length is out of the root
  - extension bit = 1
  - general length encoding is used (which is always octet aligned)

sample35 BIT STRING (SIZE(0 ..6), …, SIZE (7..31)) ::= 'AAAA'H

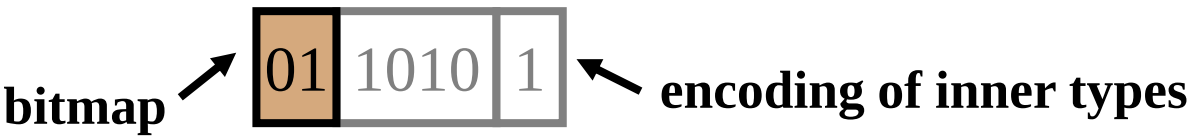
sample36 OCTET STRING (SIZE(0 ..1), …, SIZE (2..4)) ::= 'AAAA'H



PER: SEQUENCE & SET - 1

- SET
  - Reordered to a canonical order using distinct tags
  - Encoded identically as SEQUENCE
- SEQUENCE, non-extensible
  - If no OPTIONAL/DEFAULT fields: just the list of the fields
  - If OPTIONAL/DEFAULT fields: starts by an „optional bitmap”: one bit for each OPTIONAL or DEFAULT element
  - Optional bitmap ≤ 64 K : always unaligned, without length indication
  - Optional bitmap > 64 K : fragmented using general length indication

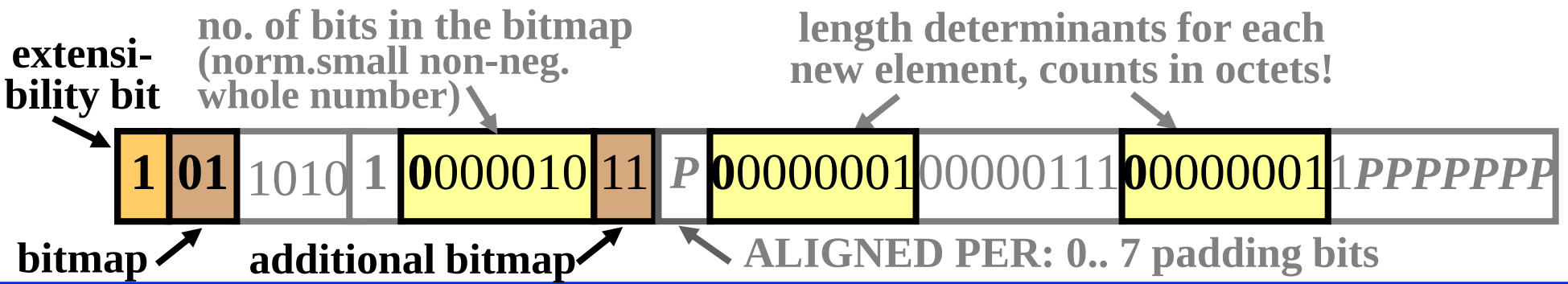
```
sample40  SEQUENCE{
          first      INTEGER (0..15)  OPTIONAL,
          second     INTEGER (0..15),
          third      BOOLEAN          OPTIONAL}  ::= { second 10, third TRUE }
```



PER: SEQUENCE & SET - 2

- SEQUENCE, extensible type
  - No additional elements within the actual sequence: extension bit = 0
  - Extension bit = 1 if there is/are additional element(s)
  - Additional bitmap is added at the insertion point: one bit for EACH additional element, number of bits counted by a normally small non-negative whole number; "1" for each element present
  - EACH additional element uses a general length field -> counts in octets!

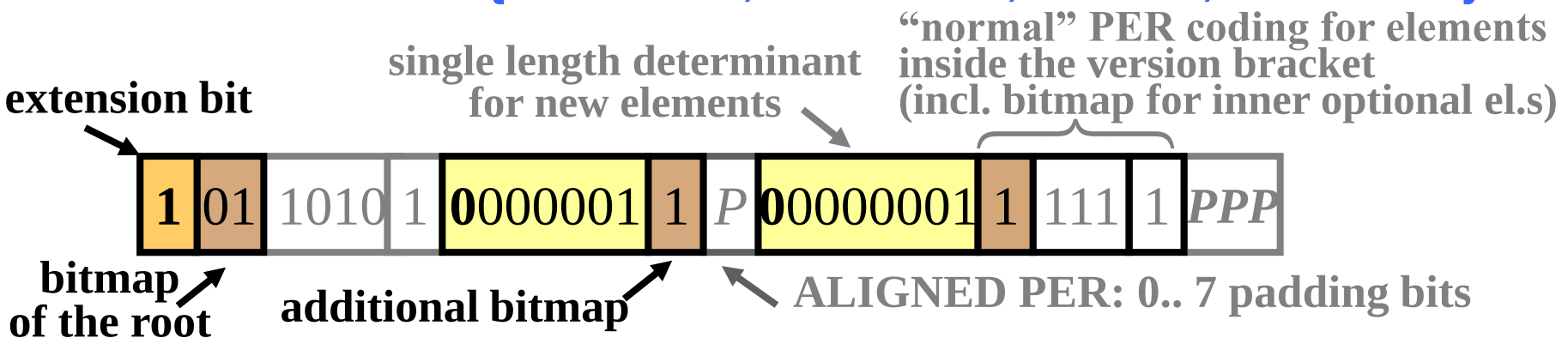
```
sample41      SEQUENCE{
   first      INTEGER (0..15)   OPTIONAL,
   second INTEGER (0..15),
   third    BOOLEAN             OPTIONAL,      ...,
   fourth  INTEGER (0..7),
   fifth     BOOLEAN       OPTIONAL} ::= { second 10, third TRUE, fourth 7, fifth TRUE}
```



PER: SEQUENCE & SET - 3

- Version brackets
  - Allows to use a single length wrapper for all additional elements
  - Allows optimized coding for additional elements
  - Elements within the version bracket are encoded as a SEQUENCE, e.g. adding another extension bit and optional bitmap, if appropriate

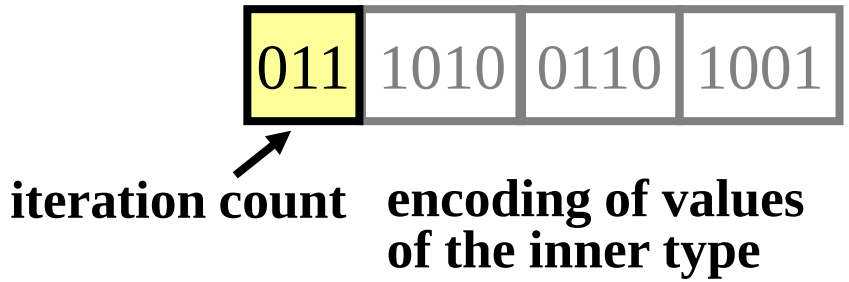
```
sample42  SEQUENCE{
    first          INTEGER (0..15)  OPTIONAL,
    second         INTEGER (0..15),
    third          BOOLEAN          OPTIONAL, …,
    [[ fourth      INTEGER (0..7),
       fifth       BOOLEAN          OPTIONAL]] }
                        ::= { second 10, third TRUE, fourth 7, fifth TRUE }
```



PER: SEQUENCE OF & SET OF

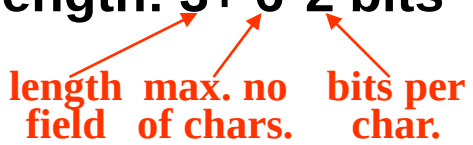
- Only an iteration count is added
  - Encoding of the iteration count is the same as the length field of strings in all the cases
    Shows the number of iterations
  - Extension marker can be added both to the outer or to the inner type

sample43 SEQUENCE (SIZE (0..7)) OF INTEGER (0..15) ::= {10, 6, 9 }



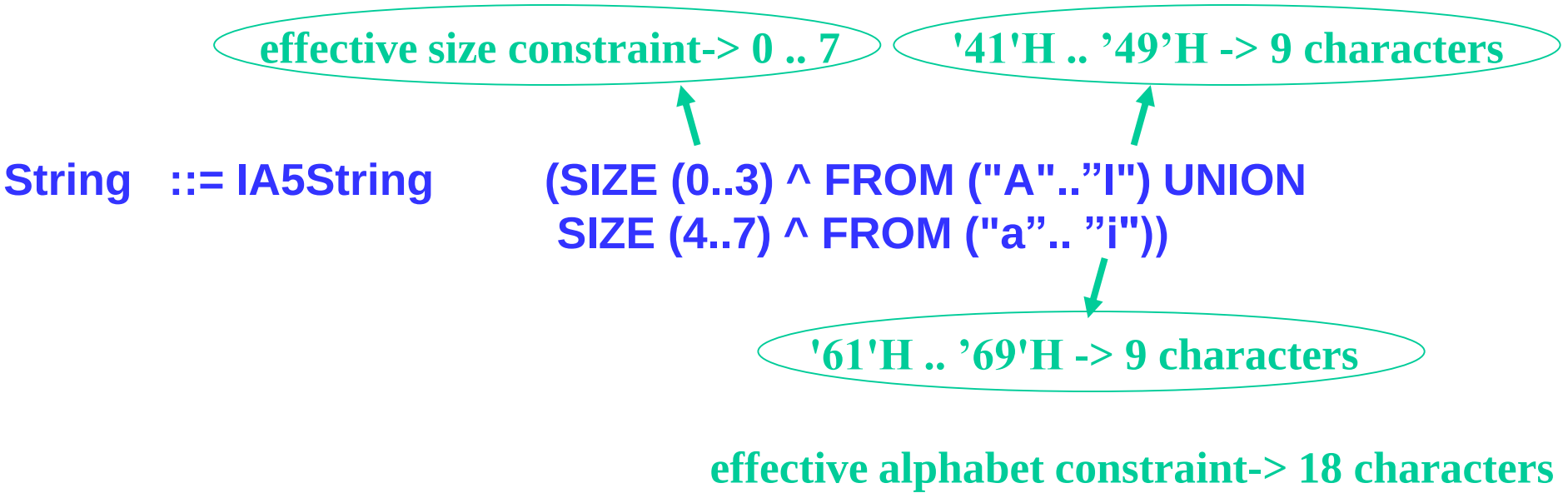
PER: Character string types - 1

- **Known-multiplier character string types**
  - **IA5String, PrintableString, VisibleString (ISO646String), NumericString, UniversalString and BMPString**
  - **Number of bits/character in the set is FIXED and a priori known**
  - **Length determinant counts number of *characters*!**
  - **Effective size and effective character constraints shall be identified separately**
  - ***Fixed length constraint*:**
    IF length of the string < 2octets -> unaligned;
    IF length ≥ 2 octets, the string is octet aligned
  - ***Range constraint*:**
    IF the total length (length bits + all characters) NEVER exceeds 2 octets -> unaligned;
    String ::= IA5String (SIZE(0..6) FROM("A".."D")) **max length: 3+ 6*2 bits**
    IF it can exceed 2 octets -> ALWAYS aligned



PER: Character string types - 2

- Known-multiplier character string types, defining effective size and effective alphabet constraints
  - effective size constraint : max length in subtype - min length in subtype
  - effective alphabet constraint: set of all characters in subtype
  - interrelation of length and characters of individual strings are indifferent



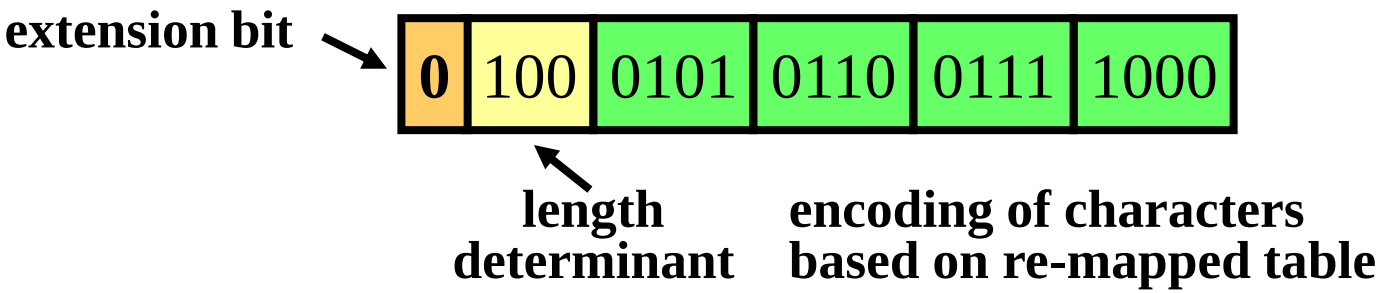
PER: Character string types - 4

- Known-multiplier character string types, non-extensible
  - calculation of effective size and effective alphabet constraints (0..7, 10)
  - calculation of no. of length bits (3) and bits/character (4)
  - if no. of bits/char calculated < then no. of bits/char of the unconstrained type -> re-map characters: new value is zero up based on the canonical order in the original character table -> “A”: 0, “B”: 1 … “a”: 5, “b”: 6 ...
- Extensible type or extended but actual value is within the root

effective size constraint-> 0 .. 7

eff. alph. constraint-> 10 char.

```
string    IA5String  (SIZE (0..3) ^ FROM ("A"..”E") UNION
                      SIZE (4..7) ^ FROM  ("a”.. ”e"), ... )      ::=”abcd"
```



PER: Character string types - 3

- Number of bits/character in constrained character sets
  - in aligned PER: number of bits/char = 2^i
  - in unaligned PER: number of bits/char = 0..max

ALIGNED PER!	No. of char.s	No. of characters in the effective alphabet constraint						
No.bit/character		32	16	8	4	2	1	0
NumericString	11	/	/	/	5..11	3..4	2	1
PrintableString	74			17..74	5..16			
VisibleString	95			17..95				
IA5String	128			17..128				
BMPString	65536		257..65536	17..256				
UniversalString	4E+09	$2^{16}+1$ ..2^{32}	257..65536					

Characters are coded according to remapped codes

Characters are coded according to the original table

PER: Character string types - 5

- Known-multiplier character string types, extended constraint, actual value is out of the root
  - extension bit =1
  - No optimized coding for bits/characters (e.g. IA5String -> 8 bits, UniversalString -> 32 bits)
  - General length encoding is used, length indicates characters

Coding of uncon-strained type

- NOT known-multiplier character string types
  - Number of bits/character is variable
  - No constraint is PER-visible
  - Encoded using general length forms; length identifies no. of octets
  - Constraint is not PER-visible -> NEVER an extension bit is added, even if the ASN.1 type is extensible!

PER visibility of subtypes

Summary

Yes — Subtyping applicable

Subtyping PER-visible

	Bit String	Boolean	Choice	Embedded-pdv	Enumerated	External	Instance-of	Integer	Null	Object class field type	Object Identifier	Octet String	open type	Real	Relative ObjectIdentifier	Restricted Character String Types	Sequence	Sequence-of	Set	Set-of	Unrestricted Character String
Single Value	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Contained Subtype	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes (1)	Yes	Yes	Yes	Yes	No
Value Range	No	No	No	No	No	No	No	Yes	No	No	No	No	No	Yes	No	Yes (1)	No	No	No	No	No
Size Constraint	Yes	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	Yes (1)	No	Yes	No	Yes	Yes
Permitted Alphabet	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes (2)	No	No	No	No	No
Type constraint	No	No	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No
Inner Subtyping	No	No	Yes	Yes (3)	No	Yes	Yes	No	No	No	No	No	No	Yes	No	No	Yes	Yes	Yes	Yes	Yes (3)

(1) - for known-multiplier character types only

(2) - for non-extendable known-multiplier character types only

(3) - when an inner type restricts the "syntaxes" to a single value, or when "identification" is restricted to the "fixed" alternative