KLV : LENGTH : LA TAILLE DES DONNÉES

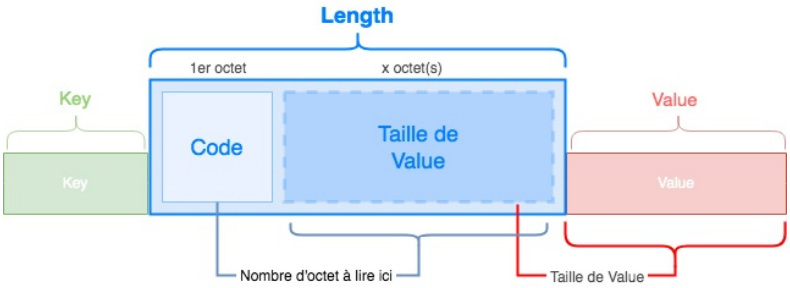
La taille est encodée selon le format BER (Basic Encoding Rules) et ASN.1 - normalisée dans X.690 ITU-T et ISO/IEC 8825-1, et dans la documentation SMPTE **ST-377-1-2011** - MXF - File Format Specification, paragraphe **6.3.4 - KLV Lengths**.

Length stocke la valeur qui donne la taille totale de **Value**.

```
Valeur dans Length = Taille de Value
```

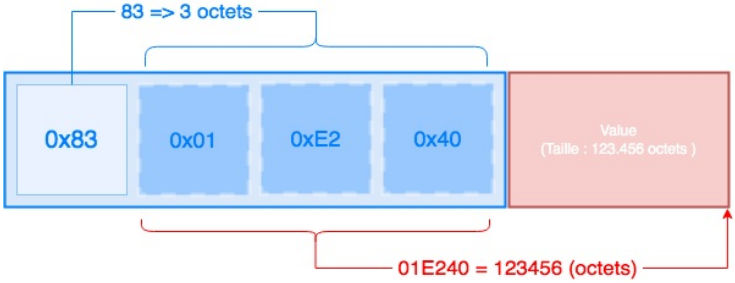
Length a une particularité : sa propre taille est variable, elle est comprise entre 1 octet et 8 octets. La seule manière de déterminer sa taille réelle est de lire son premier octet.

Voici la représentation d'un **Length** spécifique et plutôt commun :



Comme vous le voyez, nous allons devoir lire le 1er octet qui nous donnera une valeur. Cette valeur nous donnera un chiffre nous indiquant le nombre d'octets que nous devrons lire après le premier octet afin de connaître la véritable taille de **Value**.

Voici l'exemple en détail d'un **Length** :



Le premier octet est `0x83` : c'est une valeur spécifique nous indiquant que :

- Length est encodé sous une forme longue (**long-form coded**)
- Qu'il faudra lire les 3 octets suivants pour déterminer la taille de **Value**

Dans notre cas, les 3 octets suivants seront `0x01`, `0xE2` et `0x40`. Cette suite est notre taille pour **Value**. Si nous convertissons `01E240`, de l'hexadécimal en décimal, nous obtenons un nombre entier `123456` : **Value** a donc une taille de **123.456 octets**.

Voici un exemple d'output d'un MXF avec des **Length** débutants par `0x83`, la 3eme colonne (**ber**) indique le **Length** en hexadecimal :

offset	uuid	ber	data-size	data	name
0	060e2b34.02050101.0d010201.01020400	83.000078	120	00010002000000010000000000000000	Partition Pack - Header - Closed & Comp
140	060e2b34.02050101.0d010201.01050100	83.000584	1412	0000004e000000123c0a060e2b340101	Primer Pack
1572	060e2b34.02530101.0d010101.01012f00	83.0000be	190	3c0a001017f71d06d7b640ef9d672550	Preface
1782	060e2b34.02530101.0d010101.01013000	83.0000a2	162	3c0a00109699c30a295445b99f5019f2	Identification
1964	060e2b34.02530101.0d010101.01011800	83.00005c	92	3c0a00109b40021d6abf4943b148b91f	Content Storage
2076	060e2b34.02530101.0d010101.01012300	83.000048	72	3c0a001051000a745f7d44b9a9e8a878	Essence Container Data
2168	060e2b34.02530101.0d010101.01013600	83.0000ae	174	3c0a0010c41a0dcac91c4be89dd28017	Material Package
2362	060e2b34.02530101.0d010101.01013b00	83.000070	112	3c0a0010ecb452147fee46579aaa02f7	Timeline Track
2494	060e2b34.02530101.0d010101.01010f00	83.000050	80	3c0a001026d2748c8e4443d78565bdc1	Sequence
2594	060e2b34.02530101.0d010101.01011400	83.00004b	75	3c0a0010719b398a18a74fe38d080838	Timecode Component
2689	060e2b34.02530101.0d010101.01013b00	83.00006e	110	3c0a00104e84046d552d4725bfae0d89	Timeline Track
2819	060e2b34.02530101.0d010101.01010f00	83.000050	80	3c0a00101e25529f779143c491982c0f	Sequence
2919	060e2b34.02530101.0d010101.01011100	83.00006c	108	3c0a0010354dbb9b457b4092b9dc96f0	Source Clip
3047	060e2b34.02530101.0d010101.01013700	83.000116	278	3c0a0010d6225af012f94c0087c24abe	Source Package
3345	060e2b34.02530101.0d010101.01013b00	83.000070	112	3c0a001034f1b796aff74676bfaef0ea	Timeline Track
3477	060e2b34.02530101.0d010101.01010f00	83.000050	80	3c0a00105df2d0be096e43e5afcb63e9	Sequence
3577	060e2b34.02530101.0d010101.01011400	83.00004b	75	3c0a00105f10db9e0cb84eb1974ff299	Timecode Component
3672	060e2b34.02530101.0d010101.01013b00	83.00006e	110	3c0a0010ae239ae677d94afd84cf51d1	Timeline Track
3802	060e2b34.02530101.0d010101.01010f00	83.000050	80	3c0a0010cb4b54dd6b894f709e35340b	Sequence
3902	060e2b34.02530101.0d010101.01011100	83.00006c	108	3c0a00108897716af0a549ed87db63f8	Source Clip
4030	060e2b34.02530101.0d010101.01012900	83.0000a9	169	3c0a0010e4a7cf73087345a9a5a88cfa	RGBA Essence Descriptor
4219	060e2b34.02530101.0d010101.01015a00	83.0000ae	174	3c0a00105d70488ea6fa407bae04f9e1	JPEG2000 Picture Sub-Descriptor
4413	060e2b34.01010102.03010210.01000000	83.002eaf	11951	00000000000000000000000000000000	KLV Fill item
16384	060e2b34.02050101.0d010201.01030400	83.000078	120	00010002000000010000000000004000	Partition Pack - Body - Closed & Comple
16524	060e2b34.01020101.0d010301.15010801	83.092e8e	601742	ff4fff51002f00030000080000000438	Picture Essence - Line Wrapped Data, No
618286	060e2b34.01020101.0d010301.15010801	83.092e8e	601742	ff4fff51002f00030000080000000438	Picture Essence - Line Wrapped Data, No
1220048	060e2b34.01020101.0d010301.15010801	83.092e8e	601742	ff4fff51002f00030000080000000438	Picture Essence - Line Wrapped Data, No

```
1821810 | 060e2b34.01020101.0d010301.15010801 | 83.092e8e :     601742 | ff4fff51002f00030000080000000438 | Picture Essence - Line Wrapped Data, No
2423572 | 060e2b34.01020101.0d010301.15010801 | 83.092e8e :     601742 | ff4fff51002f00030000080000000438 | Picture Essence - Line Wrapped Data, No
3025334 | 060e2b34.02050101.0d010201.01040400 | 83.000078 :        120 | 0001000200000001000000000002e29b6 | Partition Pack - Footer - Closed & Comp
3025474 | 060e2b34.02530101.0d010201.01100100 | 83.0000af :        175 | 3c0a0010488decd9f83d4147ab3a75a9 | Index Table Segment (53)
3025669 | 060e2b34.02050101.0d010201.01110100 | 83.000028 :         40 | 00000000000000000000000000000001 | Random Index Pack
```

Pour le coup, vous vous demandez pourquoi la valeur `0x83` représente 3 octets alors que `0x83` est égale à 131 en décimal ?

Et si je vous disais que **Length** ne débute pas forcément par `0x83` ? Non, ne partez pas ! Revenez, je vous jure, ça va bien se passer ! (ou presque)

Length peut aussi débuter avec d'autres valeurs comme dans cet output de MXF, regardez toujours la 3ème colonne (**ber**) :

```
offset | uuid                                | ber       :        data-size |                                data | name
     0 | 060e2b34.02050101.0d010201.01020400 | 83.000068 :              104 | 0001000300000200000000000000000000 | Partition Pack - Header - Closed &
   124 | 060e2b34.01010102.03010210.01000000 | 83.000170 :              368 | 00000000000000000000000000000000 | KLV Fill item
   512 | 060e2b34.02050101.0d010201.01050100 | 82.0710   :             1808 | 0000006400000123c0a060e2b340101 | Primer Pack
  2339 | 060e2b34.01010102.03010210.01000000 | 83.0000c9 :              201 | 00000000000000000000000000000000 | KLV Fill item
  2560 | 060e2b34.02530101.0d010101.01012f00 | 81.9a     :              154 | 3c0a0010adab44242f254dc792ff29bd | Preface
  2732 | 060e2b34.02530101.0d010101.01013000 | 81.c4     :              196 | 3c0a0010adab44242f254dc792ff29bd | Identification
  2946 | 060e2b34.02530101.0d010101.01011800 | 5c        :               92 | 3c0a0010adab44242f254dc792ff29bd | Content Storage
  3055 | 060e2b34.02530101.0d010101.01013600 | 81.b0     :              176 | 3c0a0010adab44242f254dc792ff29bd | Material Package
  3249 | 060e2b34.02530101.0d010101.01013b00 | 50        :               80 | 3c0a0010adab44242f254dc792ff29bd | Timeline Track
  3346 | 060e2b34.02530101.0d010101.01010f00 | 50        :               80 | 3c0a0010adab44242f254dc792ff29bd | Sequence
  3443 | 060e2b34.02530101.0d010101.01011400 | 4b        :               75 | 3c0a0010adab44242f254dc792ff29bd | Timecode Component
  3535 | 060e2b34.02530101.0d010101.01013b00 | 50        :               80 | 3c0a0010adab44242f254dc792ff29bd | Timeline Track
  3632 | 060e2b34.02530101.0d010101.01010f00 | 50        :               80 | 3c0a0010adab44242f254dc792ff29bd | Sequence
  3729 | 060e2b34.02530101.0d010101.01011100 | 6c        :              108 | 3c0a0010adab44242f254dc792ff29bd | Source Clip
  3854 | 060e2b34.02530101.0d010101.01013700 | 82.0118   :              280 | 3c0a0010adab44242f254dc792ff29bd | Source Package
  (..)
```

Et le pourquoi est simple ... c'est l'alcool.

Enfin, pour de vrai, c'est à cause des règles d'encodage du **Length** : si vous vous souvenez, je vous avais dit que la taille même de **Length** pouvait être variable. Et c'est à cause de cela, que nous avons des valeurs différentes.

L'ENCODAGE DE LENGTH : C'EST QUOI CE BORDEL ?

Dans la documentation SMPTE, il est indiqué que **Length** respecte l'encodage BER/ASN.1, c'est une sorte de formatage binaire. Vous pouvez vous amuser à lire la documentation BER/ASN.1 notamment la partie sur l'encodage d'un entier non-signée (unsigned inter), c'est relativement long et totalement chiant et ... ne vous servira à rien : **Length** ne respecte (presque) pas ce formatage. Je viens de vous éviter des jours d'incompréhensions.

> **Note**
> Cette partie est totalement dispensable dans la compréhension globale de ce qu'est un KLV.
> Si vous avez compris que `0x83` = 3 octets, alors vous avez déjà l'essentiel.

EXPLICATIONS DE CE BORDEL

Si vous vous souvenez, j'ai dit que **Length** avait une taille variable. Cela est dû au fait que **Length** possèdent deux formats, dont l'un n'est pas fixe :

- Un format court (**short-form coding**) : avec une valeur de 1er octet comprises entre `0x00` et `0x7F` : sa taille est fixe et de 1 octet.
- Un format long (**long-form coding**) : avec une valeur de 1er octet comprises entre `0x80` et `0xFF` : sa taille est variable et peut monter jusqu'à 8 octets.

On voit assez rapidement que notre `0x83` fait partie des formats longs. (pour les plus lents : `0x83` est supérieur à `0x80` :)

Mais qu'est ce qui diffère entre le format long et le format court ? Et pourquoi cette séparation entre `0x7F` et `0x80` :

La réponse est **un bit de différence**.

Le **1er bit** de l'octet **Length** détermine le type de format :

- Si c'est 0, alors le format sera **court**
- Si c'est 1, alors le format sera **long**

Pour mieux comprendre, voyons leurs formes binaires avec leurs valeurs minimales et maximales :

Format court

Le 1er bit ne changeant pas, il doit être '0', on utilise donc les 7 autres bits :

1	2	3	4	5	6	7	8	Valeur	Décimal
0	Ø	Ø	Ø	Ø	Ø	Ø	Ø	`0x00`	0
0	1	1	1	1	1	1	1	`0x7F`	127

Format long

Le 1er bit ne changeant pas, il doit être '1', on utilise donc aussi les 7 autres bits :

1	2	3	4	5	6	7	8	Valeur	Décimal
1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	`0x80`	128
1	1	1	1	1	1	1	1	`0xFF`	255

Si le 1er bit est utilisé pour déterminer son format, les 7 autres servent pour encoder une valeur :

- Pour le **format court**, ses derniers 7 bits serviront à encoder **directement** la taille de **Value**.
- Pour le **format long**, ses 7 derniers bits serviront à encoder la taille réelle de **Length**, ou plus exactement, le nombre d'octet qu'il faudra lire après qui donneront la taille de **Value**.

Le format long sert à compenser un problème du format court : celui de pouvoir encoder des valeurs au delà de `127`. Le format long va définir une valeur indiquant le nombre d'octets supplémentaire qu'il va utiliser pour encoder la véritable taille de **Value**.

Cette structure de données inclut : une clef, une taille et une valeur. Oui, encore du KLV (mais ils appellent cela TLV, l'alcool je vous dis !).

Analysons plus en détail pourquoi ce code spécifique vaut `0x83` et pas une autre valeur.

Pour comprendre, nous allons d'abord voir pourquoi nous sommes avec un chiffre au dessus de `0x80`.

Voyons comment nous encodons `0x80` en binaire :

```
  1   0   0   0   0   0   0   0
```

Nous voyons tout de suite quelque chose qui nous saute au visage : le premier bit de notre octet est à 1 !

Mais pourquoi ?

Tout simple parce que certains bits sont des identifiants ou des paramètres qui donneront une définition à cet octet :

```
  +---+---------------------------  Tag class
  +---+---------------------------  Context-specific
  |   |   +-----------------------  Primitive
  |   |   |
  1   0   0   0   0   0   0   0
'------+------'
       `--------------------------  La base BER/ASN.1
```

Les quatre premiers bits représentent :

- `10xx.xxxx` : Une utilisation spécifique (**Context-specific**)
- `xx0x.xxxx` : Un type primitif (**Primitive**)
- `1000.xxxx` : La valeur est de type **chiffre, entier non-signé** (unsigned integer)

Il existe d'autres types - par exemple du texte (string), du nombre à virgule (float), booléen (boolean), etc. - mais cela est en dehors de notre scope, nous n'avons besoin que du type entier non-signé (unsigned int).

Dans les KLV d'un MXF d'un DCP, **la valeur de ce code est compris entre `0x81` et `0x87` inclus** [1] - avec une nette prépondérance à l'utilisation de `0x83`.

Voici un output de MXF avec plusieurs valeurs de BER/ASN.1 (3ème colonne) :

```
  offset | uuid                                 | ber        :        data-size |                                 data | name
       0 | 060e2b34.02050101.0d010201.01020400 | 83.000068  :              104 | 0001000300000200000000000000000 | Partition Pack - Header - Closed
& Complete
     124 | 060e2b34.01010102.03010210.01000000 | 83.000170  :              368 | 00000000000000000000000000000000 | KLV Fill item
     512 | 060e2b34.02050101.0d010201.01050100 | 82.0710    :             1808 | 00000064000000123c0a060e2b340101 | Primer Pack
    2339 | 060e2b34.01010102.03010210.01000000 | 83.0000c9  :              201 | 00000000000000000000000000000000 | KLV Fill item
    2560 | 060e2b34.02530101.0d010101.01012f00 | 81.9a      :              154 | 3c0a0010adab44242f254dc792ff29bd | Preface
    2732 | 060e2b34.02530101.0d010101.01013000 | 81.c4      :              196 | 3c0a0010adab44242f254dc792ff29bd | Identification
```

Vous voyez qu'il existe plusieurs codes possibles. Ici, 81, 82 et 83, avec des tailles variées (après le point de séparation).

Nous allons voir pourquoi cette plage spécifique de `0x81` à `0x87` en reprenant notre exemple du `0x80` et en le splittant en deux parties de 4 bits :

```
  1   0   0   0   0   0   0   0     =  0x80
'------+------' '------+------'
       |              `-------------  La valeur
       `----------------------------  La base BER/ASN.1
```

- La 1ère partie est la base BER/ASN.1 qui va décrire cet octet. Voyez cela comme le "KL" d'un KLV.
- La 2nd partie va intégrer notre valeur, notre chiffre. Voyez cela comme le "V" d'un KLV.

Si nous voulons encoder le chiffre `3` en binaire, avec quatre bits, nous aurions ceci :

```
                0   0   1   1    =    0x3 ou 0x03
              '------+------'
                     `-------------  La valeur (3)
```

Mais pour compléter l'octet, nous devons intégrer aussi la base avec ses paramètres BER / ASN.1 :

```
 1   0   0   0
'------+------'
       `----------------------------  La base BER/ASN.1 (unsigned int)
```

Si nous intégrons la base `0x80` et la valeur `0x03`, nous avons ceci :

```
.-------------. .-------------.
 1   0   0   0   0   0   0   0    =    0x80
 0   0   0   0   0   0   1   1    =    0x03  +
'-------------' '-------------'
.-------------. .-------------.
 1   0   0   0   0   0   1   1    =    0x83
'------+------' '------+------'
       |              `-------------  La valeur
       `----------------------------  La base BER/ASN.1
```

Nous voyons aussi qu'avec un simple masquage binaire (bit-mask) - ou même un simple calcul - nous pouvons récupérer la valeur très facilement :

```
10000011 - 10000000  =  00000011  =>  3
    0x83 -     0x80  =      0x03  =>  3
```

Vous me direz pourquoi avoir plusieurs codes ?

Tout simplement parce qu'on ne peut encoder certaines valeurs avec si peu d'octets disponibles : Avec seulement 3 octets, nous ne pourrons avoir qu'une valeur maximum de `0xFFFFFF` (soit `16.777.215`) : nos données ne pourront dépasser 16 Mo.

D'où le fait de proposer des codes allant jusqu'à `0x87`, nous permettant d'avoir 7 octets pour encoder des entiers non-signés plus long, par exemple `0xFFFFFFFFFFFFFF` donne un chiffre de plus de 72 millions de milliards - donc plus de 72 Peta-octets !

Voici un tableau récapitulatif des différentes valeurs possibles de `0x81` à `0x87` :

1er Octet	Binaire	Octet à lire après 1er octet	Valeur maximale	Exemple d'utilisation
`0x81`	`1000-0001`	1	255	**81**05 = 5
`0x82`	`1000-0010`	2	65.535 (65 Ko)	**82**0AFF = 2815
`0x83`	`1000-0011`	3	16.777.215 (16 Mo)	**83**0B42DE = 738.014
`0x84`	`1000-0100`	4	4.294.967.295 (4 Go)	**84**F0A80123 = 4.037.542.179
...				
`0x87`	`1000-0111`	7	72.057.594.037.927.935 (72 Po)	**87**ABCDEF12345678 = 48.358.647.703.819.896

Selon les besoins, on utilisera le bon code pour le premier octet. Par exemple, si vous voulez définir une taille de **Value** de 2 Go, vous voyez rapidement qu'il faudra au minimum le code `0x84` :

```
 1   0   0   0   0   1   0   0      =  0x84
'------+------' '------+------'
       |              `-------------  La valeur (4)
       `----------------------------  La base BER/ASN.1 (type unsigned int)
```

Avec notre code `0x84`, nous aurons donc 4 octets de plus après de disponible pour encoder notre valeur (2 Go). 2 Go faisant `2000000000` octets soit, en hexadécimal, `0x77359400`. Nous aurons donc un **Length** sous cette forme complète :

```
              4 octets
   .-->|----------------------|
   |
 code     1     2      3     4

 0x84   0x77  0x35   0x94  0x00    = 2000000000
'----' '----------+------------'
  |              |
  |              `-------  La taille de notre Value (2.000.000.000 = 2G) sur 4 octets
  `----------------------  Le code (type: unsigned int, taille des données prévues: 4 octets)
```

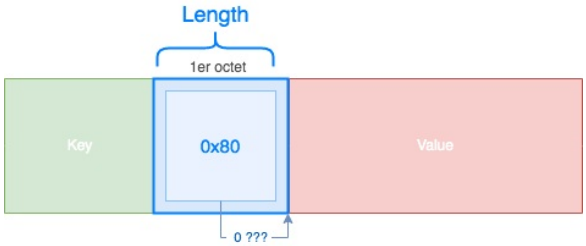
Si vous avez compris cela, c'est parfait, vous comprendrez maintenant pourquoi dans un KLV de MXF de DCP, vous aurez la plupart du temps, un **Length** démarrant par `0x83` et parfois à `0x82` ou encore `0x81`, etc.

Le `0x83` sera largement utilisée, car il permet d'avoir une taille de **Value** de 16 Mo maximum, ce qui est largement suffisant pour une frame au format JPEG2000 même à la résolution 4K.

Vous remarquerez que `0x80` n'est pas intégrée dans la tableau : il fait partie des exceptions.

LES EXCEPTIONS

La valeur `0x80` est particulière, il indique qu'il n'y a aucun octet après pour la valeur.



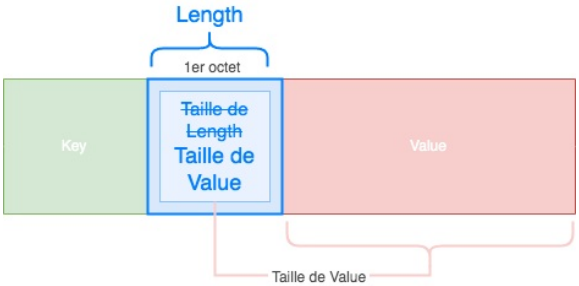
On voit bien que si on interprète `0x80` comme les autres, le nombre d'octet devrait être de 0 et donc, on va lire dans la partie **Value**, c'est impossible : il n'y a donc aucune donnée exploitable pour la taille.

Au niveau des normes, ce cas de figure est considéré comme une "*taille non-déterminée*" :

- Soit il n'y a aucune donnée
- Soit il faut (probablement) lire toutes les données (soit jusqu'à la fin, soit jusqu'à détecter un nouvel en-tête de KLV). La norme indique qu'il faut utiliser "*une méthode alternative pour trouver la fin des données*". Ce qui est en langage profane veut dire "démerdez-vous".

  Je n'ai encore jamais rencontré ce cas de figure dans un MXF de DCP, il peut être utilisé lors de streaming où il est impossible de connaître la taille auparavant. La norme indique de ne jamais utiliser ce code pour des fichiers MXF.

Autre exception (un peu plus répandue que le précédent) : si le premier octet est une valeur non comprise entre `80` et `87`. Cela veut donc dire que **le premier octet est directement la taille de Value**.



Par exemple, si le premier octet est `0xAA`, alors la taille des données sera de 170 octets (`0xAA` = 170 en décimal).

Voyez dans l'output de ce MXF, une multitude de cette forme de **Length** (toujours la 3eme colonne) :

```
offset | uuid                                 | ber  : size |                                data | name
  3249 | 060e2b34.02530101.0d010101.01013b00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Timeline Track
  3346 | 060e2b34.02530101.0d010101.01010f00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Sequence
  3443 | 060e2b34.02530101.0d010101.01011400 | 4b   :   75  | 3c0a0010adab44242f254dc792ff29bd | Timecode Component
  3535 | 060e2b34.02530101.0d010101.01013b00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Timeline Track
  3632 | 060e2b34.02530101.0d010101.01010f00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Sequence
  3729 | 060e2b34.02530101.0d010101.01011100 | 6c   : 108   | 3c0a0010adab44242f254dc792ff29bd | Source Clip
  (...)
  4153 | 060e2b34.02530101.0d010101.01013b00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Timeline Track
  4250 | 060e2b34.02530101.0d010101.01010f00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Sequence
  4347 | 060e2b34.02530101.0d010101.01011400 | 4b   :   75  | 3c0a0010adab44242f254dc792ff29bd | Timecode Component
  4439 | 060e2b34.02530101.0d010101.01013b00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Timeline Track
  4536 | 060e2b34.02530101.0d010101.01010f00 | 50   :   80  | 3c0a0010adab44242f254dc792ff29bd | Sequence
  4633 | 060e2b34.02530101.0d010101.01011100 | 6c   : 108   | 3c0a0010adab44242f254dc792ff29bd | Source Clip
  (...)
  5010 | 060e2b34.02530101.0d010101.01012300 | 48   :   72  | 3c0a0010adab44242f254dc792ff29bd | Essence Container Data
```

Ils l'appellent cela un "**short form coding**" (à contrario de l'autre forme appelée "**long form coding**").

Je la considère comme une hérésie, en sachant qu'on manipule des fichiers pouvant peser des giga-octets, on est pas un octet près pour rester dans le "long form coding" et avoir un codage propre et régulier.

Je vous recommande de ne pas utiliser cette facilité dans vos MXF. Par contre, si vous écrivez un lecteur MXF, vous devrez supporter ces exceptions bizarroïdes.

A noter

Il n'est pas obligatoire d'encoder une valeur de taille avec une valeur précise pour le premier octet.

Donnons un exemple : vous voulez encoder le chiffre 170 (`0xAA`). Si vous regardez le tableau, on voit que `0x81` accepte une valeur de 0 à 255. On peut donc utiliser le code `0x81` et encoder notre 170 comme-ci :

`0x81 + 0xAA = 0x81AA`

Mais on peut tout autant utiliser les autres codes de premier octets.
En effet, rien n'empêche d'encoder 170 sous ces autres formes :

1er octet	Encodage de 170 (`0xAA`)
0x81	0x81**AA**
0x82	0x8200**AA**
0x83	0x830000**AA**
…	…
0x87	0x870000000000**AA**

Toutes ces valeurs d'encodage sont égales à 170.

Bien évidemment, vous ne pourrez pas faire l'inverse : Si vous voulez encoder un très grand nombre, vous ne pourrez pas le faire avec un code dont le maximum est inférieur à ce nombre.

CONCLUSION

Cela a été un gros morceau pour pas grand-chose ;-)

Normalement, vous comprenez maintenant pourquoi **Length** a cette forme, nous pouvons donc passer à la suite très importante : **Value**

NOTES

1. Sauf pour les exceptions, voir le paragraphe **Les exceptions** dans ce chapitre. ↩