MXF : CODES & FILES

JS-MXF

Prototype of MXF analyzer without needs to upload the MXF file : everything is analyzed in your local machine.

You can test with this MXF file.

DRAG YOUR MXF HERE

0 elements found
0 %

GET THE MXF IDENTIFIER (ASDCPLIB)

```
$ asdcp-info -i "picture.mxf" | awk '/AssetUUID/ { print $2 }'
3bd3d849-117b-46b0-bc45-3d3228c987c6
```

GET THE MXF IDENTIFIER (PYTHON)

The prototype is available here mxf-get-assetuid.py

```
$ mxf-get-assetuid.py "jp2k_video.mxf"
3bd3d849-117b-46b0-bc45-3d3228c987c6
```

GET THE CRYPTOGRAPHIC-KEYID OF THE MXF (ASDCPLIB)

```
$ asdcp-info -i "picture.mxf" | awk '/CryptographicKeyID/ { print $2 }'
cf2ab7c6-c00f-4d52-aae2-3c3396a89b93
```

GET THE CRYPTOGRAPHIC-KEYID OF THE MXF (PYTHON)

TODO

SHOW THE SMPTE UNIVERSAL LABEL - 16 OCTETS (PYTHON)

4x4 octets Format :

```
bytes = b'\x06\x0e\x2b\x34\x02\x05\x01\x01\x0d\x01\x02\x01\x01\x02\x01\x00'
print("{:02x}{:02x}{:02x}{:02x}.{:02x}{:02x}{:02x}{:02x}.{:02x}{:02x}{:02x}{:02x}.{:02x}{:02x}{:02x}{:02x}".format(*
060e2b34.02050101.0d010201.01020100
```

UUID Format :

```
bytes = b'\x06\x0e\x2b\x34\x02\x05\x01\x01\x0d\x01\x02\x01\x01\x02\x01\x00'
print("{:02x}{:02x}{:02x}{:02x}-{:02x}{:02x}-{:02x}{:02x}-{:02x}{:02x}-{:02x}{:02x}{:02x}{:02x}{:02x}{:02x}".format(*
060e2b34-0205-0101-0d01-020101020100
```

SHOW THE SMPTE UNIVERSAL LABEL - 12 OCTETS (PYTHON)

3x4 octets Format:

```
bytes = b'\x06\x0e\x2b\x34\x02\x05\x01\x01\x0d\x01\x02\x01'
print("{:02x}{:02x}{:02x}{:02x}.{:02x}{:02x}{:02x}{:02x}.{:02x}{:02x}{:02x}{:02x}".format(*bytes))
060e2b34.02050101.0d010201
```

BER : DETERMINATE IF A BER IS SHORT-FORM OR LONG-FORM (C/C++)

```
bool ber_get_form(uint8_t bercode) {
    return (bool)(bercode & 0x80);
}

// Exemple d'utilisation
ber_get_form(0x83);  /* long form */
>>> true

ber_get_form(0x10);  /* short form */
>>> false
```

BER : GET THE BER SIZE ACCORDING TO THE HEADER CODE (C/C++)

```
uint8_t ber_get_size(uint8_t bercode) {
    if (bercode >= 0x80)
        return (uint8_t)(bercode ^ 0x80);
    return 0;
}

// Usage Example - (0x83 == BER à 3)
ber_get_size(0x83);
>>> 3
```

BER : CALCULATE THE SIZE (C/C++)

Réecrire cette fonction

```
unsigned long int ber_get_datasize(char *berdata, unsigned int bersize) {
    unsigned long int size = 0;
    unsigned short int index, shift, i;
    for(i = 0; i < bersize; i++) {
        index = 1-(i-(bersize-1))-1;   // swap-index
        shift = 8 * i;
        size |= ((unsigned long int)berdata[index] & 0xFF) << shift;
    }
    return size;
}

// Usage Example - (0x000170 == 368)
ber_get_datasize("\x00\x01\x70", 3);
>>> 368   /* octets */
```

BER : CALCULATE THE SIZE FROM A BER 0X83 (PYTHON)

Rewrite this function to handle all BER

```
def ber(bercode):
    return int.from_bytes(bercode, byteorder='big') ^ 0x83000000
```

IS SMTPE UNIVERSAL LABEL (C/C++)

Check if the MXF (or the KLV) is a SMPTE Universal Label (SMPTE UL) :

```
bool is_smpte_ul(char *buffer) {
    return (bool)(strncmp("\x06\x0e\x2b\x34", buffer, 4) == 0);
}
```

SHOW THE SMPTE UNIVERSAL LABEL - 16 OCTETS (C/C++)

Change 4x4o => uuid format

```
int smpte_uuid(char *src, char *dst) {
    return snprintf(dst,
        36,
        "%02x%02x%02x%02x.%02x%02x%02x%02x.%02x%02x%02x%02x.%02x%02x%02x%02x",
        src[0],  src[1],  src[2],  src[3],
        src[4],  src[5],  src[6],  src[7],
        src[8],  src[9],  src[10], src[11],
        src[12], src[13], src[14], src[15]
    );
}

// Exemple d'utilisation
char uuid_src[16] = "\x06\x0e\x2b\x34\x01\x02\x03\x04\x05\x06\x07\x08\x09\x1A\x1B\x1C";
char uuid_dst[36];
smpte_uuid(uuid_src, uuid_dst);
>>> uuid_dst = "060e2b34.01020304.05060708.091a1b1c"
```

READ AN MXF AND GET THE UNIVERSAL LABEL, THE SIZE, AND PART OF THE DATA (C/C++)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

/* this functions are defined in another place in documentation */
extern int smpte_uuid(char *, char *);
extern bool is_smpte_ul(char *);
extern uint8_t ber_get_datasize(char *, unsigned int);
extern uint8_t ber_get_size(uint8_t);
extern uint8_t ber_get_form(uint8_t);

/*
    This code is neither optimized nor adheres to best C programming practices.
    It's intended for learning purposes: for example, declarations and malloc calls
    are placed inside the while loop when they ideally shouldn’t be.
    Some parts have been merged to avoid unnecessary back-and-forth operations.
*/

int main(int argc, char **argv) {

    if (argc < 2) {
        printf("Usage: %s <mxf>\n", argv[0]);
```

```
        return 1;
    }

    FILE *fh = fopen(argv[1], "r");

    while(!feof(fh)) {

        // Getting the file position
        // (optional: just for the display)
        unsigned long long int offset;
        offset = ftell(fh);
        printf("%12llu | ", offset);

        // -------------------------
        // ---------- Key ----------
        // -------------------------

        // Get SMPTE UUID
        char key[16];
        fread(key, 16, 1, fh);

        // Show the key (UUID format)
        char *uuid = (char *)malloc(sizeof(char *) * 36);
        smpte_uuid(key, uuid);
        printf("%s | ", uuid);
        free(uuid);

        // Stop if the key isn't an UUID SMPTE (optional)
        if( !is_smpte_ul(key) ) {
            printf("\n**SMPTE UL not found**\n");
            break;
        }

        // -------------------------
        // -------- Length ---------
        // -------------------------

        // Get the first octet of the BER (ex. 0x83)
        uint8_t ber_code;
        fread(&ber_code, 1, 1, fh);

        // Calculate the real size of the data
        unsigned long long int data_size;
        if ( ber_get_form(ber_code) == 1 ) {      // Long form (>= 0x80)
            // get size of length
            unsigned int ber_size;
            ber_size = ber_get_size(ber_code);
            // read full length data
            char ber[ber_size];
            fread(ber, ber_size, 1, fh);
            // convert length data to size of data
            data_size = ber_get_datasize(ber, ber_size);
        } else {                                  // short BER (< 0x80)
            // get size of data
            data_size = ber_code;
        }
        printf("%10llu | ", data_size);

        // -------------------------
        // --------- Value ---------
        // -------------------------

        char *data = (char *)malloc(sizeof(char *) * data_size);
        fread(data, data_size, 1, fh);

        // Show the 16 first octets
        for(int i=0; i<16; i++) {
            printf("%02x", data[i] & 0xFF);
        }
        printf("\n");

        free(data);
    }

    fclose(fh);
```

```
    return 0;
}

// Usage :
$ gcc mxf-reader.c -o mxf-reader

$ ./mxf-reader
$   Usage: ./mxf-reader <mxf>

$ ./mxf-reader video.mxf
           0 | 060e2b34.02050101.0d010201.01020400 |        104 | 0001000300000200000000000000000
         124 | 060e2b34.01010102.03010210.01000000 |        368 | 0000000000000000000000000000000
         512 | 060e2b34.02050101.0d010201.01050100 |       1772 | 0000006200000123c0a060e2b340101
        2303 | 060e2b34.01010102.03010210.01000000 |        237 | 0000000000000000000000000000000
        2560 | 060e2b34.02530101.0d010101.01012f00 |        154 | 3c0a0010adab44242f254dc792ff29bd
        2732 | 060e2b34.02530101.0d010101.01013000 |        168 | 3c0a0010adab44242f254dc792ff29bd
        2918 | 060e2b34.02530101.0d010101.01011800 |         92 | 3c0a0010adab44242f254dc792ff29bd
        3027 | 060e2b34.02530101.0d010101.01013600 |        124 | 3c0a0010adab44242f254dc792ff29bd
         ... |                  ...                |        ... |              ...
    62565888 | 060e2b34.02050101.0d010201.01040400 |        104 | 0001000300002000000000003baae00
    62566012 | 060e2b34.01010102.03010210.01000000 |        368 | 0000000000000000000000000000000
    62566400 | 060e2b34.02530101.0d010201.01100100 |        271 | 3c0a0010adab44242f254dc792ff29bd
    62566691 | 060e2b34.01010102.03010210.01000000 |        201 | 0000000000000000000000000000000
    62566912 | 060e2b34.02050101.0d010201.01110100 |        256 | 0000000000000000000000000000001
    62567187 | 060e2b34.02050101.0d010201.01110100 |        256 | 0000000000000000000000000000001
```

Bug on the last line displayed - to be fixed soon

CRYPTOGRAPHIE

XOR

```
def xor(v1, v2):
    return bytes(a ^ b for a, b in zip(v1, v2))

print(xor(
    b'\x01\x23\x45\x67\x89\xAB\xCD\xEF',
    b'\x45\xA9\x1E\x06\xB4\xCD\x87\x1A'
).hex().upper())
```

AES-128-CBC ENCRYPTION

```
from cryptography.hazmat.primitives.ciphers import ( Cipher, algorithms, modes )
from cryptography.hazmat.backends import default_backend

# The Encryption Key : 'UNE CLEF SECRETE'
key       = b'\x55\x4E\x45\x20\x43\x4C\x45\x46\x20\x53\x45\x43\x52\x45\x54\x45'

# IV not really random ;-)
iv        = b'\x01\x02\x03\x04\x05\x06\x07\x08\x09\x10\x11\x12\x13\x14\x15\x16'

# The plaintext : 'BONJOUR LE MONDE' (sorry, it's in French: 'HELLO WORLD' ;)
plaintext = b'\x42\x4F\x4E\x4A\x4F\x55\x52\x20\x4C\x45\x20\x4D\x4F\x4E\x44\x45'

# Cryptographic Engine Settings
cipher = Cipher(
    algorithms.AES(key),
    modes.CBC(iv),
    backend=default_backend()
)
encryptor = cipher.encryptor()

# Encryption of the plaintext
ciphertext = encryptor.update(plaintext)

# Give :
#   0xEB 0x36 0xD4 0x79
#   0x25 0x6C 0x9E 0x6D
#   0x1B 0xB4 0x11 0x52
#   0x6f 0x00 0x9C 0x1B
print(ciphertext, ciphertext.hex())
```

AES-128-CBC DECRYPTION

```
from cryptography.hazmat.primitives.ciphers import ( Cipher, algorithms, modes )
from cryptography.hazmat.backends import default_backend

# The Decryption Key : 'UNE CLEF SECRETE'
key       = b'\x55\x4E\x45\x20\x43\x4C\x45\x46\x20\x53\x45\x43\x52\x45\x54\x45'

# IV not really random ;-)
iv        = b'\x01\x02\x03\x04\x05\x06\x07\x08\x09\x10\x11\x12\x13\x14\x15\x16'

# The ciphertext (encrypted content)
ciphertext = b'\xeb\x36\xd4\x79\x25\x6c\x9e\x6d\x1b\xb4\x11\x52\x6f\x00\x9c\x1b'

# Cryptographic Engine Settings
cipher = Cipher(
    algorithms.AES(key),
    modes.CBC(iv),
    backend=default_backend()
)
decryptor = cipher.decryptor()

# Decryption of the ciphertext (encrypted content)
plaintext = decryptor.update(ciphertext)

# Give 'BONJOUR LE MONDE'
print(plaintext, plaintext.hex())
```

DECRYPTION OF THE VALUE FROM A KLV ENCRYPTED ESSENCE CONTAINER

This code is an example of a decryption of the 16 first octets of the encrypted content following the "SMPTE-429-6 - Essence Encryption" (KLV JPEG2000, KLV WAV PCM, KLV Dolby Atmos, KLV Subtitles, etc...) standard.

```
#!/usr/bin/env python3

from cryptography.hazmat.primitives.ciphers import ( Cipher, algorithms, modes )
from cryptography.hazmat.backends import default_backend

# Decryption key used to decrypt the MXF content.
key = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

# Initialization Vector (IV)
# Block of 16 octets used to initiate the cryptographic engine
iv  = b'\xc7\x51\x65\x82\xd2\x74\x77\x9e\x09\xf8\x44\x01\x77\x0a\x61\x60'

# "CheckValue" is the 16 first octets that include a fixed-value.
# Decrypted, the content is '43:48:55:4B:43:48:55:4B:43:48:55:4B:43:48:55:4B' or 'CHUKCHUKCHUKCHUK'
checkvalue = b'\x44\x8d\xf3\xbc\x12\x60\x6b\xb2\xe3\xcc\x4a\xed\xec\x31\xbe\x78'

# The 16 first octets of the "Value" field of the KLV (this is the encrypted JPEG2000 header)
data = b'\xbb\x07\x4c\x4f\x11\x37\x30\x6a\x64\x45\xad\x3d\xaf\x0c\xc3\x05'

# Init Cryptographic Engine
cipher = Cipher(
    algorithms.AES(key),
    modes.CBC(iv),
    backend=default_backend()
)
decryptor = cipher.decryptor()

# We add the 16 first octets which is the encrypted "CheckValue"
decryptor.update(checkvalue)

# We add the 16 first octets of our encrypted data
plaintext = decryptor.update(data)

# The decrypted data must be "ff4fff51002f00030000080000000438"
# This is a JPEG2000 Header
print(plaintext.hex())
```

To get the complete (and simplified) version of this tool which allows to read an encrypted MXF and to extract encrypted essences : mxf-encrypted-decryption.py

```
$ mxf-encrypted-decryption.py
Usage: ./parse-klv-encrypted-essence.py <mxf> <aeskey>

$ mxf-encrypted-decryption.py encrypted-key-00000000000000000000000000000000.mxf 00000000000000000000000000000000
060e2b34020401010d010301027e0100 -  40300 - 8300001067bec4fc40de4996aac7fa42...
CryptographicContextLink Length         : 83000010
CryptographicContextLink Value          : 67bec4fc40de4996aac7fa42a6b0ed5e
PlaintextOffset Length                  : 83000008
PlaintextOffset Value                   : 0000000000000000 (0 bytes)
SourceKey Length                        : 83000010
SourceKey Value                         : 060e2b34010201010d01030115010801
SourceLength Length                     : 83000008
SourceLength Value                      : 0000000000009cc8 (40136 bytes)
Encrypted Source Length                 : 83009cf0 (40176 bytes)
Encrypted Source Value - IV             : 765a067b36dfd2e89da94a9c6af0902f
Encrypted Source Value - CheckValue     : 7d95b3c594116732ed0b2d9b13ac5283
Encrypted Source Value - Plaintext Data :
Encrypted Source Value - Encrypted Data : 9c52432ad90a1bba64fd0ac5c604a1c9...
TrackFile ID Length                     : 83000010
TrackFile ID Value                      : 89af85f04a1545ec8a769008829b2029
Sequence Number Length                  : 83000008
Sequence Number Value                   : 0000000000000001
Message Integrity Code (MIC) Length     : 83000014
Message Integrity Code (MIC) Value      : 5b594d66d09cf6ddfda8f6e691e4291ea7097bc8
Plaintext Source Value                  : 40144 bytes
Padding                                 : 8 bytes
```

Note that this tool is focused on JPEG2000 frame extraction, but could work with another essences (some exceptions, see the different chapters for each essence type)

Encrypted MXF samples : See Annex

CREATE AN ENCRYPTED MXF (ASDCPLIB)

```
asdcp-test \
    -L \
    -j deadbeefdeadbeefdeadbeefdeadbeef \
    -k 00000000000000000000000000000000 \
    -c output.mxf \
    ./
```

Expaination:

```
-L : create an SMPTE MXF (not Interop)
-j <keyId>  : defined manually the "Cryptographic Key Id"
-k <aeskey> : the AES key for the encryption
-c <outputfile>
```

CREATE AN ENCRYPTED MXF WITH A PLAINTEXT OFFSET

```
asdcp-test \
    -L \
    -E \
    -j deadbeefdeadbeefdeadbeefdeadbeef \
    -k 00000000000000000000000000000000 \
    -c output.mxf \
    ./
```

Explication :

```
-L : create an SMPTE MXF (not Interop)
-E : no headers encryption (36 first octets)
-j <keyId>  : defined manually the "Cryptographic Key Id"
-k <aeskey> : the AES key for the encryption
-c <outputfile>
```

ESSENCE EXTRACTION FROM AN ENCRYPTED MXF (ASDCPLIB)

```
asdcp-unwrap \
    -v \
    -k 00000000000000000000000000000000 \
    encrypted.mxf \
    extract/
```

Explication :

```
-v : Verbose
-k <aeskey> : the AES key for the decryption
```

ESSENCE EXTRACTION FROM AN ENCRYPTED MXF (MXF-ANALYZER)

Disponible ici : assets/MXF/mxf-analyzer/

```
mxf-analyzer.py \
    -f encrypted.mxf \
    -x extract_directory/ \
    -k 00000000000000000000000000000000
```

CREATE AN ENCRYPTED MXF WITH ASDCPLIB (C++)

This code is a proof-of-concept, it create an MXF with a single encrypted frame : (the code is over-simplified, you

have no check/verify)

```
#include <AS_DCP.h>
#include <KM_prng.h>  /* FortunaRNG */
#include <Metadata.h> /* MXF:: */

using namespace ASDCP;

int main(void) {

    WriterInfo Info;
    JP2K::MXFWriter Writer;
    JP2K::FrameBuffer FrameBuffer(1024 * 1024);
    JP2K::PictureDescriptor PDesc;
    JP2K::SequenceParser Parser;
    // AES
    AESEncContext* Context = 0;
    const byte_t aes_key[16] = {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    };
    Kumu::FortunaRNG RNG;
    byte_t IV_buf[CBC_BLOCK_SIZE];
    // HMAC
    HMACContext* HMAC = 0;

    // Set Header
    Info.LabelSetType = LS_MXF_SMPTE;
    Kumu::GenRandomUUID(Info.ContextID);
    // AssetUUID == TrackFile ID (MIC)
    Kumu::GenRandomUUID(Info.AssetUUID);

    // Set Cryptographic
    Kumu::GenRandomUUID(Info.CryptographicKeyID);
    Info.EncryptedEssence = true;
    Context = new AESEncContext;
    Context->InitKey(aes_key);
    Context->SetIVec(RNG.FillRandom(IV_buf, CBC_BLOCK_SIZE));

    // Set HMAC
    Info.UsesHMAC = true;
    HMAC = new HMACContext;
    HMAC->InitKey(aes_key, Info.LabelSetType);

    // Set Parser from files
    Parser.OpenRead("essences/");
    Parser.FillPictureDescriptor(PDesc);

    // Go to the first file
    Parser.Reset();

    // Open MXF
    Writer.OpenWrite("dump.mxf", Info, PDesc);

    // --- foreach frame -------------------------------------
    // Read each frame (only one here)
    // Each call of ReadFrame() shift to the next frame
    // ReadFrame() returns a zero if no new frame
    Parser.ReadFrame(FrameBuffer);
    FrameBuffer.PlaintextOffset(0);  // force no plaintext
    // Write each frame into MXF (only one here)
    Writer.WriteFrame(FrameBuffer, Context, HMAC);
    // -------------------------------------------------------

    // Close MXF
    Writer.Finalize();

    // Show 256 bytes from JPEG2000
    FrameBuffer.Dump(stderr, 256);
    // Show all metadatas from JPEG2000
    JP2K::PictureDescriptorDump(PDesc);

    return 0;
}
```

To compile it :

```
# You must compile asdcplib first
# to have the libasdcp et libkumu librairies
ASDCPLIB="/chemin/asdcplib/"

g++ -g -O2 \
    -lpthread \
    -Wl,-bind_at_load \
    -DHAVE_OPENSSL=1 \
    -I$(ASDCPLIB)/src/ \
    $(ASDCPLIB)/src/.libs/libasdcp.so \    # Linux
    $(ASDCPLIB)/src/.libs/libkumu.so \     # Linux
    $(ASDCPLIB)/src/.libs/libasdcp.dylib \ # MacOS
    $(ASDCPLIB)/src/.libs/libkumu.dylib \  # MacOS
    `pkg-config openssl --cflags` \
    `pkg-config openssl --libs` \
    asdcplib-create-encrypted-mxf.cpp \
    -o asdcplib-create-encrypted-mxf
```

To run :

```
# You must compile asdcplib first
# to have the libasdcp et libkumu librairies
ASDCPLIB="/chemin/asdcplib/"

# Linux
LD_LIBRARY_PATH="$(ASDCPLIB)/src/.libs:${LD_LIBRARY_PATH}" ./asdcplib-create-encrypted-mxf

# MacOS
DYLD_LIBRARY_PATH="$(ASDCPLIB)/src/.libs:${DYLD_LIBRARY_PATH}" ./asdcplib-create-encrypted-mxf
```

You can find the source code here : asdcplib-create-encrypted-mxf.cpp + Makefile

FILES

- **Tools** :
  - encrypted-decryption.py Allows to read, decrypt and extract data from an encrypted MXF - in Python
  - encrypted-hmac.py Calculation of the Message Integrity Code (MIC) (HMAC-SHA1-128 format) - in Python
  - get-assetuid.py Gives the AssetUUID of the MXF - in Python
  - proto-hmac.c Prototype of algorithm implementation HMAC-SHA1-128 - in C
  - proto-hmac.py Prototype of algorithm implementation HMAC-SHA1-128 - in Python
- **Librairie asdcplib** :
  - hmac.cpp + patch Makefile Derivation Key Generation and Message Integrity Code (MIC) (HMAC-SHA1-128 format) - in C++ - using the ascplib librairie.
  - asdcplib-create-encrypted-mxf.cpp Creation of an encrypted MXF - in C++ - using the asdcplib librairie.
  - asdcplib.plaintextoffset.patch Patch to allows an Plaintext Offset via PLAINTEXT_OFFSET environment variable when we use the asdcplib tools. (quick & dirty patch)
  - create-samples.sh Usage example using asdcp-test to create an MXF.
  - asdcplib-fips-lite.cpp Unit Function to create a derivation key that follows the FIPS 186-2, "General Purpose Random Number Generation" standard - in C++
- **Assets MXF** :
  - **Encrypted MXF** :
    - MXF encrypted.mxf (AES key: `00000000000000000000000000000000`)
    - MXF encrypted-plaintextoffset.mxf (AES key: `00000000000000000000000000000000`)
    - MXF `DCP-INSIDE-CRYPTE` jp2k_video.mxf (AES key: `6e256ec2308835ea1d46d8a359296f38`)

        - MXF `DCP-INSIDE-CRYPTE` wav_audio.mxf (AES key: `f5a3d36ab03412984de4aa313199437a`)
        - Layers **KLV** from encrypted MXFs :
            - KLV encrypted (dir)
            - KLV encrypted-plaintextoffset (dir)
    - **non-encrypted MXF** :
        - MXF `DCP-INSIDE` (non-chiffré) jp2k_video.mxf & wav_audio.mxf
        - MXF audio.smpte.mxf - 2 channels, 48 kHz, 24 bits
        - MXF 2D.mxf - 1 frame 4K JPEG2000 XYZ - 24/1 fps
- **Assets sources** :
    - Frame JPEG2000 4K XYZ (generate by a DVS Clipster 5.10)

    - Son WAV PCM 2 channels 48 kHz 32 bits - Tambour (~182 ms) Replace it by the 24-bit version