
A Padding Detection Algorithm

F.-Otto Witte, TEConcept GmbH

info@teconcept.de

This report describes an algorithm that allows the correct decoding of ISO IEC 11172-3 Layer 3 audio bitstreams (mp3-files) without the usage of the padding bit

The padding problem is a natural consequence of the definition of

- bitrates
- sample / framerates
- header distances (incorrectly defined as framesize) in MPEG audio. The problem arises when the relation between the rate of the uncompressed audio samples grouped in frames and the rate of the compressed bits grouped in slots do not allow to place a certain number of slots within a frame.:

1.0 Background information

In a real time digital transmission systems equidistant samples of an analog signal are transmitted. The rate of the sampling is defined by the transmitter and is based on its timing reference. These samples are transmitted to a receiver. The receiver decodes the signal and converts them again into an analog signal e.g. an analog audio signal.

In order to perform this task the receiver needs the numerical value of the samples to be converted and the exact rate in which the samples have been digitized. The sample rate is recovered by the receiver by means of a clock recovery module during the reception that locks onto the transmitted signal.

MPEG audio compression is based on frames. Within a frame a well defined number of uncompressed audio samples is processed and converted into a compressed signal. The duration t_f of the frame is directly defined by the sample rate „F“ and the number of samples per frame as $t_f = \frac{ns}{F}$ because the time between two samples equals

$\frac{1}{F}$. Considering the compressed signal the question how many bit-slots fit into the time of the frame. The duration of one transferred bit is defined by the bitrate BR to $t_b = \frac{1}{BR}$. Because one slot consist of N bits duration of a slot $t_s = \frac{N}{BR}$. In order to get the number of slots per frame P the duration of the frame has to be divided by the duration of a slot $P = \frac{t_f}{t_s} = \frac{ns}{F} \div \frac{N}{BR} = \frac{ns \cdot BR}{F \cdot N}$

And this is exactly the formula that is given by MPEG for the frame capacity.

$$P = \frac{BR}{N} \cdot \frac{ns}{F}$$

Putting numbers into the formula as shown in Table 1 it becomes obvious that for certain combinations of BR and F a non integer number of slots fit into a frame. As the compression algorithm always processes one frame of uncompressed data and converts them into compressed slots of bits the problem occurs how to generate a non integer number of slots. The most obvious and natural solution to this problem is: Round the number of slots per frame to the next smaller or bigger integer number and do this in a way that in the average for many frames the packet capacity is met as close as possible. This approach is called padding in the MPEG standard.

TABLE 1.

Variables in the Frame Capacity Formula

Variable	Description	Example
N	number of bits per packet	8
BR	the bitrate i.e. the number of bits to be transmitted per second	192000
ns	The number of samples per frame	1152
F	the sample frequency	44100
P	resultant frame capacity	626 46/49

The MPEG standard defines a set of numbers for the 4 variables that occur in the frame capacity formula. The bitrate BR is typically a number that is a multiple of 8 kHz. The number of bits N per slot is set to 8 or 32. The sample rates are either 48, 32 or 44.1 kHz. Half and quarter sample rates are also covered. Only for CD-related sample rates (44.1 kHz) the average frame capacity P becomes a non-integer number. A typical numerical example for a MPEG bitstream of a CD-record results in the following values: $BR = 192\text{ kbit/s}$, $N=8$, $ns=1152$, $F=44.1\text{ kHz}$

In this case P becomes

$$\frac{192000 \cdot 1152}{8 \cdot 44100} = \frac{221184000}{8 \cdot 44100} = \frac{2211840}{8 \cdot 441} = 626 + \frac{46}{49}$$

This means that the average frame capacity is $626\frac{46}{49}$ packets

(bytes). Because only an integer number of slots can be assigned to one frame, a pattern can be applied in which within 49 frames 46 frames are transmitted with 627 slots and 3 frames are transmitted with 626 slots. In this case the average frame capacity for 49

frames is exactly: $\frac{(626 + 1) \cdot 46 + 626 \cdot 3}{49} = \frac{626 \cdot 49 + 46}{49} = 626\frac{46}{49}$.

An encoder could fulfil this condition easily by generating a sequence (15P 1p) (15P 1p) (16P 1p) = 46P+3p with P standing for a slot with 627 frames (with padding slot) and p stands for a frame with 626 slots (without padding slot). This is exactly what happens in most encoders as can be analyzed with the Padding Demonstrator.

It is important to notice that always a regular padding pattern can be defined because the padding capacity formula always generates a rational number.

A receiver can figure out a padding pattern based on the known parameters in the capacity formula. If a receiver manages to detect reliably how big the current frame capacity is (i.e. P' or P'+1) where P' is the biggest integer number that is smaller than P, the signalization of padding is completely superfluous.

The next section describes an algorithm that performs a detection of the header distance without padding signalization and thus shows that the padding signalization is not required for proper decoding of MPEG bitstreams.

2.0 MPEG decoding without padding signalization

The proposed algorithm is from its principle straight forward. It affects an existing MPEG decoder that operates as a memory based digital audio player in only two points: the initial synchronization and the prediction of the next header position.

2.1 Initial Header synchronization

More robust MPEG audio players do not require that a bitstream to be replayed starts exactly with an MPEG header. The only constraint is that the bits are aligned to byte (L2 and L3) or 4byte (L1) borders. The MPEG header consists of 32 bits that are defined in ISO 11172-3 as follows:

20 syncword	12 bits bslbf
19 ID	1 bit bslbf
17 layer	2 bits bslbf
16 protection_bit	1 bit bslbf
12 bitrate_index	4 bits bslbf
10 sampling_frequency	2 bits bslbf
9 padding_bit	1 bit bslbf
8 private_bit	1 bit bslbf
6 mode	2 bits bslbf
4 mode_extension	2 bits bslbf
3 copyright	1 bit bslbf
2 original/home	1 bit bslbf
0 emphasis	2 bits bslbf

- For initial synchronization the bitstream is bitwise scanned for the existence of the header synchronization information that is a sequence of 12 (11)¹ set bits.
- The other 21 bits of the header provide information about the bitrate the Layer, the mode etc. that are required to decode the bitstream and to detect the position of the next header. A header is only accepted if the sanity check passes. It checks for invalid MPEG mode, invalid sample rate, invalid layer and invalid version.

Then a target headersync word is defined that is the result of applying a mask
 $int\ HEADER_SYNC_MASK = 0xFFFE0CCF$ to the header that masks *sync header, layer, sr, mode, copyright, original emphasis and channel* from the header that should be invariant within the bitstream.

Based on the header content the position of the next header (without padding) is calculated. If the detected header was a right one, the calculated next header position is now either correct or one slot (byte) to early in the bitstream.

The algorithm now checks at the calculated header position, whether at this position a new header can be found that matches the headersync word that has been extracted at the first header position.

If the check fails for the first position, it is repeated one byte later. If it fails again, the extracted headersync word is obviously wrong and the search for the first header is resumed. This continues until

1. changes to the header length have been applied in the later MPEG 2 / MPEG 2.5

a valid 2nd header has been found. Then standard decoding of the bitstream starts.

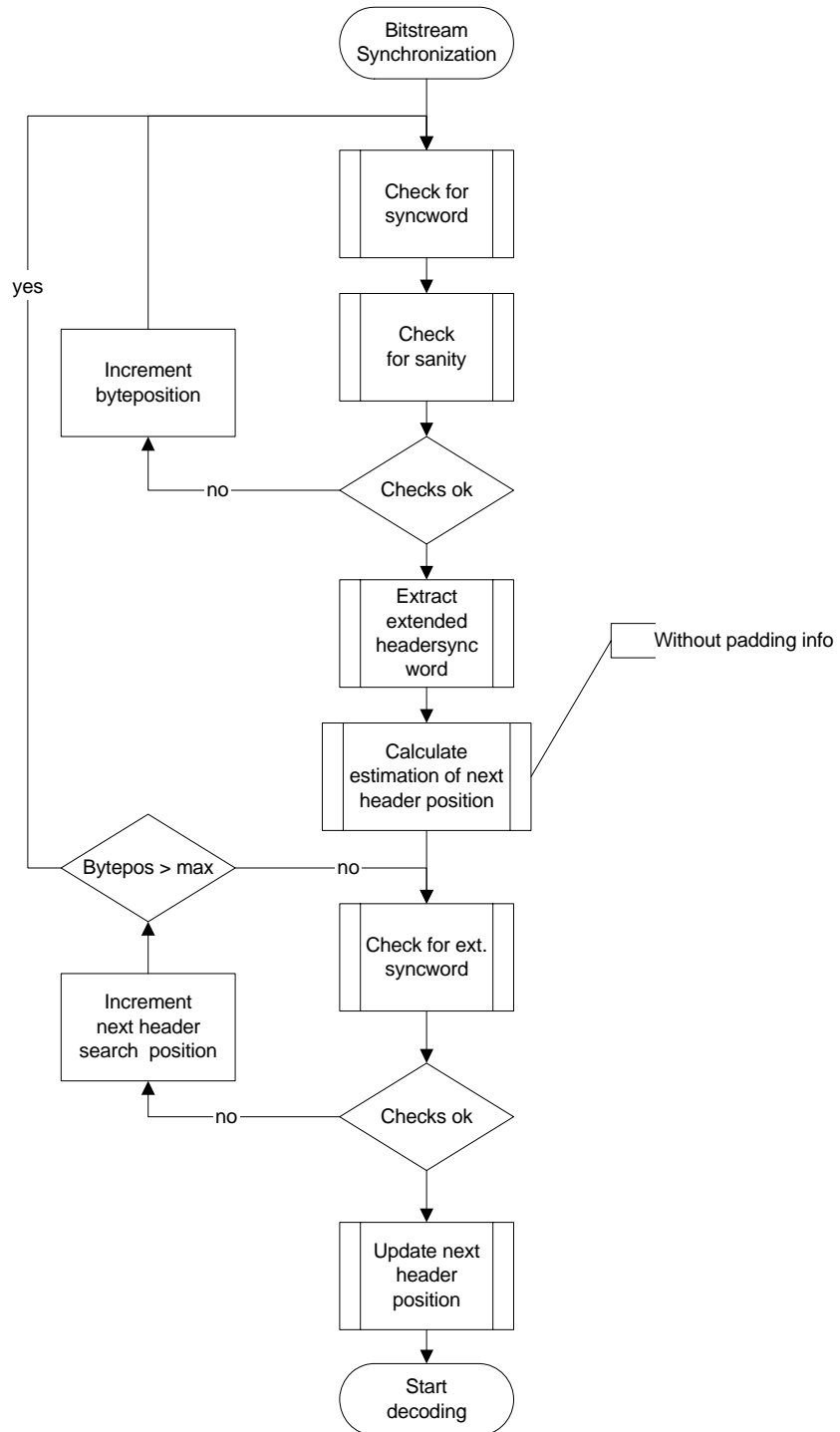


FIGURE 1.

Flowchart of initial header synchronization



2.2 Header synchronization during the MPEG decoding

After initial synchronization the standard decoding works in the following way. Based on the current header, the next header position without padding is calculated.

At the calculated next header position the match with the detected headersync (see initial header synchronization) is checked. If no match is found, the next position is checked until a header is detected or an upper limit for additional checks has been reached. In the proposed algorithm this upper limit has been set to 4 for compatibility with Layer 1. *0xFFEOCCF*. In Table 2 a header

TABLE 2. False matching analysis

	3	3	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0						
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2					
1	Header												Id	Layer ^a	Prot	BRI ^b				FS ^c		Pad	Priv	Mode	M-ext	copy	Orig	Emphasis							
2	F			F			F			E			O				C		C		F														
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1					1	1			1	1			1	1	1	1				
4	1	1	1	1	1	1	1	1	1	1	1	1	x	0	1					BRI != 1111				0	0			x	x			x	x	x	x
5	1	1	1	1	x	0	1	x	BRI != 1111				0	0	x					x	x			x	x			x	x	x	x				
6	Header				Id	Layer	Prot	BRI				FS	Pad	Priv	Mode	M-ext	copy	Orig	Emphasis																

- a. Layer 00 is not allowed
- b. BRI of 1111 is not allowed
- c. FS 11 is not allowed

alignment analysis is given. In line 3 a mask of relevant bits is shown. Line 4 shows a correct header matching. Line 5 shows a match that is tried one byte to the left. Due to a 0 for layer 3 at position 25 and a reserved BRI (=1111) a wrong match is not possible..

After a headersync-match has been found, the distance between headers (sometimes called incorrectly *framelength*) is updated to the new value and passed to the subsequent decoding process.

A false detect of a new header at an early position should be impossible due to the structure of the headermask and the used bits.

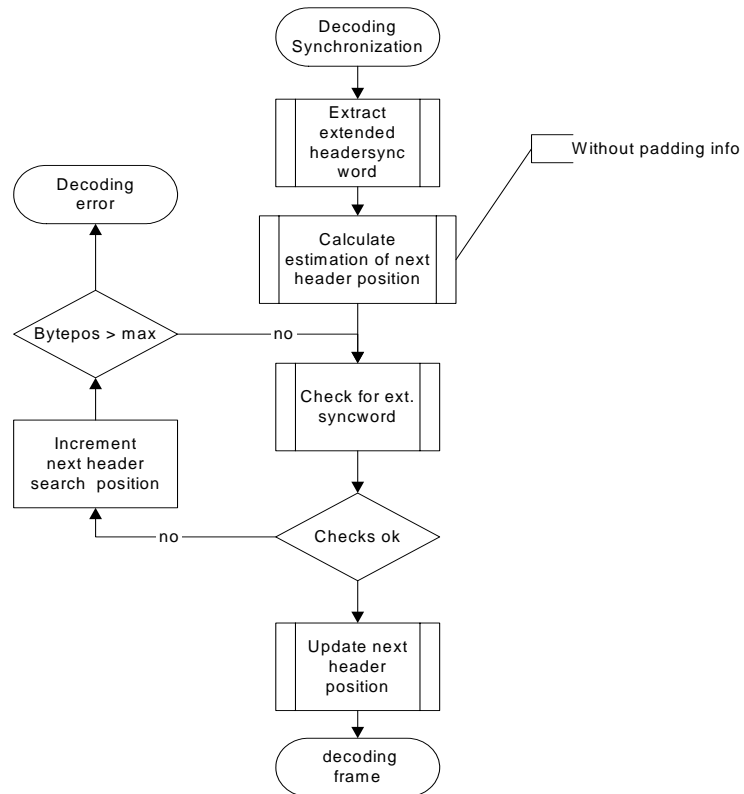


FIGURE 2. Decoding synchronization

3.0 Hardware implementation on players

It has been found, that the implementation of the relatively simple basic algorithm presented here, needs thorough understanding of internal structures and implementation details of the decoder software. A more detailed description of the real modification done to the JAVA program used here is not helpful because this affects implementation details that are very likely completely different for every implementation. Especially the following points have to be considered on an individual base:

3.1 Implementation issues

- How are data bits extracted from the bitstream?
- Is there a possibility of a pushback to the bitstream?
- Which side effects are caused by a modification of the synchronization?
- Is the implementation memory limited?

- Is the decoder software written sufficiently modular to allow local modifications?

What can be derived from the current implementation in JAVA are the following results:

3.2 Hardware constraints

- The increment in processing power is marginal (first benchmarks show less than 1% additional processing power)
- Internal data memory usage is not incremented significantly (<100bytes)
- The effect on the size of the program code is difficult to predict and depends on current software. If the decoder software is written properly, only a local modification needs to be done that causes only no considerable side effects.

3.3 Application to streaming and broadcasting

Streaming and broadcasting applications are push services. In these applications a transmitter defines a bitrate based on its internal time reference.

Real time receivers need to restore the transmitted digital signal. In many applications the clock / time reference information is not transmitted properly. Thus the receiver needs either

- a clock recovery

or

- a time-shift compensation buffer

to overcome the problem of the unsynchronized sample clocks. In both applications a replay algorithm based on the proposed algorithm should work properly because it decodes a bitstream as encoded by the transmitter with/without padding. In the streaming mode the push service assures that a sufficient input buffer reservoir is build up that allows the decoder to take data from the buffer based on its own rate without facing buffer- over/under runs within a reasonable time.