BESA®

BESA V1.0 File Format Specification

Document name:	BESA File Format Specification
Revision number:	007
Revision date:	14. December 2015

Table of Contents

1. Introduction

1.1. File extension

The file extension of files written in BESA File Format is **.besa**.

1.2. Tagged file format

The BESA File Format is being realized as a tagged file format.

A file data written in this format is composed of a set of data elements.

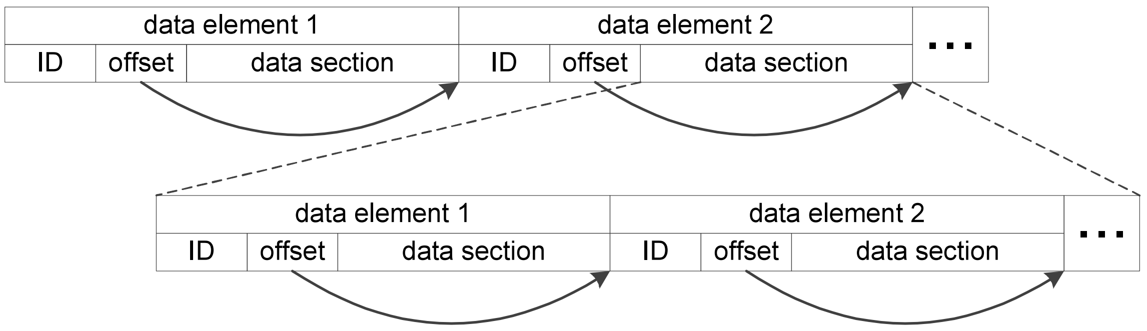
data element		
tag		data section
ID	offset	
4 Bytes: data section ID	4 Bytes: data section size	<data section size> Bytes: data (format depends on data section ID)

Each **data element** is started by a **tag** which describes the following **data section**. Each tag is composed of a **tag ID** and a **tag offset**.

The **tag ID** is a unique identifier specifying the kind of data written in the data section. It is stored using four 8-bit characters (4 Bytes). Please find more details to all available tag IDs in chapter 2.

The **tag offset** specifies the length of the following data section. The offset is stored as one 32-Bit Unsigned Integer (4 Bytes) which specifies the length of the following data section in bytes. The offset can also be used to skip the current data element, e.g. if an unknown tag ID is read.

A **data section** itself may contain a set of (data-section-specific) data elements, e.g. the event block explained in chapter 2.5.



Note: If file writing is still in progress or an error occurred during file writing, the offset value of the corrupt or incomplete section is set to 0xFFFFFFFF. For some special cases the data section may be empty. Then the offset is set to 0.

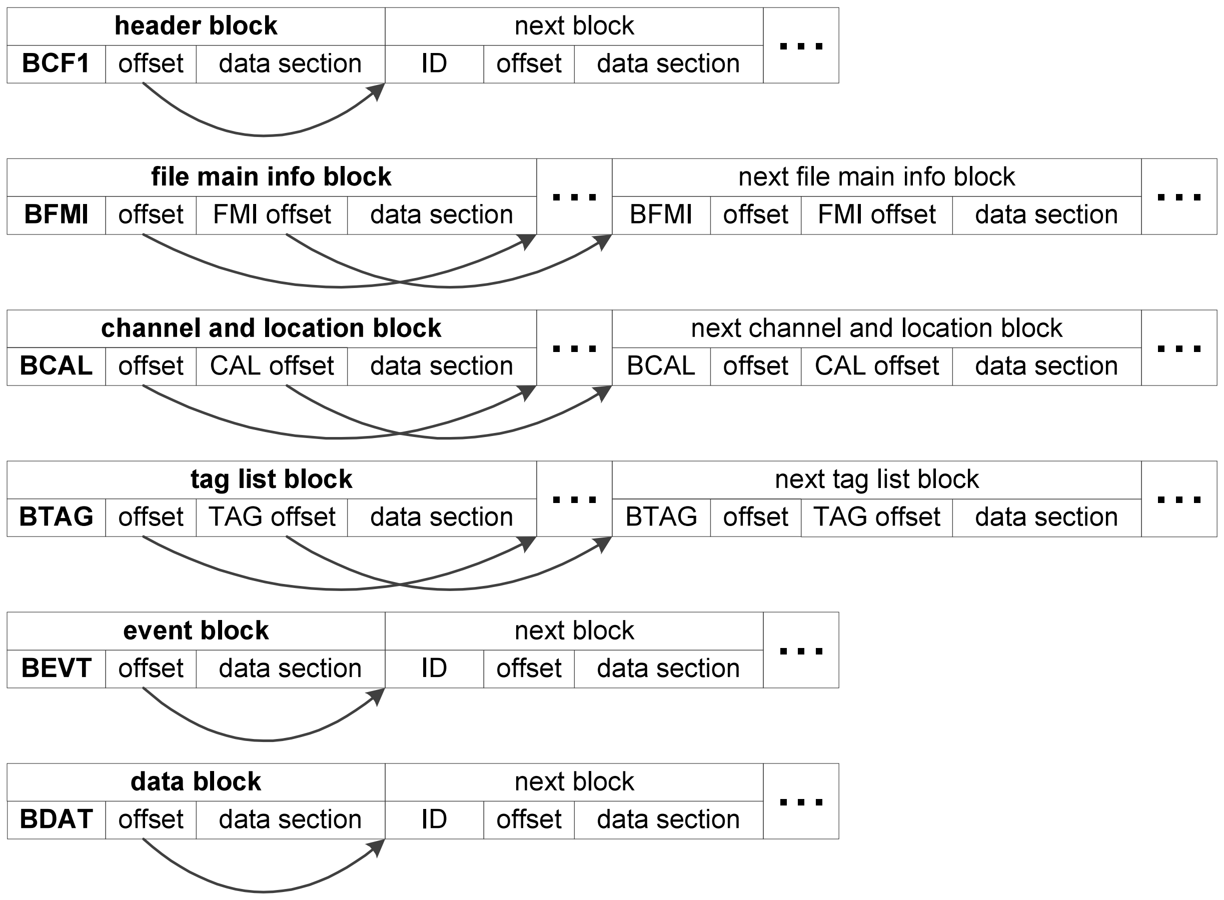
1.3. File compression

The recorded data are stored in data blocks in a compressed or non compressed data format. Please find more details to data blocks in chapter 2.6 and to the compression algorithm in chapter 3.

2. BESA File Format

As introduced in chapter 1.2, the BESA File Format is composed of a set of data elements. Each data element starts with a tag that describes the data format of the following data section. This chapter describes the meaning of all tags that are part of the BESA File Format.

The data elements of the BESA File Format are arranged in blocks. The following blocks are used to group data elements together by content:


header block
BCF1 | offset | data section
next block
ID | offset | data section
...
file main info block
BFMI | offset | FMI offset | data section
next file main info block
BFMI | offset | FMI offset | data section
channel and location block
BCAL | offset | CAL offset | data section
next channel and location block
BCAL | offset | CAL offset | data section
tag list block
BTAG | offset | TAG offset | data section
next tag list block
BTAG | offset | TAG offset | data section
event block
BEVT | offset | data section
data block
BDAT | offset | data section

Exclusive of the header block that is stored once at the beginning of the file, all other blocks may be written multiple times to the file. The reason is that it shall be possible to modify already stored information or to add data to the end of the file without changing already written parts of file. In addition the blocks must not be stored in the order above.

For fast access to multiple blocks of the same type some of the blocks have an additional 64-bit integer offset that points on the next data element of the same block type (e.g. <FMI offset> in figure above). This additional offset is not tagged! It is followed by a data section with several block-specific (tagged) data elements.

The blocks may never exceed a size of 4GB minus one byte (0xFFFFFFFE) since 0xFFFFFFFF is used as flag to indicate an error during the write process. If e.g. data for a data section gets larger than 0xFFFFFFFE, it has to be written to multiple blocks. This way we get a faster data access and the tag offsets can be stored as 32-bit integers.

2.1. Header Block

The header block identified by the tag ID **BCF1** is used for specifying the file header information. It is always written as the first data element in BESA Files.

The data section consists of block specific data elements with the following tag IDs:

BCF1

VERS	File version
OFFM	Offset/Link to the first file main info block
OFTL	Offset/Link to the first tag list block
OFBI	Offset/Link to the first channel and location block

Detailed description of the header block specific data elements written to its data section:

VERS

File version string.

Stored as a series of 2-byte characters.

OFFM

Offset that specifies the position of the first main info block (see chapter 2.2).

Stored as 64-bit integer.

OFTL

Offset that specifies the position of the first tag list block (see chapter 2.4).

Stored as 64-bit integer.

OFCL

Offset that specifies the position of the first channel and location block (see chapter 2.3).

Stored as 64-bit integer.

2.2. File Main Info Block (FMI Block)

The file main info block (FMI block) identified by the tag ID **BFMI** is used for specifying common acquisition information.

The data section starts with a 64-bit offset (**<__int64>**) that specifies the position of the next FMI block since a BESA File may contain multiple FMI blocks.

Note: If multiple FMI blocks contain data of the same type (tag ID), the data of the last block have to be used since the “old” data is not deleted. It has been overwritten by adding a new FMI block with the new data at the end of the file. This way we do not have to rewrite the whole file for the case a larger data content than the currently set shall be written.

The rest of the data section consists of block specific data elements with following tag IDs:

BFMI

<__int64>		Offset/Link to the next file FMI block (UNTAGGED DATA)
SAMT		Total number of samples
SAMP		Number of samples per second
FINN		Name of the institution
FINA		Address of the institution
	ASTR	Street name
	ASTA	State
	ACIT	City
	APOS	Post code
	ACOU	Country
	APHO	Phone number
FENA		Encryption algorithm
FCOA		Compression algorithm
RECD		Recording start date and time
RECE		Recording end date and time
RECO		Recording time offset to GMT
RECS		Recording system
RIBN		Name of the input box
RESW		Name of recording software
RATC		Amplifier time constant
RSEQ		Sequence number
RSID		Session unique identifier
RSNR		Session number
RSTC		Study comment
RSTA		Responsible staff
	SNAM	Name
	SINI	Initials
	SFUN	Function

PNAF	Subject first name
PNAM	Subject middle name
PATN	Subject last name
PNAA	Anonymized subject name
PNAT	Subject title
PATD	Subject date of birth
PDOD	Subject date of death
PAGE	Subject gender
PAWE	Subject weight
PAHE	Subject height
PAMS	Subject marital status
PAAD	Subject address
PALA	Subject language
PAMH	Subject medical history
PATC	Subject comment
PATI	Subject ID
INF1	Additional information 1
INF2	Additional information 2

Detailed description of the FMI block specific data elements written to its data section:

SAMT

Number of samples.

Stored as 64-bit integer value.

SAMP

Number of samples per second (Sampling rate).

Stored as double value.

FINN

Name of the institution.

Stored as a series of 2-byte characters.

FINA

Address of the institution.

For more information for the address sub tags see chapter 2.2.1.

FENA

Name of the encryption algorithm.

Stored as a series of 2-byte characters.

FCOA

Name of the compression algorithm.

Stored as a series of 2-byte characters.

RECD

Start date and time of recording.

The date/time of the first sample in the file. This does not need to be the very first date and time over all samples in the file, e.g. the file contains more than one segment and a segment with an earlier recording time is stored behind a segment with a later recording time in the file.

Stored as a series of 2-byte characters.

Syntax: YYYYMMDDHHMMSSmmmuuu for Year, Month, Day, Hour, Minute, Second, milliseconds and microseconds.

RECE

End date and time of recording.

NOT the date/time of the last sample in the file but specifies the date/time of the last sample of the segment with the latest recording time (which might have been stored in front of segments with earlier recoding times). This value might be written to file for specific file formats, but it is ignored when a file is opened. Instead, it is calculated after opening.

Stored as a series of 2-byte characters.

Syntax: YYYYMMDDHHMMSSmmmuuu for Year, Month, Day, Hour, Minute, Second, milliseconds and microseconds.

RECO

Recording time offset to GMT.

Stored as float value.

RECS

Recording system.

Stored as a series of 2-byte characters.

RIBN

Input box name. (Additional recording system information.)

Stored as a series of 2-byte characters.

RESW

Recording software name.

Stored as a series of 2-byte characters.

RATC

Amplifier time constant.

Stored as float value.

RSEQ

Sequence number. (File format specific value)

Stored as unsigned 32bit-integer value (DWORD).

RSID

Session unique identifier.

Stored as a series of 2-byte characters.

RSNR

Session number.

Stored as 32-bit integer value.

RSTC

Study comment.

Stored as a series of 2-byte characters.

RSTA

Responsible staff.

For more information for the staff sub tags see chapter 2.2.2.

PNAF

Subject first name.

Stored as a series of 2-byte characters.

PNAM

Subject middle name.

Stored as a series of 2-byte characters.

PATN

Subject last name.

Stored as a series of 2-byte characters.

PNAA

Anonymized subject name.

Stored as a series of 2-byte characters.

PNAT

Subject title.

Stored as a series of 2-byte characters.

PATD

Subject birth date.

Stored as a series of 2-byte characters.

Syntax: YYYYMMDD for Year, Month and Day.

PDOD

Subject death date.

Stored as a series of 2-byte characters.

Syntax: YYYYMMDD for Year, Month and Day.

PAGE

Subject gender.

Stored as 2-byte character. One of {‘M’ = men, ‘F’ = female}.

PAWE

Subject weight.

Stored as float value.

PAHE

Subject height.

Stored as float value.

PAMS

Subject marital status.

Stored as 2-byte character. One of {‘S’ = single, ‘M’ = married, ‘W’ = widowed, ‘D’ = divorced}.

PAAD

Subject address.

For more information for the address sub tags see chapter 2.2.1.

PALA

Subject language.

Stored as a series of 2-byte characters.

PAMH

Subject medical history.

Stored as a series of 2-byte characters.

PATC

Subject specific comment.

Stored as a series of 2-byte characters.

PATI

Subject specific ID.

Stored as a series of 2-byte characters.

INF1

Additional information 1.

Stored as a series of 2-byte characters.

INF2

Additional information 2.

Stored as a series of 2-byte characters.

2.2.1. Address Block

The address block is a section of sub tags and has no own identifier.

ASTR	Street name
ASTA	State
ACIT	City
APOS	Post code
ACOU	Country
APHO	Phone number

Detailed description of the address block specific data elements written to its data section:

ASTR

Street name.

Stored as a series of 2-byte characters.

ASTA

State.

Stored as a series of 2-byte characters.

ACIT

City.

Stored as a series of 2-byte characters.

APOS

Post code.

Stored as a series of 2-byte characters.

ACOU

Country.

Stored as a series of 2-byte characters.

APHO

Phone number.

Stored as a series of 2-byte characters.

2.2.2. Staff Block

The staff block is a section of sub tags and has no own identifier.

SNAM	Name
SINI	Initials
SFUN	Function

Detailed description of the staff block specific data elements written to its data section:

SNAM

Name.

Stored as a series of 2-byte characters.

SINI

Initials.

Stored as a series of 2-byte characters.

SFUN

Function.

Stored as a series of 2-byte characters.

2.3. Channel And Location Block (BSA Block)

The channel and location block (BSA block) identified by the tag ID **BCAL** is used for specifying the channel and coordinate related acquisition information.

The data section starts with a 64-bit offset **(<__int64>**) that specifies the position of the next BSA block since a BESA File may contain multiple BSA blocks.

Note: If multiple BSA blocks contain data of the same type (tag ID), the data of the last block have to be used since the “old” data is not deleted. It has been overwritten by adding a new BSA block with the new data at the end of the file. This way we do not have to rewrite the whole file for the case a larger data content than the currently set shall be written.

The rest of the data section consists of block specific data elements with following tag IDs:

BCAL

<__int64>	Offset/Link to the next channel and location block (UNTAGGED DATA)
COMC	CTF components
COMH	Head transformations
CHFL	Channel flag
CHNR	Total number of channels
CHTS	Channel type and states of a channel with the specified index
CHCO	Channel coordinates in mm
CHET	Electrode thickness
CHLA	Channel label of a channel with the specified index
CHSC	Spatial components
CHSI	Spline interpolation smoothing constant
CHLS	Least significant bits of data
CHSF	Sampling frequency
CHCM	Channel comments
CHNU	Channel numbers
CHFI	Filter information
CHCU	Channel units
HCMM	Head center in mm
HRMM	Head radius in mm
FIDC	Fiducial coordinates in mm
HSPN	Total number of head surface points
HSPC	Head surface point coordinates
HSPD	Head surface point labels

Detailed description of the BSA block specific data elements written to its data section:

COMC

BESA CTF component.

For internal use only (BSACtfComponentsArray).

COMH

BESA head transformation

For internal use only (BSAHeadTransformationStruct).

CHFL

Channel flag.

Stored as unsigned 32bit-integer value (DWORD).

This flag specifies additional information concerning the channel coordinates. It is set as a bitwise combination of the following bit flags:

0x0001 (BSA_ELECTRODE_COORDINATES_FROM_LABELS)

> This bit is set if electrode coordinates are not available. In this case, default coordinates are assigned according to electrode labels (see description of the **CHLA** data element). Note that these labels must be standard names according to the 10-20 and 10-10 electrode definitions.
>
> If this bit is set, the electrode coordinate information set in the **CHCO** data element is ignored.

0x0002 (BSA_SUPPRESS_SPHERE_TO_ELLIPSOID_TRANSFORMATION)

> This bit is set if the data originated from a spherical MEG phantom or from simulated EEG or MEG using a spherical head model.
>
> In this case, the sphere-to-ellipsoid transformation is suppressed.
>
> Note that this bit flag must not be combined with the bit flag BSA_ELECTRODE_COORDINATES_FROM_LABELS.

0x0004 (BSA_ELECTRODE_COORDINATES_ON_SPHERE)

> This bit is set if electrode coordinates are predefined based on a sphere, e.g. from simulated data.
>
> In this case, a sphere is not fitted to the electrode cloud, the sphere is not rotated to known electrode coordinates, and the sphere-to-ellipsoid transformation is suppressed.
>
> The coordinates must be defined so that the x axis is oriented to the right ear, the y axis towards the nose, and the z axis upwards. The Cz electrode has the coordinates 0,0,85 for a head radius of 85 mm and head center at 0,0,0.
>
> If the center of the electrode sphere is not 0,0,0 or the sphere radius is not 85 mm, the head center or head radius must be specified using the **HCMM** data element and the **HRMM** data element.
>
> Note that this bit flag must not be combined with the bit flag BSA_ELECTRODE_COORDINATES_FROM_LABELS.
>
> Note that this bit flag must not be combined with the bit flag BSA_SUPPRESS_SPHERE_TO_ELLIPSOID_TRANSFORMATION.

0x0008 (BSA_ADAPT_SPHERICAL_EEG_TO_MEG_COORDS)

> This bit is set if electrode coordinates were not digitized, and digitized head surface points (e.g. fiducials) are available.
>
> This would be the case if MEG was recorded (always digitized), and electrode coordinates were not. If this bit is set, then the spherical electrode coordinates will be scaled to the head center and radius computed from the head surface points. This assumes that the electrode coordinates are provided relative to a sphere, but with zero point through the T9-T10 axis.

0x0010 (BSA_SOURCE_CHANNELS_DERIVED_FROM_MEG):

> This bit is set if source channels were derived from MEG.
>
> For internal use only (TFC module).

CHNR

Total number of channels.

Stored as unsigned 16bit-integer value (WORD).

This value specifies the total number of channels in each condition. This is the sum of all channels of any type, bad and not bad. Note that the EEG common reference should be included if it is a single electrode (and NOT a linked reference).

CHTS

Channel index followed by channel type and state.

Stored as unsigned 16bit-integer value (WORD) followed by unsigned 32bit-integer value (DWORD).

The channel index is written as WORD and identifies the position in the channel type and state array.

The channel type and state is written as DWORD. It is set as a bitwise combination of a flag that specifies the channel type and a flag that specifies the channel state.

This data element is written <number of channels (**CHNR**)> times.

Note: If at least one MEG channel is specified (channel type), the fiducials must be defined in the **FIDC** data element.

The channel type can be set as follows:

0x00000000 (BSA_CHANTYPE_UNDEFINED)

The channel is not set / invalid.

0x00010000 (BSA_CHANTYPE_POLYGRAPHIC)

Polygraphic channel, coordinate information is ignored.

0x00020000 (BSA_CHANTYPE_TRIGGER)

Trigger channel, coordinate information is ignored.

0x00040000 (BSA_CHANTYPE_CORTICALGRID)

Cortical grid electrode, coordinate information is ignored for the moment.

Later we will use the information to display the grid electrode in the mapping view.

0x00080000 (BSA_CHANTYPE_INTRACRANIAL)

Intracranial electrode, coordinate information is ignored for the moment.

Later we will use the information to display the grid electrode in the mapping view.

0x00100000 (BSA_CHANTYPE_SCALPELECTRODE)

Scalp electrode. Coordinate information is required, unless the coordinates are to be defined from their labels (the bit BSA_ELECTRODE_COORDINATES_FROM_LABELS is set in the member **CHFL** data element).

This channel is used for SA fitting (if not marked as 'bad').

0x00200000 (BSA_CHANTYPE_MAGNETOMETER)

MEG sensor, magnetometer. Coordinate information is required.

This channel is used for SA fitting (if not marked as 'bad').

0x00400000 (BSA_CHANTYPE_AXIAL_GRADIOMETER)

MEG sensor, axial gradiometer. Coordinate information is required. This channel is used for SA fitting (if not marked as 'bad').

Note: If measured data is specified in the ppfData member of the "DATA" section, data must be given in femtotesla.

0x01000000 (BSA_CHANTYPE_PLANAR_GRADIOMETER)

MEG sensor, planar gradiometer. Coordinate information is required. This channel is used for SA fitting (if not marked as 'bad').

Note: If measured data is specified in the ppfData member of the "DATA" section, data must be given in femtotesla per cm!

0x00800000 (BSA_CHANTYPE_MEGREFERENCE)

MEG reference channel. Coordinate information is required.

This channel is not used for fitting but it is used to correct the fit. The kind of correction depends on the MEG system type.

Note that an MEG reference channel will be used whether or not it is marked as 'bad'.

0x02000000 (BSA_CHANTYPE_NKC_REFERENCE)

Special reference channel used by NKC, coordinate information is ignored.

The channel state can be set as follows:

0x0001 (BSA_CHANSTATE_BAD)

The channel is marked as 'bad'.

0x0002 (BSA_CHANSTATE_REFERENCE)

The channel is the EEG reference channel.

Note: Only one channel may be defined as the EEG reference!

0x0004 (BSA_CHANSTATE_INTERPOLRECORDED)

Set this state if the channel has to be interpolated by the montage module. If this value is set, the state value BSA_CHANSTATE_BAD is ignored.

If this flag is set when calling other modules, the channel is treated as good.

0x1000 (BSA_CHANSTATE_INVISIBLE)

Set this state if the channel should be ignored by the montage module. The channel is treated in this case as if it did not exist. If this flag is set when calling other modules, it is currently ignored.

CHCO

Channel locations (in mm) and orientations (normalized vectors).

Stored as an array of 9 * <number of channels (**CHNR**)> float values.

The locations and orientations are arranged as follows:

9 location/orientation coordinates of the first channel, followed by

9 location/orientation coordinates of the second channel, etc.

The location/orientation values are set depending on the channel type that was set in **CHTS** section for the associated channel:

0x00100000 (BSA_CHANTYPE_SCALPELECTRODE)

Entries 0 to 2 contain the location (x,y,z).

Entries 3 to 8 are ignored.

0x00200000 (BSA_CHANTYPE_MAGNETOMETER)

Entries 0 to 2 contain the location (x,y,z).

Entries 3 to 5 are ignored.

Entries 6 to 8 contain the orientation (x,y,z).

0x00400000 (BSA_CHANTYPE_AXIAL_GRADIOMETER)

0x01000000 (BSA_CHANTYPE_PLANAR_GRADIOMETER)

Entries 0 to 2 contain the location (x,y,z) of the 1st coil.

Entries 3 to 5 contain the location of the 2nd coil.

Entries 6 to 8 contain the orientation (x,y,z).

0x00800000 (BSA_CHANTYPE_MEGREFERENCE)

Entries 0 to 2 contain the location (x,y,z).

Entries 6 to 8 contain the orientation (x,y,z).

Entries 3 to 5 are checked to decide if the channel is a magnetometer or gradiometer: If all three coordinates are zero the sensor is assumed to be a magnetometer. Otherwise the sensor is assumed to be a gradiometer and the entries 3 to 5 specify the location of the 2nd coil.

CHET

Electrode thickness (in mm).

Stored as float value.

Used to specify the distance from the digitization point (e.g. on top of the electrode) to the scalp surface. Only required if electrode locations were digitized.

CHLA

Channel index followed by channel label.

Stored as unsigned 16bit-integer value (WORD) followed by a series of 2-byte characters.

The channel index is written to WORD and identifies the position to store the data in the channel label array.

The channel label is written to 2-byte characters. Note that the length of the string is given by tag offset (-2 bytes for the channel index).

This data element is written <number of channels (**CHNR**)> times.

CHSC

BESA spatial components

For internal use only (BSASpatialComponentsArray).

CHSI

Spline interpolation smoothing constant.

Stored as float value.

Used for interpolating data for channel positions (e.g. for virtual montages) using spherical spline interpolation.

If this value is set to zero or a negative value the default value is used.

CHLS

Least significant bits (LSB)

Stored as an array <number of channels (**CHNR**)> float values.

Used for reinterpretation of data values stored in 16-bit integer format.

Example:

A least significant bit value of 0.4 [µV] means that the 16-bit integer value 1 is interpreted as 0.4 [µV].

Please note that zero or negative LSB values are not allowed. If a non-positive value is found in the array, a value of "1.f" is used instead.

CHSF

Sampling frequency.

Stored as an array <number of channels (**CHNR**)> double values.

CHCM

Channel comments.

Stored as an array <number of channels (**CHNR**)> series of characters.

CHNU

Physical channel numbers.

Stored as an array <number of channels (**CHNR**)> integer values.

CHFI

Filter information.

Stored as an array <number of channels (**CHNR**)> filter info objects, which contain filter settings.

One filter info object stores following information:

- Structure size, stored as unsigned 32-byte integer value.
- Filter state, stored as unsigned 32-byte integer value and can be set as follows:

  0x00000001 (FLT_LOWCUTOFF_ACTIVE)
  0x00000002 (FLT_HIGHCUTOFF_ACTIVE)
  0x00000004 (FLT_NOTCH_ACTIVE)
  0x00000008 (FLT_BAND_PASS_ACTIVE)
  0x01000000 (FLT_PRE_LOWCUTOFF_ACTIVE)
  0x02000000 (FLT_PRE_SUBTRACT_BASELINE)
  These flags specify which filters are active.

  0x00000000 (FLT_LOWCUTOFF_TYPE_FORWARD)
  0x00000010 (FLT_LOWCUTOFF_TYPE_ZERO_PHASE)
  0x00000020 (FLT_LOWCUTOFF_TYPE_BACKWARD)
  These flags specify the low cutoff filter type.

  0x00000000 (FLT_LOWCUTOFF_SLOPE_06DB)
  0x00000100 (FLT_LOWCUTOFF_SLOPE_12DB)
  0x00000200 (FLT_LOWCUTOFF_SLOPE_24DB)
  0x00000300 (FLT_LOWCUTOFF_SLOPE_48DB)
  These flags specify the slope of the low cutoff filter.

  0x00000000 (FLT_HIGHCUTOFF_TYPE_ZERO_PHASE)
  0x00001000 (FLT_HIGHCUTOFF_TYPE_FORWARD)
  0x00002000 (FLT_HIGHCUTOFF_TYPE_BACKWARD)
  These flags specify the high cutoff filter type.

  0x00000000 (FLT_HIGHCUTOFF_SLOPE_12DB)
  0x00010000 (FLT_HIGHCUTOFF_SLOPE_06DB)
  0x00020000 (FLT_HIGHCUTOFF_SLOPE_24DB)
  0x00030000 (FLT_HIGHCUTOFF_SLOPE_48DB)
  These flags specify the slope of the high cutoff filter.

  0x00100000 (FLT_NOTCH_REMOVE_2ND_HARMONIC)
  0x00200000 (FLT_NOTCH_REMOVE_3RD_HARMONIC)
  If one of these flag is set, the notch filter is applied repeatedly to remove the second and/or third harmonics of the notch frequency.

- Filter frequencies and band widths if the corresponding filters are active, stored as 6 float values:

  Frequency of the low cutoff filter,
  frequency of the high cutoff filter,
  frequency of the notch filter,
  width of the notch filter,
  frequency of the band pass filter,
  width of the band pass filter.

CHCU

Channel units.

Stored as an array <number of channels (**CHNR**)> series of 2-byte characters.

HCMM

Head center (in mm).

Stored as 3 float values (x, y, z).

HRMM

Head radius (in mm).

Stored as float value.

Set this value to zero if no head radius is available. In this case, the head radius is computed using the electrode locations, the fiducials (see **FIDC**), and the additional head surface points (see **HSPN**).

FIDC

Fiducial coordinates (in mm).

Stored as 9 float values:

- Nasion (Nz) position (x, y, z),
- left preauricular point (T9) position (x, y, z) and
- right preauricular point (T10) position (x, y, z).

The fiducials are required, if MEG channels are defined. The fiducials are only recommended if digitized locations (electrodes, etc.) are available.

HSPN

Total number of additional head surface points.

Stored as unsigned 16bit-integer value (WORD).

HSPC

Head surface point index followed by head surface point coordinates (in mm).

Stored as unsigned 16bit-integer value (WORD) followed by 3 floats.

The channel index is written to WORD and identifies the position to store the data in the head surface point coordinates array.

The head surface point coordinates (x, y, z) are written to the float values.

This data element is written <number of head surface points (**HSPN**)> times.

HSPD

Head surface point index followed by head surface point label.

Stored as unsigned 16bit-integer value (WORD) followed by a series of 2-byte charactes.

The channel index is written to WORD and identifies the position to store the data in the head surface point labels array.

The head surface point label is written to 2-byte characters. Note that the length of the string is given by tag offset (-2 bytes for the channel index).

This data element is written <number of head surface points (**HSPN**)> times.

2.4. Tag List Block

The tag list block identified by the tag ID **BTAG** is used for storing file positions of block tags to allow a faster access to the FMI, BSA, event, data and tag list blocks introduced in chapter 2.

The data section starts with a 64-bit offset **(<__int64>**) that specifies the position of the next tag list block since a BESA File may contain multiple tag list blocks.

The rest of the data section consists of block specific data elements with following tag IDs:

BTAG

<__int64>	Offset/Link to the next tag list block (UNTAGGED DATA)
TAGE	Tag list entry

Detailed description of the tag list block specific data elements written to its data section:

TAGE

Tag list entry.

Stored as four 2-byte characters followed by a 64-bit integer and a 32-bit integer.

The four 2-byte characters represent the block tag ID (**BFMI**, **BCAL**, **BTAG**, **BEVT**, **BDAT**).

The 64-bit integer represents the file position of the block with the ID above.

The 32-bit integer represents the number of samples stored in a data block (**BDAT**). Note that this value is only set for data blocks.

2.5. Event Block

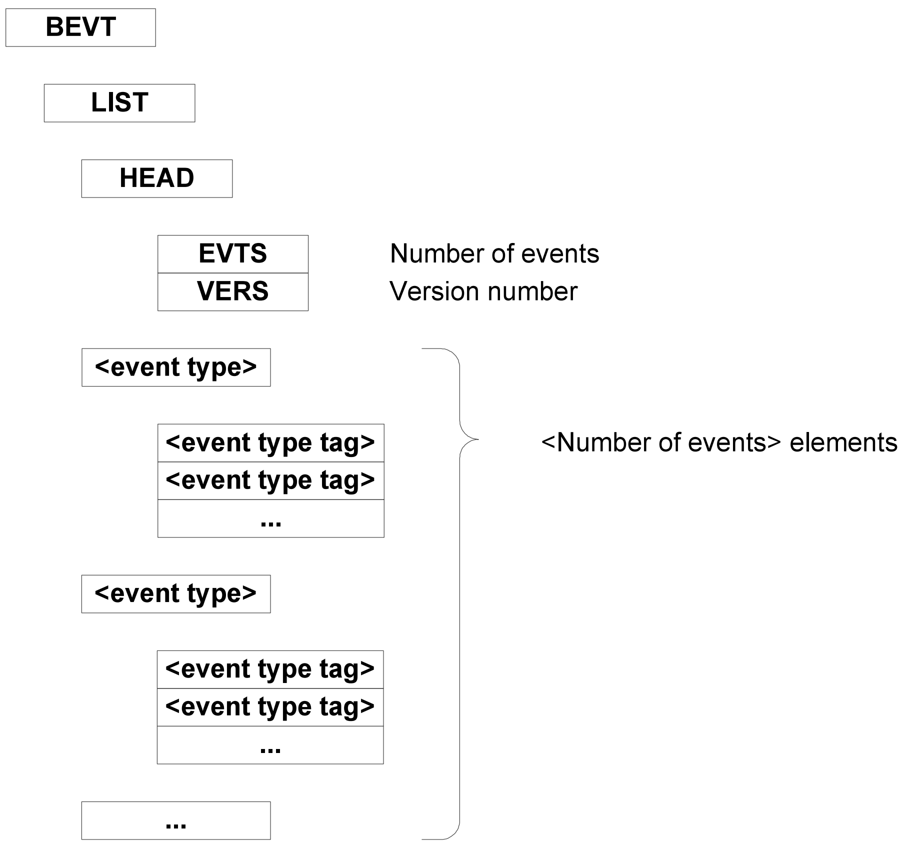
The event block identified by the tag ID **BEVT** is used for storing event information.

A BESA File may contain multiple event blocks.

Note: If multiple event blocks contain data of the same event, the data of the last block have to be used since the "old" data is not deleted. It has been overwritten by adding a new event block with the new data at the end of the file. This way we do not have to rewrite the whole file for the case a larger data content than the currently set shall be written.

The tag ID of the event block is **BEVT**. This data section contains only one **LIST** data element.

The data section of a **LIST** data element is composed of a header (**HEAD**) data element and multiple **<event type>** data elements:



Detailed description of the event block specific data elements written to its data section:

EVTS

Number of events.

Stored as 32-bit integer value.

Specifies the number of <event type> sections part of the **LIST** data section.

VERS

Version number.

Stored as 32-bit integer value.

Version number is currently set to 1.

The **<event type>** tag may be one of the following tags:

BASE	Base event
COMM	Comment event
MARK	Marker event
GENE	Generic event
SEGM	Segment start event
ASGM	Average segment start event
MPS	Multiple pattern search event
MPSC	Classified multiple pattern search event
PATT	Pattern event
TRIG	Trigger event
PAIR	Paired event
ARTI	Artifact event
EPOC	Epoch event
IMP	Impedance event

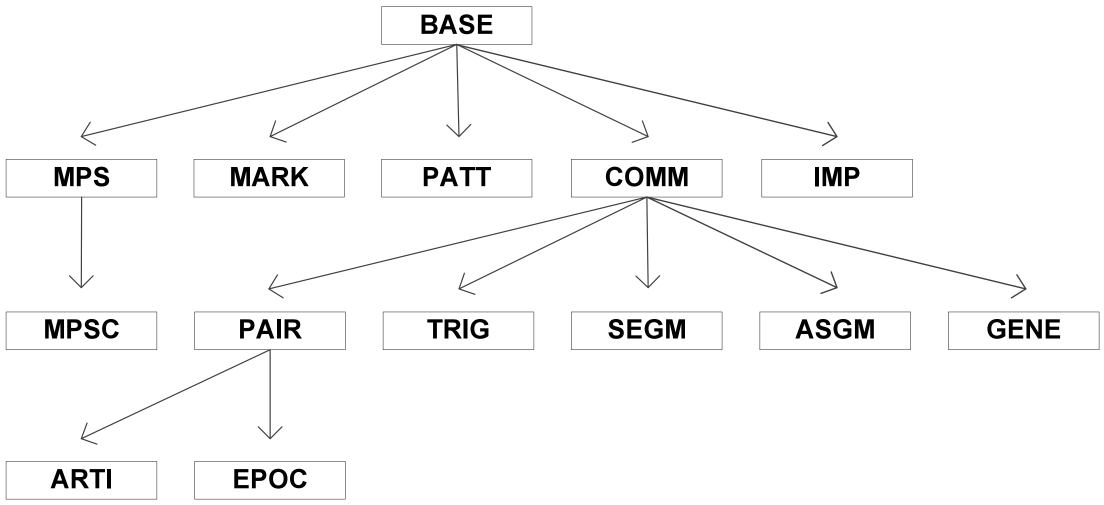
The event data section consists of data elements with event type specific tag IDs (specified as **<event type tag>** in the figure on the previous page). These IDs and the corresponding data are described in detail in the following subsections.

Note:

All events are derived from the base event (**BASE**), which stores the basic information of the event.

Furthermore there are other derivations for events which store their information of several event types. E.g. the artifact (**ARTI**) event is derived from the paired event (**PAIR**), to store the start event and the end event information. And each of the 'partners' contain an annotation stored in the comment event (**COMM**).

Here is an overview over the event dependencies:



2.5.1. Base Event

The base event is identified by the tag ID **BASE.**

The data section consists of block specific data elements with following tag IDs:

BASE

SAMP	(Zero-based) file sample index
TIME	Event time
SIDX	(Zero-based) segment index
CODE	(Zero-based) event code
EVID	(Zero-based) internal BESA event ID
STAT	Event state

Detailed description of the base event specific data elements written to its data section:

SAMP

(Zero-based) file sample index.

Stored as 64-bit integer value.

TIME

Event time.

Stored as a time information sequence:

WORD	Year
WORD	Month
WORD	DayOfWeek
WORD	Day
WORD	Hour
WORD	Minute
WORD	Second
WORD	Milliseconds
double	Microseconds
DWORD	State flag (for internal use).

SIDX

(Zero-based) segment index.

Stored as 32-bit integer value.

CODE

(Zero-based) event code.

Stored as 32-bit integer value.

This value is used by events of type Pattern (**PATT**) and Trigger (**TRIG**) to store the pattern number and the trigger code. Note that the number/code minus 1 is stored. Additionally, events of type Artifact (**ARTI**) and Epoch (**EPOC**) use the code value internally due to historical reasons. Other event types may use the code value to store additional information.

EVID

(Zero-based) internal BESA event ID.

Stored as 32-bit integer value.

STAT

Event state.

Stored as unsigned 32-bit integer value (DWORD).

The event state can be set as follows:

0x00000010 (EVT_STATE_MARKED1),

0x00000020 (EVT_STATE_MARKED2),

0x00000040 (EVT_STATE_MARKED3),

0x00000080 (EVT_STATE_MARKED4):

Mark and group specific events.

0x01000000 (EVT_STATE_DELETED):

The "Deleted" state specifies, that this event has been deleted or this event deletes an identical second event in the event list.

2.5.2. Comment Event

The comment event is identified by the tag ID **COMM**.

The data section consists of block specific data elements with following tag IDs:

COMM

...	Data elements of **BASE** section See description of base event section
TEXT	Comment text

Detailed description of the comment event specific data elements written to its data section:

TEXT

Comment text.

Stored as a series of 2-byte characters.

2.5.3. Marker Event

The marker event is identified by the tag ID **MARK**.

The data section consists of block specific data elements with following tag IDs:

MARK

...	Data elements of **BASE** section See description of base event section

2.5.4. Segment Start Event

The marker event is identified by the tag ID **SEGM**.

The data section consists of block specific data elements with following tag IDs:

SEGM

...	Data elements of **COMM** section See description of comment event section
SBEG	Segment start time
DAYT	Day time of segment start in microseconds
INTE	Sampling interval in microseconds

Detailed description of the segment start event specific data elements written to its data section:

SBEG

Segment start time.

Stored as a time information sequence:

WORD	Year
WORD	Month
WORD	DayOfWeek
WORD	Day
WORD	Hour
WORD	Minute
WORD	Second
WORD	Milliseconds
double	Microseconds
DWORD	State flag (for internal use).

DAYT

Day time of segment start in microseconds from the recording start or date of the last segment start (at 00:00:00) to the date and time of this segment start.

Stored as double value.

INTE

Sampling interval in microseconds, means time between samples (the inverse of the sampling rate)

Stored as double value.

2.5.5. Average Segment Start Event

The marker event is identified by the tag ID **ASGM**.

The data section consists of block specific data elements with following tag IDs:

ASGM

...	Data elements of **COMM** section See description of comment event section
STIM	Prestimulus interval in microseconds
AVRS	Number of averages

Detailed description of the average segment start event specific data elements written to its data section:

STIM

Prestimulus baseline interval in microseconds.

Stored as double value.

AVRS

Number of averages.

Stored as 32-bit integer value.

2.5.6. Multiple Pattern Search Event

The multiple pattern search event is identified by the tag ID **MPS**. It is used by BESA internally.

2.5.7. Classified Multiple Pattern Search Event

The classified multiple pattern search event is identified by the tag ID **MPSC**. It is used by BESA internally.

2.5.8. Pattern Event

The pattern event is identified by the tag ID **PATT**.

The data section consists of block specific data elements with following tag IDs:

PATT

...	Data elements of **BASE** section See description of base event section

2.5.9. Trigger Event

The trigger event is identified by the tag ID **TRIG**.

The data section consists of block specific data elements with following tag IDs:

TRIG

...	Data elements of **COMM** section See description of comment event section
CODE	Reaction code
TIME	Reaction time

Detailed description of the trigger event specific data elements written to its data section:

CODE

Event reaction code.

Stored as 32-bit integer value.

TIME

Event reaction time in seconds.

Stored as float value.

2.5.10. Paired Event

The paired event is identified by the tag ID **PAIR**.

The data section consists of block specific data elements with following tag IDs:

PAIR

...	Data elements of **COMM** section See description of comment event section
PART	Event information of the partner event.

Detailed description of the paired event specific data elements written to its data section:

PART

Event information of the partner event.

The data section is used to write the partner event information as a data element (starting with **<event type>** tag ID of partner event).

2.5.11. Artifact Event

The artifact event is identified by the tag ID **ARTI**.

The data section consists of block specific data elements with following tag IDs:

ARTI

...	Data elements of **PAIR** section See description of paired event section

2.5.12. Epoch Event

The epoch event is identified by the tag ID **EPOC**.

The data section consists of block specific data elements with following tag IDs:

EPOC

...	Data elements of **PAIR** section See description of paired event section

2.5.13. Impedance Event

The impendance event is identified by the tag ID **IMP**.

The data section consists of block specific data elements with following tag IDs:

IMP

...	Data elements of **BASE** section See description of base event section
FORM	Format
NR	Number of channels
TYPE	Channel type
LABL	Channel label
VAL	Value

Detailed description of the impedance event specific data elements written to its data section:

FORM

Indicates the format used to store impedance values in **VAL**.

Stored as 32-bit integer value.

Set to 0 if the impedance status (valid/invalid) is stored.
Set to 1 if impedance values (in kOhm) are stored.

NR

Number of channels for which impedance information is stored. (Number of elements stored in **TYPE**, **LABL** and **VAL**)

Stored as unsigned 32bit-integer value (DWORD).

TYPE

Array with **NR** channel types.

Stored as unsigned 32bit-integer values (DWORDs).

The flags used for channel type description are the same as used for the **CHTS** data elements (specified in chapter 2.3).

Note: Channel type and channel label are used for identification of channel for which impedance information is set. (Compare to data elements **CHTS** and **CHLA**, specified in chapter 2.3).

LABL

Array with **NR** channel labels.

Stored as a series of 2-byte characters.

Note: Channel type and channel label are used for identification of channel for which impedance information is set. (Compare to data elements **CHTS** and **CHLA** as specified in chapter 2.3).

VAL

Array with **NR** impedance values.

Stored as float values

Depending on format set in **FORM** either an impedance STATUS (ok/not ok) or an impedance VALUE (in kOhm) is stored. A value of -1 means that the impedance is not set or invalid

2.5.14. Generic Event

The generic event type identified by the tag ID **GENE** was defined to be able to store events that do not belong to any of the other specified event types.

The data section consists of block specific data elements with following tag IDs:

GENE

...	Data elements of **COMM** section See description of comment event section

2.6. Data Block

The data block identified by the tag ID **BDAT** is used for storing data.

The data block data section consists of data elements with block-specific tag IDs and offsets that specify the position of the next data element.

The data section consists of block specific data elements with following tag IDs:

BDAT

DATT	Data type of written data (int16 or float, compressed or not)
DATS	Number of samples in data block
DATA	Data

Detailed description of the data block specific data elements written to its data section:

DATT

Specifies the data type of the written data and the information whether data are written compressed or uncompressed.

Stored as unsigned 32bit-integer value (DWORD).

It is set as a bitwise combination of flags specifying the data type and flags specifying the compression.

The data type can be set as follows:

0x0000UL (BESA_DATA_TYPE_FLOAT)

Stored as float.

0x0001UL (BESA_DATA_TYPE_INT16)

Stored as 16-bit integer.

Compare to description of the least significant bit (as specified in the **CHLS** data section, see chapter 2.3).

The compression can be set as follows:

0x0000UL (BESA_DATA_TYPE_UNCOMPRESSED)

Do not use compression.

0x0010UL (BESA_DATA_TYPE_COMPRESSED)

Use data compression as described in the next chapter.

DATS

Specifies the number of samples written to block of data.

Stored as 32-bit-integer value.

DATA

Block of data. Consists of <number of channels> * <number of samples> data values.

First all samples of channel 1 are stored, then all samples of channel 2 are stored etc.

The number of channels is specified in the **CHNR** data section (see chapter 2.3) and the number of samples is specified in the **DATS** data element introduced in this chapter.

The format of data depends on data type and compression as set in data element **DATT**:

- Uncompressed float data are stored as data block of <number of channels> * <number of samples> float values.
- Uncompressed 16-bit integer data are stored as data block of <number of channels> * <number of samples> 16-bit integer values.
- Compressed data (float or 16-bit integer) are stored as a data block of the BYTEs that are output of the compression algorithm as described in chapter 3.

3. File Compression format

The file compression performed by BESA works in combination with the ZLib library freely available at http://zlib.net/.

In short the BESA file format encoding consists of two differentiations, a pre-compression and finally the ZLib compression for each channel. The decoding is achieved by simply applying the reverse operation.

The BESA file compression format is **lossless** provided the data to be compressed is of 16-bit integer or 32-bit integer type. If the input data is float or double there is a loss when truncating to integers.

We will always assume to be working in a little endian byte system. I.e. the least significant byte of any multi-byte integer will be the first in memory and the most significant byte the last.

In the following we detail the encoding and decoding steps further.

3.1. Encoding

Each channel is encoded separately and stored consecutively in the output byte array `output`.

The Figure 1 shows an overview over the encoding process. Please refer to the text for more details regarding the differentiation, pre-compression and compression steps.

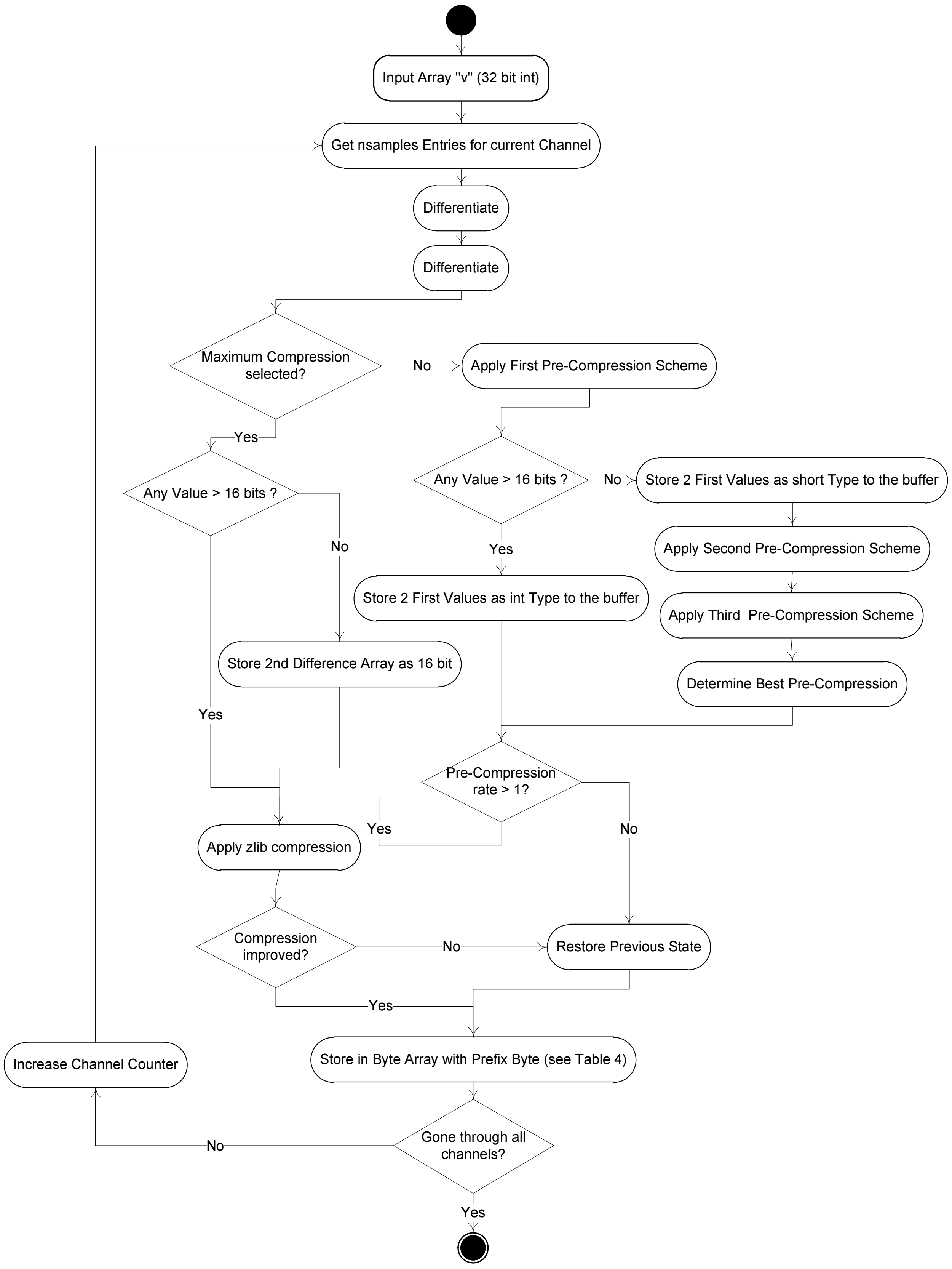

Input Array "v" (32 bit int)
Get nsamples Entries for current Channel
Differentiate
Differentiate
Maximum Compression selected?
No
Apply First Pre-Compression Scheme
Yes
Any Value > 16 bits ?
Any Value > 16 bits ?
No
Store 2 First Values as short Type to the buffer
No
Yes
Apply Second Pre-Compression Scheme
Store 2 First Values as int Type to the buffer
Apply Third Pre-Compression Scheme
Store 2nd Difference Array as 16 bit
Determine Best Pre-Compression
Yes
Pre-Compression rate > 1?
Yes
No
Apply zlib compression
Compression improved?
No
Restore Previous State
Yes
Increase Channel Counter
Store in Byte Array with Prefix Byte (see Table 4)
No
Gone through all channels?
Yes

Figure 1: Summary of the encoding procedure. Please refer to the text for more details regarding the differentiation, pre-compression and compression steps.

3.1.1. Differentiation

Given the vector `v`, for the current channel, with `nsamples` 32-bit integer entries (hereafter simply `int`) we compute the difference vector `d` with `nsamples` entries of type `int` according to

```
d[i] = v[i] - v[i-1]
```

where `i=1,...,nsamples-1` is the index. Furthermore we choose `d[0] = v[0]`.

The same procedure is applied to obtain the double difference vector `dd` of length `nsamples` and type `int`.

```
dd[i] = d[i] - d[i-1] = v[i] - v[i-1] - v[i-1] + v[i-2]
```

where `i=2,...,nsamples-1` is the index and analogous to above we set `dd[1] = d[1]` and also set `dd[0] = d[0]`.

3.1.2. Pre-Compression

In this step the encoder determines if it is possible to store the 32-bit difference vector `dd` entries as 8-bit or 16-bit numbers and furthermore determines if up to 4 subsequent entries in `dd` can be fitted in an 8-bit number. The result is stored in an unsigned char array `buffer`.

3.1.2.1. The First Two Entries

Since the two first values of the second differences array `dd` (see chapter 3.1.1) do not undergo the differentiation process it is likely, assuming neighboring data points are correlated, that their values will be larger than the rest of the array. They will be stored either as `short int` type (16-bit) values, provided they fit, or as `int` type values. The two options are differentiated by the prefix byte, which we describe later on (see section 3.1.2.5). Note also that the pre-compression schemes skip the first two entries and work only from the third `dd` entry onwards.

3.1.2.2. Pre-Compression Schemes

The `dd` input is tested for compressibility using three different compression schemes. Each of these schemes basically checks if **m** subsequent values of the vector `dd` can be fitted in a single byte instead of the **m** * 4 bytes (of each of the 32-bit input integers) with `m` ranging from 2 to 4. The sequences of values which satisfy that condition are encoded into 1 byte in the output array `buffer` as the index to a table which uniquely determines the sequence.

Note that **only one scheme,** the best performing one, is used for the pre-compression. In case none of them achieves a compression rate (i.e. output bytes divided by input bytes) < 1 for some given input, no pre-compression is employed.

First Compression Scheme

We will illustrate the idea in more detail in the following example which corresponds to the first out of three compression schemes used:

- We check if the subsequent values `dd[q], dd[q+1]` are both contained in the interval `[-7, 7]`. Note that `q` starts with the third entry of `dd`.
- If so they can be encoded as an index of the table shown in Table 1.
- The index for any entry of the table is computed according to the convention

  ```
  index = j + i * Δ
  ```

  where Δ is the width of the table (15 in our case), `j` stands for the column and `i` is the row. Please note that `i,j` belong to the interval [0,14]. Their value is obtained by adding (Δ - 1)/2 to the respective dd[q], dd[q+1] to be encoded. In the example in Table 1, where `dd[q] = 6` and `dd[q+1] = -4`,

  ```
  i = 6 + (15 - 1)/2 = 13 and j = -4 + 7 = 3.
  ```

The obtained index is than stored in the output byte array buffer.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	(-7,-7)	(-7,-6)	(-7,-5)	(-7,-4)	(-7,-3)	(-7,-2)	(-7,-1)	(-7,0)	(-7,1)	(-7,2)	(-7,3)	(-7,4)	(-7,5)	(-7,6)	(-7,7)
1	(-6,-7)	(-6,-6)	(-6,-5)	(-6,-4)	(-6,-3)	(-6,-2)	(-6,-1)	(-6,0)	(-6,1)	(-6,2)	(-6,3)	(-6,4)	(-6,5)	(-6,6)	(-6,7)
2	(-5,-7)	(-5,-6)	(-5,-5)	(-5,-4)	(-5,-3)	(-5,-2)	(-5,-1)	(-5,0)	(-5,1)	(-5,2)	(-5,3)	(-5,4)	(-5,5)	(-5,6)	(-5,7)
3	(-4,-7)	(-4,-6)	(-4,-5)	(-4,-4)	(-4,-3)	(-4,-2)	(-4,-1)	(-4,0)	(-4,1)	(-4,2)	(-4,3)	(-4,4)	(-4,5)	(-4,6)	(-4,7)
4	(-3,-7)	(-3,-6)	(-3,-5)	(-3,-4)	(-3,-3)	(-3,-2)	(-3,-1)	(-3,0)	(-3,1)	(-3,2)	(-3,3)	(-3,4)	(-3,5)	(-3,6)	(-3,7)
5	(-2,-7)	(-2,-6)	(-2,-5)	(-2,-4)	(-2,-3)	(-2,-2)	(-2,-1)	(-2,0)	(-2,1)	(-2,2)	(-2,3)	(-2,4)	(-2,5)	(-2,6)	(-2,7)
6	(-1,-7)	(-1,-6)	(-1,-5)	(-1,-4)	(-1,-3)	(-1,-2)	(-1,-1)	(-1,0)	(-1,1)	(-1,2)	(-1,3)	(-1,4)	(-1,5)	(-1,6)	(-1,7)
7	(0,-7)	(0,-6)	(0,-5)	(0,-4)	(0,-3)	(0,-2)	(0,-1)	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
8	(1,-7)	(1,-6)	(1,-5)	(1,-4)	(1,-3)	(1,-2)	(1,-1)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
9	(2,-7)	(2,-6)	(2,-5)	(2,-4)	(2,-3)	(2,-2)	(2,-1)	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
10	(3,-7)	(3,-6)	(3,-5)	(3,-4)	(3,-3)	(3,-2)	(3,-1)	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
11	(4,-7)	(4,-6)	(4,-5)	(4,-4)	(4,-3)	(4,-2)	(4,-1)	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
12	(5,-7)	(5,-6)	(5,-5)	(5,-4)	(5,-3)	(5,-2)	(5,-1)	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
13	(6,-7)	(6,-6)	(6,-5)	(6,-4)	(6,-3)	(6,-2)	(6,-1)	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
14	(7,-7)	(7,-6)	(7,-5)	(7,-4)	(7,-3)	(7,-2)	(7,-1)	(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Table 1: The table has 225 entries (the index thus ranges from 0 to 224). The orange marked cell represents the entry for `dd[q] = 6, dd[q+1] = -4` which corresponds to the index `index = 198`.

Second Compression Scheme

Other than the above mentioned the encoding can also be done with a second scheme.

a. First the subsequent values of `dd[q], dd[q+1], dd[q+2]` are considered (thus instead of a 2 dimensional array for the indexing one obtains a 3 dimensional array). The values are probed to see whether all fall in the interval `[-2, 2]`. The indexing in this case is just an extension of the aforementioned:

   `index = k + (j + i * Δ) * Δ .`

   Here `Δ = 5` and `i,j,k` belong to the interval `[0,4]`. The index thus ranges from 0 to 124.

b. In case the condition (a). cannot be achieved a second check is done to determine if the subsequent values `dd[q], dd[q+1]` fall in the interval `[-5, 5]`. The index being computed for the 2 dimensional case now added to an offset of 125 to avoid interference with the item (a) above.

   `index = j + i * Δ + 125`

   The width of the table is Δ = 11 which results in an index for (b) ranging from 125 to 245, and an overall (considering a and b) index ranging from 0 to 245.

Third Compression Scheme

The final compression scheme, analogously to the second scheme tries two possibilities out.

a. We consider the four subsequent values `dd[q], dd[q+1], dd[q+2], dd[q+3]` (thus the indexing happens over a 4 dimensional array). We check if the values are all in the interval `[-1, 1]`. Thus the indexing is of the form

   `index = l + (k + (j + i * Δ) * Δ) * Δ.`

   The width `Δ = 3` and `i,j,k,l` belong to the interval `[0, 2]`. The index ranges from 0 to 80.

b. If (a) is not achieved a second test will check if `dd[q], dd[q+1]` both fall in the interval `[-6, 6]`. Thus we have `Δ = 13` and `i,j` belong to the interval `[0,12]`. The index carries an offset of 81 to avoid conflicts with (a)

```
index = j + i * Δ + 81 .
```

Thus (b) ranges from 81 to 249 and overall the index ranges from 0 to 249, which is then stored in the output byte buffer.

A summary of the pre-compression schemes is given in Table 2.

Pre-Compression Scheme	#Sequential Entries to be Compressed	Value Range	Index Range	Δ	Table Indexing
First	2	[-7,7]	[0,224]	15	`j + i * Δ`
Second a	3	[-2,2]	[0,124]	5	`k + (j + i * Δ) * Δ`
Second b	2	[-5,5]	[125,245]	11	`j + i * Δ + 125`
Third a	4	[-1,1]	[0,80]	3	`l + (k + (j + i * Δ) * Δ) * Δ`
Third b	2	[-6,6]	[81,249]	13	`j + i * Δ + 81`

Table 2: Summary of the pre-compression schemes. Note that the indexes i,j,k,l run from [0, Δ - 1]. Add (Δ - 1) / 2 to the respective sequential entry to obtain the corresponding i,j,k,l index.

3.1.2.3. Handling non encodable values

In case values arise which do not fall within the limitations described in the preceding section 3.1.2.2 the values are stored, provided they fit, in byte (8-bit), `short int` (16-bit) or `int` (32-bit) format in the output byte `buffer`. They are respectively preceded by one byte announcing the number `x` of byte, `short int` or `int` types following one after the other. The maximum value of `x` depends on what compression scheme (and hence which values in the range 0 to 255 are still free) is being used.

Table 3 shows the respective values for each scheme.

The interval for `x` is supposed to be read in the reverse direction. I.e. if the value of the announcing byte corresponds to the upper bound in the table, it means that only one entry of a respective type (byte, `short int`, or `int`) is being announced. While if the value corresponds to the lower bound the maximum number sequential inputs is being announced.

Should `x` be too small to accommodate the number of sequential bytes, `short int`s or `int`s, one writes the remaining values by simply issuing a new announcing byte after the first maximum sequential inputs have been stored to the output buffer.

	maximum sequential inputs x			**interval for x**		
Pre-Compression Scheme	**byte (8bits)**	**short int (16bits)**	**int (32bits)**	**byte (8bits)**	**short int (16bits)**	**int (32bits)**
First	7	6	6	[248,254]	[242,247]	[236,241]
Second	5	4	unused	[250,254]	[246,249]	unused
Third	3	2	unused	[252,254]	[250,251]	unused

Table 3: Maximum amount of sequential byte, short int or int type values to be announced by each signaling byte as a function of pre-compression schemes. The index interval into which the information is written is also shown.

Moreover, as can be seen from

Table 3 the schemes **second** and **third** do not consider the `int` type values. Indeed the schemes shall always be tried out sequentially, and if on the first scheme `int` type data is detected the two other schemes are not to be used.

3.1.2.4. Pre-Compression Criterion

The length of the second difference array in bytes is `nsamples` * 4 (as it is an array of 32-bit integers). The pre-compressed buffer shall have a maximum size in bytes corresponding `nsamples` * 2. If however values are detected which cannot fit in 2 byte numbers the pre-compressed buffer shall have the maximum length corresponding to the length of the second difference array.

Should the pre-compressed buffer exceed the maximum size at any time the pre-compression is not convenient and should therefore not be used.

3.1.2.5. Storing pre-compressed data

The result of the most successful pre-compression (i.e. the one which generates the shortest output byte array) is stored with a prefixed byte specifying what pre-compression scheme was used and whether any uncompressed data is contained in the pre-compressed array.

Table 4 shows the values the prefix byte should have depending on the pre-compression and also the Zlib compression which is described in the next section.

Prefix Byte value	pre-compression mode / compression
0	No Compression
3	First Scheme
4	Second Scheme
5	Third Scheme
6	No Compression + first 2 entries `int`
7	First Scheme + first 2 entries `int`
8	No Compression (all `int`)
9	zlib on Second Difference `dd`
13	zlib on First Scheme
14	zlib on Second Scheme
15	zlib on Third Scheme
17	zlib on First Scheme + first 2 entries `int`
18	zlib on Second Scheme + first 2 entries `int`
19	zlib on Third Scheme + first 2 entries `int`
29	zlib on Second Difference `dd` (all `int`)

Table 4: Values of the prefix byte as a function of the pre-compression and compression mode. If not otherwise specified the array onto which the compression is applied is of type short int. The label "first 2 entries int" refers to the fact that the first two entries of the dd array are of type int (i.e. cannot be stored in short int). The label "all int" means the dd array is of type int.

3.1.3. ZLib Compression

Following the pre-compression step we apply the Zlib `compress` function to the `buffer` (i.e. without the prefix byte).

The Zlib library and its documentation can be downloaded at http://zlib.net/. The current version of Zlib used for the BESA file format is version 1.2.5, April 19th, 2010.

The resulting compressed byte array overwrites the aforementioned `buffer` array. There is one particularity, the first 4 bytes give the length in bytes of the following zlib compressed array.

If the option "maximum compression" is specified in the BESA software the second difference array is going to be compressed directly by zlib, without the pre-compression step.

In that case the second difference array `dd` is treated analogously to the pre-compressed array `buffer`. It is also checked if the difference array is completely containable in 16-bit numbers, in which case the compression is performed on a 16-bit copy thereof and the prefix byte is set to code 9. Otherwise the zlib compression is directly applied to the 32-bit array in which case the prefix byte code is set to 29.

3.1.4. Final Output

Once the compression for one channel is complete the resulting byte array, including the prefix byte is stored in the `output` byte array. The next channel is appended immediately after that. An example on how the `output` array should look like can be seen in Figure 2.

3.2. Decoding

The decoding of the BESA compression format is achieved by a reverse application of the steps applied in the previous section 3.1. The resulting array should have `nsamples` times the number of channels entries.

The size of the encoded block has to be obtained from the offset of the data block (see section 1.2 and 2.6). Each channel is encoded separately. Thus for each channel we have to perform the decompression up to the point that `nsamples` points have been recovered. After which the offset `chan_offset` from the start of the data block for the beginning of the next channel is known (simply one byte more than the last byte necessary to read out all `nsamples` points).

The encoding steps are stored in the first byte of the compressed string. Please refer to

Table 4 for the possible codes.

In case the buffer is Zlib compressed the first 32 bits after the prefix byte give the length in bytes of the Zlib compressed buffer, which follows immediately after. For convenience we present an example on how the compressed array should look like (see Figure 2).

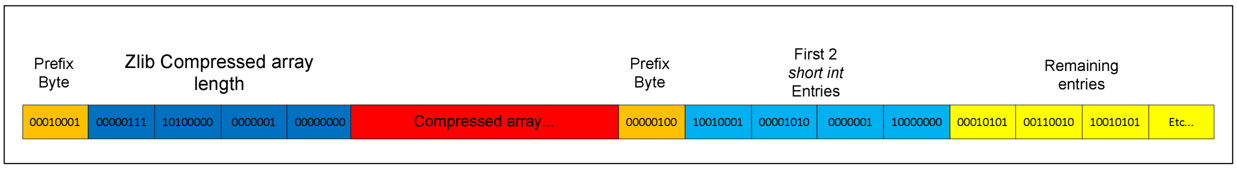


Figure 2 Example for the structure in memory of the BESA file compression format. Here the code in the first Prefix Byte, representing the first stored channel is 17. Thus according to Table 4 the following bytes will be zlib compressed, and once the zlib compression is undone the first two entries of the first scheme pre-compressed array shall be integers. The dark blue piece of the memory indicates the length in bytes of the red zlib compressed buffer. It is a 32-bit number written in little endian byte convention and corresponds to 106503. The second Prefix Byte marks the beginning of the next channel, which has code 4, indicating it was compressed according to the second scheme. The light blue part shows the first two *short int* entries of the second difference array (2705, -32767), while the yellow part indicates the remaining bytes of the second difference array encoded according to the second scheme (the decoded values following 3.1.2.2 being: -2, 2, 1, 0, -2, -2, -3, -3, etc...).

3.2.1. ZLib Decompression

The inverse to the the zlib `compress` function (see section 3.1.3) is the `uncompress` function. Given the length of the zlib compressed array (the number in blue in Figure 2) `uncompress` is applied to the compressed array (red marked section in Figure 2).

Please refer to the zlib documentation for any further details.

3.2.2. De-Pre-Compression

Once the zlib decompression is done (if even required), we have to apply the inverse pre-compression scheme given in the prefix byte.

Each byte of the buffer is probed to see if it carries the code for a sequence of second difference array entries (e.g.

Table 1) or whether the byte announces a sequence of byte, `short int` or `int` type numbers (please note that they will be stored according to the little endian byte convention. I.e. least significant byte first).

The buffer shall then be translated to the original second differences array.

3.2.3. Integration

The recovered second differences array dd can be transformed back to the original input array v according to the sequential application of the following operations

```
d[i] = dd[i] + d[i-1],
v[i] = d[i] + v[i-1].
```

Recalling that `dd[0]=d[0]=v[0], dd[1]=d[1]`. Once the array `v` has been recovered decoding has been completed.

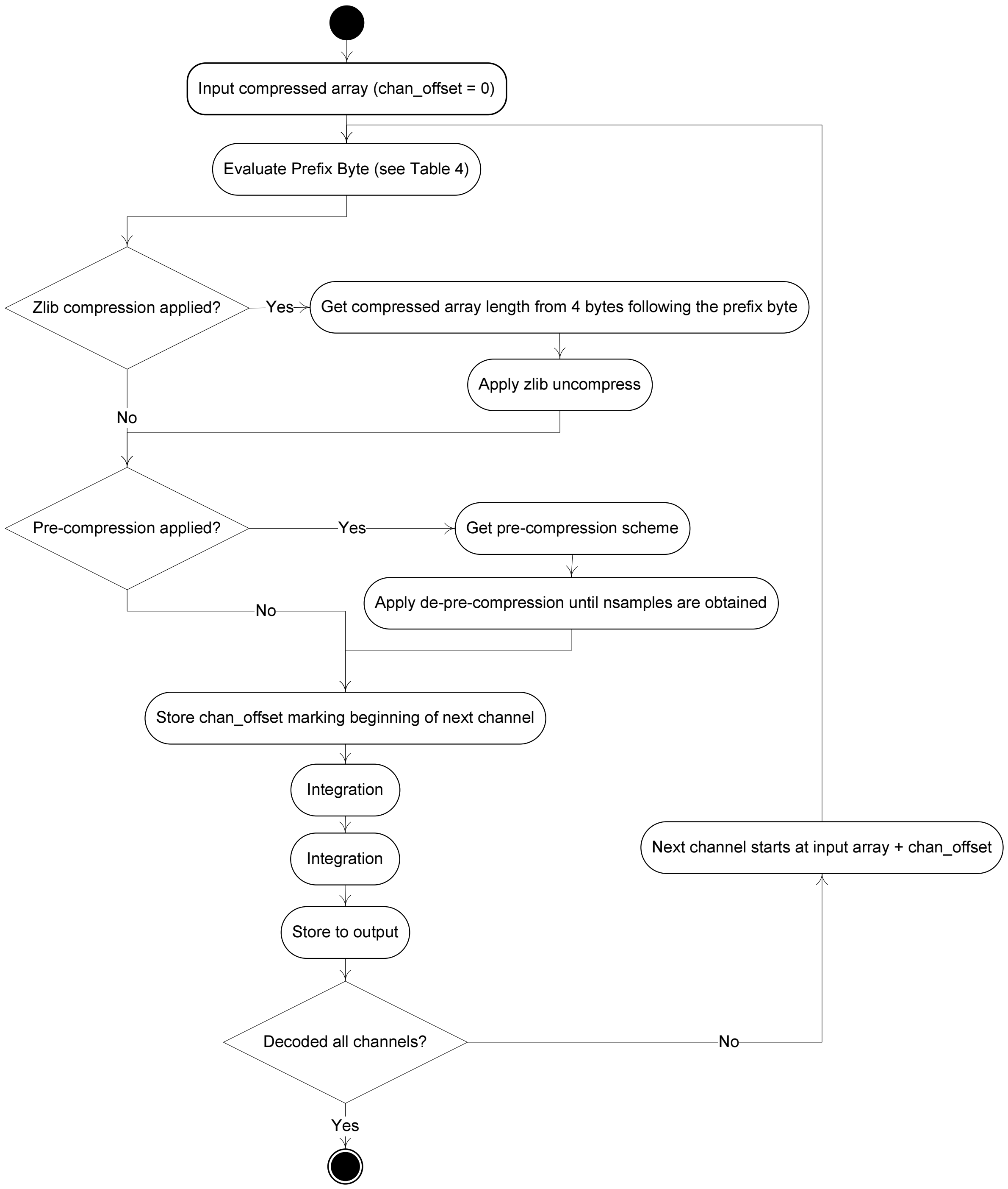

Input compressed array (chan_offset = 0)
Evaluate Prefix Byte (see Table 4)
Zlib compression applied?
Yes
Get compressed array length from 4 bytes following the prefix byte
Apply zlib uncompress
No
Pre-compression applied?
Yes
Get pre-compression scheme
Apply de-pre-compression until nsamples are obtained
No
Store chan_offset marking beginning of next channel
Integration
Integration
Store to output
Next channel starts at input array + chan_offset
Decoded all channels?
No
Yes

Figure 3: Decoding procedure. The steps concerning the decompression, de-pre-compression and integration are further detailed in the text.