Copyright and Trademarks

This document is copyrighted © 2015 by BESA GmbH. All rights reserved
BESA® is a registered trademark of BESA GmbH, Gräfelfing, Germany.
All other trademarks, brands and product names used in this specification are trademarks of the respective owners.

Document name: BESA File Format Specification
Revision number: 007
Revision date: 14. December 2015
# Table of Contents

## 1. Introduction
- 1.1. File extension ................................................................. 4
- 1.2. Tagged file format ............................................................. 4
- 1.3. File compression ............................................................... 4

## 2. BESA File Format
- 2.1. Header Block ........................................................................ 6
- 2.2. File Main Info Block (FMI Block) .............................................. 7
  - 2.2.1. Address Block ................................................................. 12
  - 2.2.2. Staff Block ................................................................... 13
- 2.3. Channel And Location Block (BSA Block) ........................... 14
- 2.4. Tag List Block ...................................................................... 21
- 2.5. Event Block .......................................................................... 22
  - 2.5.1. Base Event ...................................................................... 24
  - 2.5.2. Comment Event .............................................................. 25
  - 2.5.3. Marker Event ................................................................. 25
  - 2.5.4. Segment Start Event ........................................................ 26
  - 2.5.5. Average Segment Start Event .......................................... 27
  - 2.5.6. Multiple Pattern Search Event ......................................... 27
  - 2.5.7. Classified Multiple Pattern Search Event ......................... 27
  - 2.5.8. Pattern Event ................................................................. 27
  - 2.5.9. Trigger Event ................................................................. 28
  - 2.5.10. Paired Event ............................................................... 28
  - 2.5.11. Artifact Event .............................................................. 28
  - 2.5.12. Epoch Event ............................................................... 29
  - 2.5.13. Impedance Event ........................................................ 29
  - 2.5.14. Generic Event ............................................................. 30
- 2.6. Data Block ........................................................................... 31

## 3. File Compression format
- 3.1. Encoding ............................................................................. 33
  - 3.1.1. Differentiation ............................................................... 35
  - 3.1.2. Pre-Compression .......................................................... 35
  - 3.1.3. ZLib Compression ........................................................ 38
  - 3.1.4. Final Output ............................................................... 40

- 3.2. Decoding ............................................................................. 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1.</td>
<td>ZLib Decompression</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2.</td>
<td>De-Pre-Compression</td>
<td>40</td>
</tr>
<tr>
<td>3.2.3.</td>
<td>Integration</td>
<td>41</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>data element</th>
<th>data section</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>offset</td>
</tr>
<tr>
<td>ID</td>
<td>offset</td>
</tr>
<tr>
<td>4 Bytes:</td>
<td>4 Bytes:</td>
</tr>
<tr>
<td>data section ID</td>
<td>data section size</td>
</tr>
<tr>
<td>&lt;data section size&gt; Bytes:</td>
<td>data (format depends on data section ID)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>header block</th>
<th>next block</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCF1 offset data section</td>
<td>ID offset data section</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>file main info block</th>
<th>next file main info block</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BFMI offset FMI offset data section</td>
<td>BFMI offset FMI offset data section</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>channel and location block</th>
<th>next channel and location block</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCAL offset CAL offset data section</td>
<td>BCAL offset CAL offset data section</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag list block</th>
<th>next tag list block</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BTAG offset TAG offset data section</td>
<td>BTAG offset TAG offset data section</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>event block</th>
<th>next block</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BEVT offset data section</td>
<td>ID offset data section</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data block</th>
<th>next block</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BDAT offset data section</td>
<td>ID offset data section</td>
<td></td>
</tr>
</tbody>
</table>

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 (0xFFFFFFFFF) 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 0xFFFFFFFF, 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:

<table>
<thead>
<tr>
<th>Tag ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCF1</td>
<td>File version</td>
</tr>
<tr>
<td>VERS</td>
<td>Offset/Link to the first file main info block</td>
</tr>
<tr>
<td>OFFM</td>
<td>Offset/Link to the first tag list block</td>
</tr>
<tr>
<td>OFTL</td>
<td>Offset/Link to the first channel and location block</td>
</tr>
</tbody>
</table>

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:

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATN</td>
<td>Subject last name</td>
</tr>
<tr>
<td>PNAT</td>
<td>Subject title</td>
</tr>
<tr>
<td>PATD</td>
<td>Subject date of birth</td>
</tr>
<tr>
<td>PDOD</td>
<td>Subject date of death</td>
</tr>
<tr>
<td>PAGE</td>
<td>Subject gender</td>
</tr>
<tr>
<td>PAWE</td>
<td>Subject weight</td>
</tr>
<tr>
<td>PAHE</td>
<td>Subject height</td>
</tr>
<tr>
<td>PAMS</td>
<td>Subject marital status</td>
</tr>
<tr>
<td>PAAD</td>
<td>Subject address</td>
</tr>
<tr>
<td>PALA</td>
<td>Subject language</td>
</tr>
<tr>
<td>PAMH</td>
<td>Subject medical history</td>
</tr>
<tr>
<td>PATC</td>
<td>Subject comment</td>
</tr>
<tr>
<td>PAT1</td>
<td>Subject ID</td>
</tr>
<tr>
<td>INF1</td>
<td>Additional information 1</td>
</tr>
<tr>
<td>INF2</td>
<td>Additional information 2</td>
</tr>
</tbody>
</table>

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.
2.2. File Main Info Block (FMI Block)

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.
2.2 File Main Info Block (FMI Block)

**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}. 
2.2. File Main Info Block (FMI Block)

**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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTR</td>
<td>Street name</td>
</tr>
<tr>
<td>ASTA</td>
<td>State</td>
</tr>
<tr>
<td>ACIT</td>
<td>City</td>
</tr>
<tr>
<td>APOS</td>
<td>Post code</td>
</tr>
<tr>
<td>ACOU</td>
<td>Country</td>
</tr>
<tr>
<td>APHO</td>
<td>Phone number</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>SNAM</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SINI</td>
<td>Initials</td>
</tr>
<tr>
<td>SFUN</td>
<td>Function</td>
</tr>
</tbody>
</table>

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:

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

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

**COMC**

BSA CTF component.

For internal use only (BSACtfComponentsArray).

**COMH**

BSA head transformation

For internal use only (BSAHeadTransformationStruct).
2.3. Channel And Location Block (BSA Block)

**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!
2.3. Channel And Location Block (BSA Block)

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).
2.3 Channel And Location Block (BSA Block)

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.
2.3. Channel And Location Block (BSA Block)

**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_HIGHCUTOFF_TYPE_ZERO_PHASE)
  - 0x00000000 (FLT_HIGHCUTOFF_TYPE_FORWARD)
  - 0x00000000 (FLT_HIGHCUTOFF_TYPE_BACKWARD)
  These flags specify the high cutoff filter type.
  - 0x00100000 (FLT_NOTCH_REMOVE_2ND_HARMONIC)
  - 0x00200000 (FLT_NOTCH_REMOVE_3RD_HARMONIC)
  If one of these flags 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.
2.3 Channel And Location Block (BSA Block)

**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 characteres.
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 (\texttt{<__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:

<table>
<thead>
<tr>
<th>Tag ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTAG</td>
<td>Offset/Link to the next tag list block (UNTAGGED DATA)</td>
</tr>
<tr>
<td>TAGE</td>
<td>Tag list entry</td>
</tr>
</tbody>
</table>

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:

```
BEVT

LIST

HEAD

EVTS  Number of events
VERS  Version number

<event type>

<event type tag>
<event type tag>
...

<event type>

<event type tag>
<event type tag>
...

...

<Number of events> 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 Event Block

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:

<table>
<thead>
<tr>
<th>Tag ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMP</td>
<td>(Zero-based) file sample index</td>
</tr>
<tr>
<td>TIME</td>
<td>Event time</td>
</tr>
<tr>
<td>SIDX</td>
<td>(Zero-based) segment index</td>
</tr>
<tr>
<td>CODE</td>
<td>(Zero-based) event code</td>
</tr>
<tr>
<td>EVID</td>
<td>(Zero-based) internal BESA event ID</td>
</tr>
<tr>
<td>STAT</td>
<td>Event state</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>WORD</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td></td>
</tr>
<tr>
<td>Month</td>
<td></td>
</tr>
<tr>
<td>DayOfWeek</td>
<td></td>
</tr>
<tr>
<td>Day</td>
<td></td>
</tr>
<tr>
<td>Hour</td>
<td></td>
</tr>
<tr>
<td>Minute</td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td></td>
</tr>
<tr>
<td>Milliseconds</td>
<td></td>
</tr>
<tr>
<td>Microseconds</td>
<td></td>
</tr>
<tr>
<td>State flag (for internal use).</td>
<td></td>
</tr>
</tbody>
</table>

**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.
2.5. Event Block

**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:

<table>
<thead>
<tr>
<th><strong>COMM</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;...&quot;</td>
</tr>
<tr>
<td><strong>TEXT</strong></td>
</tr>
<tr>
<td>Data elements of <strong>BASE</strong> section</td>
</tr>
<tr>
<td>See description of base event section</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th><strong>MARK</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;...&quot;</td>
</tr>
<tr>
<td>Data elements of <strong>BASE</strong> section</td>
</tr>
<tr>
<td>See description of base event section</td>
</tr>
</tbody>
</table>
### 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:

<table>
<thead>
<tr>
<th>Tag ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGM</td>
<td>Data elements of COMM section See description of comment event section</td>
</tr>
<tr>
<td>SBEG</td>
<td>Segment start time</td>
</tr>
<tr>
<td>DAYT</td>
<td>Day time of segment start in microseconds</td>
</tr>
<tr>
<td>INTE</td>
<td>Sampling interval in microseconds</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>ASGM</th>
<th>Data elements of <strong>COMM</strong> section</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>See description of comment event section</td>
</tr>
<tr>
<td>STIM</td>
<td>Prestimulus interval in microseconds</td>
</tr>
<tr>
<td>AVRS</td>
<td>Number of averages</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>PATT</th>
<th>Data elements of <strong>BASE</strong> section</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>See description of base event section</td>
</tr>
</tbody>
</table>
2.5.9. Trigger Event

The trigger event is identified by the tag ID \texttt{TRIG}.

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

\begin{table}[h!]
\centering
\begin{tabular}{|c|}
\hline
\texttt{TRIG} \\
\hline
\texttt{...} \\
\texttt{...} \\
\texttt{CODE} \\
\texttt{TIME} \\
\hline
\end{tabular}
\end{table}

Data elements of \texttt{COMM} section

See description of comment event section

Reaction code

Reaction time

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

\textbf{CODE}

Event reaction code.

Stored as 32-bit integer value.

\textbf{TIME}

Event reaction time in seconds.

Stored as float value.

2.5.10. Paired Event

The paired event is identified by the tag ID \texttt{PAIR}.

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

\begin{table}[h!]
\centering
\begin{tabular}{|c|}
\hline
\texttt{PAIR} \\
\hline
\texttt{...} \\
\texttt{...} \\
\texttt{PART} \\
\hline
\end{tabular}
\end{table}

Data elements of \texttt{COMM} section

See description of comment event section

Event information of the partner event.

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

\textbf{PART}

Event information of the partner event.

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

2.5.11. Artifact Event

The artifact event is identified by the tag ID \texttt{ARTI}.

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

\begin{table}[h!]
\centering
\begin{tabular}{|c|}
\hline
\texttt{ARTI} \\
\hline
\texttt{...} \\
\texttt{...} \\
\hline
\end{tabular}
\end{table}

Data elements of \texttt{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:

<table>
<thead>
<tr>
<th>EPOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Data elements of PAIR section
See description of paired event section

2.5.13. Impedance Event

The impedance event is identified by the tag ID IMP.
The data section consists of block specific data elements with following tag IDs:

<table>
<thead>
<tr>
<th>IMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Data elements of BASE section
See description of base event section

<table>
<thead>
<tr>
<th>FORM</th>
<th>NR</th>
<th>TYPE</th>
<th>LABL</th>
<th>VAL</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>GENE</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDAT</td>
<td>Data type of written data (int16 or float, compressed or not)</td>
</tr>
<tr>
<td>DATT</td>
<td>Number of samples in data block</td>
</tr>
<tr>
<td>DATS</td>
<td>Data</td>
</tr>
<tr>
<td>DATA</td>
<td>Data type of written data (int16 or float, compressed or not)</td>
</tr>
</tbody>
</table>

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 32-bit-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:

- 0x00000UL (BESA_DATA_TYPE_UNCOMPRESSED)
  
  Do not use compression.

- 0x0001UL (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.
3.1. Encoding

Input Array "v" (32 bit int)

Get nsamples Entries for current Channel

Differentiate

Differentiate

Maximum Compression selected?

No

Apply First Pre-Compression Scheme

Any Value > 16 bits ?

No

Apply zlib compression

Compression improved?

No

Restore Previous State

Yes

Apply Second Pre-Compression Scheme

Apply Third Pre-Compression Scheme

Determine Best Pre-Compression

Pre-Compression rate > 1?

No

Store 2 First Values as short Type to the buffer

Store 2 First Values as int Type to the buffer

Store 2nd Difference Array as 16 bit

Yes

Gone through all channels?

Yes

Increase Channel Counter

Store in Byte Array with Prefix Byte (see Table 4)

No

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 \( n_{\text{samples}} \) 32-bit integer entries (hereafter simply \( \text{int} \)) we compute the difference vector \( d \) with \( n_{\text{samples}} \) entries of type \( \text{int} \) according to

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

where \( i=1, \ldots, n_{\text{samples}}-1 \) is the index. Furthermore we choose \( d[0] = v[0] \).

The same procedure is applied to obtain the double difference vector \( dd \) of length \( n_{\text{samples}} \) and type \( \text{int} \).

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

where \( i=2, \ldots, n_{\text{samples}}-1 \) 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 \( \text{buffer} \).

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 \( \text{short int} \) type (16-bit) values, provided they fit, or as \( \text{int} \) 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 \( m \times 4 \) 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 \( \text{buffer} \) 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

\[
\text{index} = j + i \times \Delta
\]

where \( \Delta \) 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 \((\Delta - 1)/2\) to the respective \( dd[q], dd[q+1] \) to be encoded. In the example in Table 1, where \( dd[q] = 6 \) and \( dd[q+1] = -4 \),

\[
i = 6 + (15 - 1)/2 = 13 \text{ and } j = -4 + 7 = 3.
\]
The obtained index is then stored in the output byte array buffer.

### 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:

\[
\text{index} = k + (j + i \cdot \Delta) \cdot \Delta
\]

Here \( \Delta = 5 \) 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.

\[
\text{index} = j + i \cdot \Delta + 125
\]

The width of the table is \( \Delta = 11 \) 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

\[
\text{index} = l + (k + (j + i \cdot \Delta) \cdot \Delta) \cdot \Delta
\]

The width \( \Delta = 3 \) 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 \(d[d[q]], d[d[q+1]]\) both fall in the interval \([-6, 6]\). Thus we have \(\Delta = 13\) and \(i, j\) belong to the interval \([0, 12]\). The index carries an offset of 81 to avoid conflicts with (a)

\[
\text{index} = j + i \times \Delta + 81
\]

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.

<table>
<thead>
<tr>
<th>Pre-Compression Scheme</th>
<th>#Sequential Entries to be Compressed</th>
<th>Value Range</th>
<th>Index Range</th>
<th>(\Delta)</th>
<th>Table Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>2</td>
<td>[-7,7]</td>
<td>[0,224]</td>
<td>15</td>
<td>(j + i \times \Delta)</td>
</tr>
<tr>
<td>Second a</td>
<td>3</td>
<td>[-2,2]</td>
<td>[0,124]</td>
<td>5</td>
<td>(k + (j + i \times \Delta) \times \Delta)</td>
</tr>
<tr>
<td>Second b</td>
<td>2</td>
<td>[-5,5]</td>
<td>[125,245]</td>
<td>11</td>
<td>(j + i \times \Delta + 125)</td>
</tr>
<tr>
<td>Third a</td>
<td>4</td>
<td>[-1,1]</td>
<td>[0,80]</td>
<td>3</td>
<td>(l + (k + (j + i \times \Delta) \times \Delta) \times \Delta)</td>
</tr>
<tr>
<td>Third b</td>
<td>2</td>
<td>[-6,6]</td>
<td>[81,249]</td>
<td>13</td>
<td>(j + i \times \Delta + 81)</td>
</tr>
</tbody>
</table>

Table 2: Summary of the pre-compression schemes. Note that the indexes \(i,j,k,l\) run from \([0, \Delta - 1]\). Add \((\Delta - 1) / 2\) 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 ints or ints, 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.

<table>
<thead>
<tr>
<th>Pre-Compression Scheme</th>
<th>maximum sequential inputs</th>
<th>interval for (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>byte (8bits)</td>
<td>short int (16bits)</td>
</tr>
<tr>
<td>First</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Second</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Third</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

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. Encoding

3.1.2.4. Pre-Compression Criterion

The length of the second difference array in bytes is $n_{samples} \times 4$ (as it is an array of 32-bit integers). The pre-compressed buffer shall have a maximum size in bytes corresponding $n_{samples} \times 2$. 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.

<table>
<thead>
<tr>
<th>Prefix Byte value</th>
<th>pre-compression mode / compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No Compression</td>
</tr>
<tr>
<td>3</td>
<td>First Scheme</td>
</tr>
<tr>
<td>4</td>
<td>Second Scheme</td>
</tr>
<tr>
<td>5</td>
<td>Third Scheme</td>
</tr>
<tr>
<td>6</td>
<td>No Compression + first 2 entries $\text{int}$</td>
</tr>
<tr>
<td>7</td>
<td>First Scheme + first 2 entries $\text{int}$</td>
</tr>
<tr>
<td>8</td>
<td>No Compression (all $\text{int}$)</td>
</tr>
<tr>
<td>9</td>
<td>zlib on Second Difference $\text{dd}$</td>
</tr>
<tr>
<td>13</td>
<td>zlib on First Scheme</td>
</tr>
<tr>
<td>14</td>
<td>zlib on Second Scheme</td>
</tr>
<tr>
<td>15</td>
<td>zlib on Third Scheme</td>
</tr>
<tr>
<td>17</td>
<td>zlib on First Scheme + first 2 entries $\text{int}$</td>
</tr>
<tr>
<td>18</td>
<td>zlib on Second Scheme + first 2 entries $\text{int}$</td>
</tr>
<tr>
<td>19</td>
<td>zlib on Third Scheme + first 2 entries $\text{int}$</td>
</tr>
<tr>
<td>29</td>
<td>zlib on Second Difference $\text{dd}$ (all $\text{int}$)</td>
</tr>
</tbody>
</table>

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 $\text{compress}$ 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.
3.2. Decoding

3.2.1. ZLib Decompression

The inverse to 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. Decoding

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.

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