EDF data format specific file access routines

Module edfio (Data Format Version 2.30)

SYNOPSIS

# include edfio.h

INCLUDE FILES

edfio.h

bslio.h

TO LINK WITH

bslio.c

RESTRICTIONS

The file size is limited by the long integer format. On machines with   4 byte long integers additional blocks after 2^31 bytes = 2Gbytes cannot   be accessed. Therefore, if a file contains several data blocks the   absolute size of a single data file should not exceed 2GBytes.

DESCRIPTION

EDF data format specific file access routines (read and write routines) The file format is described in ´SaxsKeywords.pdf´. The data files can contain several data blocks that are internally accessed by block keys and chain keys. A block key is a sequence number (positive or negative). A chain key consists of several parts: a text part (a: "Image.Psd" or b: "Image.Error") followed by an unsigned number part (memory). Different memories can contain data from different detectors. The different parts are separated by a dot, except for memory 1 that is not explicitly written with a number. A data block consists of a text block ("ASCII header") followed by a binary block ("binary data"). The block type is written at the top of the text block after the keyword "EDF_DataBlockID". The value "1.Image.Psd" marks data block number 1 of memory 1 and "1.Image.Psd.2" marks block number 1 of memory 2. Positive memories contain always primary (scientific) data (extension "Psd"), negative memories contain always error data (extension "Error").

Internally, a general chain is defined. It has memory number 0 and data number 0.

Data blocks with different chain numbers but the same data numbers belong together (e.g. data array (chain number 1) and associated error array (chain number 2)).

The input routines can also read bsl files. Bsl frame numbers (frame) and memory numbers (memory) are internally converted to edf data numbers and chain numbers:

memory	frame	Chain Number	Data Number	chain key	block key	block id
1	n	1	n	Image.Psd	n	n.Image.Psd
m	n	m	n	Image.m	n	n.Image.Psd.m

EXAMPLE READ DATA

Remark: Only a single data block of a data chain can be present in memory at a given time. Subsequent reading deallocates the header data of the previous block and allocates memory for the header of the block from which is read. Binary data is not automatically deallocated and must explicitely be freed, before reading another binary data.

Read all images from data chain 1 of an edf file and convert to float.

```
# include <errno.h>
# include <stdio.h>
# include <unistd.h>
# include <fcntl.h>

# include "edfio.h"

main ()
{
  char *fnam = "datafile.edf";
  int stream;
  float *Data;
  int DataChain = 1;
  long MinNumber, MaxNumber;
  long DataNumber;
  long Dim[4];
  long DataArraySize;
  int ErrorValue;
  int status;

  stream = edf_open_data_file ( fnam, "old", &ErrorValue, &status );
  if (status) return;

  edf_search_minmax_number ( stream, DataChain, &MinNumber, &MaxNumber,
                             &ErrorValue, &status);
  if (status) return;
  for ( DataNumber = MinNumber; DataNumber <= MaxNumber; DataNumber++) {
    edf_read_data_2d_float ( stream, DataNumber,
                             DataChain, Dim, &DataArraySize, &Data,
                             &ErrorValue, &status );
    if (status) return;
    if (Data) free(Data);
    }
  edf_close_data_file ( stream, &ErrorValue, &status );
  if (status) return;
}
```

EXAMPLE WRITE DATA

Remark: Only a single data block of a data chain can be present in memory at a given time. Header values must be written before binary data is written. After writing, header and binary data are automatically deallocated and are not available any more.

Write 10 images with 512X512 float pixels into an edf file.

```
# include <errno.h>
# include <stdio.h>
# include <unistd.h>
# include <fcntl.h>

# include "edfio.h"

main ()
{
  int cols=512, rows=512;

  int stream;
  char *fname = "datafile.edf";
  float *Data;
  int DataChain = 1;
  long MinNumber = 1, MaxNumber = 10;
  long DataNumber;
  long Dim[4];
  int ErrorValue;
  int status;

  Data = (float *) malloc(sizeof(float)*cols*rows);

  Dim[0] = (long) 2;
  Dim[1] = (long) cols;
  Dim[2] = (long) rows;
  stream = edf_open_data_file ( fname, "new", &ErrorValue, &status );
  if (status) return;

  for ( DataNumber = MinNumber; DataNumber <= MaxNumber; DataNumber++) {
    // ... here, write header values ...
    edf_write_data_2d_float ( stream, DataNumber,
                              DataChain, Dim, Data,
                              &ErrorValue, &status );
    if (status) return;
    }
  edf_close_data_file ( stream, &ErrorValue, &status );
  if (status) return;

}
```

Setup Routines

edf_general_block

Write/don´t write file with general header

SYNOPSIS
```
int edf_general_block( int writetodisk );
```

DESCRIPTION

writetodisk : 1, write general header

writetodisk : 0, do not write general header (default)

RETURN VALUE

0: success

edf_set_bsl_input_byteorder

Set the byteorder for all bsl input files

SYNOPSIS
```
int edf_set_bsl_input_byteorder( int byteorder )
```

DESCRIPTION

Changes the byte order of all bsl input files to byteorder
- byteorder : HighByteFirst, big endian byte order
- byteorder : LowByteFirst, little endian byte order

(default byte order: INTERNAL_BYTEORDER)

RETURN VALUE

0: success

File Access Routines

edf_search_minmax_number

SYNOPSIS
```
int edf_search_minmax_number(int stream, int DataChain,
                             long int *pMinNumber, long int *pMaxNumber,
                             int *pErrorValue, int *pstatus)
```

DESCRIPTION

Reads the headers in 'DataChain' and searches for the minimum and maximum data number.

Parameters

int DataChain	(i)	: data chain (0: general, 1: key_1, 2: key_2)
long int * pMinNumber	(o)	: minimum data number

long int * pMaxNumber (o) : maximum data number

int *pErrorValue (o) : error value

int *pstatus (o) : SAXS status

RETURN VALUE

1 : successful
0 : failed

edf_read_data_2d_dimension

SYNOPSIS

```
int edf_read_data_2d_dimension ( int stream, long int DataNumber,
                                 int DataChain, long int Dim[],
                                 long int * pDataArraySize,
                                 int * pErrorValue, int * pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain' and inquires for the dimension of the two dimensional data array.

If the header or the dimension of the data array does not exists, the return value is FALSE and a specific error value is returned. This error is not fatal and can be used as a test for the existence of a header.

Parameters

long int DataNumber (i) : data number

int DataChain (i) : data chain (0: general, 1: key_1, 2: key_2)

long int Dim[3] (o) : Dim[0] Dimension, Dim[1], Dim[2]

long int * pDataArraySize (o) : total size of float data array in bytes

int *pErrorValue (o) : error value

int *pstatus (o) : SAXS status

RETURN VALUE

1 : data header found, if *pstatus == Success
0 : data header not found,
    *pstatus == status_error;
    *pErrorValue == CouldNotFindHeader;

HISTORY

edf_open_data_file

Open file 'fname' with "new", "old", "any" or "read"

SYNOPSIS

```
int edf_open_data_file (  const char *fname, const char * mode,
                               int *pErrorValue, int *pstatus );
```

DESCRIPTION

Opens the file 'fname' with mode "new", "old", "any" or "read" and return a stream (success: 0 .. MaxFiles-1, error: -1).

- "new": open a new file for read/write, an existing file with the same name is overwritten
- "old": open an existing file for read/write and check file format
- "any": open either an existing file and check its file format or open a new file for read/write
- "read": open an existing file for read and check its file format

If an existing file is opened with "old", "any" or "read" its file start marker is checked and according to it is opened as an edf file or a bsl file.

RETURN VALUE

-1: failed

0 or positive number: stream

HISTORY

22-Mar-1998 Peter Boesecke

edf_close_data_file

Close an edf data file

SYNOPSIS

```
void edf_close_data_file ( int stream, int *pErrorValue, int *pstatus );
```

DESCRIPTION

Closes the edf data file stream

edf_write_header

Writes a header symbol list into the header

SYNOPSIS

```
int edf_write_header ( int stream, long int DataNumber,
                         int DataChain, HSymb * symbollist,
                         int * pErrorValue, int * pstatus );
```

DESCRIPTION

RETURN VALUE

return value FALSE if not found and no other error

```
  return( int ) FALSE : data header not found,
                    *pstatus = status_error;
                    *pErrorValue=(CouldNotFindHeader, RoutineSucceeded);
              TRUE  : data header found or error,
                    *pstatus = Success or status_error;
                    *pErrorValue = <any>
```

HISTORY

currently not available

edf_write_header_line

SYNOPSIS

```
int edf_write_header_line(int stream,long int DataNumber,int DataChain,
                          const char * keyword, const char * Value,
                          int * pErrorValue, int * pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'.   If it does not exists, it is created. It writes keyword and value   into the header. The value is a character string.

RETURN VALUE

In case of success the return value is 1, otherwise 0.

edf_write_header_float

SYNOPSIS

```
int edf_write_header_float(int stream,long int DataNumber,int DataChain,
                           const char * keyword, float Value,
                           int * pErrorValue, int * pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'.   If it does not exists, it is created. It writes keyword and value as   text into the header. The value is a float value.

RETURN VALUE

In case of success the return value is 1, otherwise 0.

edf_write_header_long

SYNOPSIS
```
int edf_write_header_long(int stream,long int DataNumber,int DataChain,
                          const char * keyword, long int Value,
                          int * pErrorValue, int * pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'.   If it does not exists, it is created. It writes keyword and value as   text into the header. The value is a long integer value.

RETURN VALUE

In case of success the return value is 1, otherwise 0.

edf_write_data_2d_raw

Write 2d data without conversion

SYNOPSIS
```
void edf_write_data_2d_raw ( int stream, long int DataNumber,
                             int DataChain, long int Dim[],
                             void *pData, int DataType,
                             long DataValueOffset, int ByteOrder,
                             long RasterConfiguration,
                             int * pErrorValue, int *pstatus );
```

DESCRIPTION
Searches for header 'DataNumber' in 'DataChain'.   If it does not exists, it is created.   The data type, data size and array dimensions are written into the header.   A 2d float data array (float Data[Dim[1]][Dim[2]) is written after   the end of the header.

Parameters

flat * pData	(i)	pointer to the start of the data array
long int Dim[0]		(reserved)
Dim[1]	(i)	dimension 1
Dim[2]	(i)	dimension 2

HISTORY

23-Jul-1999 DataValueOffset added

edf_write_data_2d_float

Writes 2d float data to file

SYNOPSIS
```
void edf_write_data_2d_float ( int stream, long int DataNumber,
                           int DataChain, long int Dim[], float *pData,
                           int * pErrorValue, int *pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'. If it does not exists, it is created. The data type, data size and array dimensions are written into the header. A 2d float data array (float Data[Dim[1]][Dim[2]) is written after the end of the header.

Parameters:

float *pData	(i)	pointer to the start of the data array
long int Dim[0]		(reserved)
Dim[1]	(i)	dimension 1
Dim[2]	(i)	dimension 2

HISTORY

edf_read_header

Reads all user keys and values from a header

SYNOPSIS
```
int edf_read_header( int stream, long int DataNumber,
                     int DataChain, HSymb ** psymbollist,
                     int * pErrorValue, int * pstatus);
```

DESCRIPTION

Reads all user keys and values from a header and returns them in symbollist.

RETURN VALUE

return value FALSE if not found and no other error

return(int) FALSE : data header not found,

*pstatus = status_error;

*pErrorValue=(CouldNotFindHeader, RoutineSucceeded);

TRUE : data header found or error,

*pstatus = Success or status_error;

*pErrorValue = <any>

HISTORY

currently not available

edf_read_header_line

Reads a line

SYNOPSIS
```
int edf_read_header_line(int stream,long int DataNumber,int DataChain,
                         const char * keyword, char * Value,
                         int * pErrorValue, int * pstatus );
```

DESCRIPTION

Searches for 'keyword' in the header 'DataNumber' in 'DataChain'.   If the header or the keyword does not exists, the return value is 0 and a   specific error value is returned. This error is not fatal and can be used   as a test for the existence of the keyword or the header. The ´Value´   string specified by ´keyword´ is copied after the location pointed to by Value. The minimum allocated size for Value must be MaxValLen+1.

RETURN VALUE

return value FALSE if not found and no other error

  return(int) FALSE : data header not found,

            *pstatus = status_error;

            *pErrorValue=(CouldNotFindHeader, RoutineSucceeded);

        TRUE : data header found or error,

            *pstatus = Success or status_error;

            *pErrorValue = <any>

  --+*/

int edf_read_header_line (int stream, long int DataNumber, int DataChain,

            const char * keyword, char * Value,

            int * pErrorValue, int * pstatus)   /*---*/

edf_read_header_float

Reads a float value

SYNOPSIS
```
int edf_read_header_float(int stream,long int DataNumber,int DataChain,
                          const char * keyword, float * Value,
                          int * pErrorValue, int * pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'.   In this headers it searches for 'keyword'. If the header or the keyword   do not exists, the return value is 0 and a specific error value   is returned. This error is not fatal and can be used as a test for the   existence of the keyword or the header.   A pointer to the float value specified by 'keyword' is returned   in 'Value'.

RETURN VALUE

return value FALSE if not found and no other error

  return(int) FALSE : data header not found,

                    *pstatus = status_error;

                    *pErrorValue=(CouldNotFindHeader, RoutineSucceeded);

            TRUE : data header found or error,

                    *pstatus = Success or status_error;

                    *pErrorValue = <any>

edf_read_header_long

Reads a long integer value

SYNOPSIS

```
int edf_read_header_long(int stream,long int DataNumber,int DataChain,
                             const char * keyword, long int * Value,
                             int * pErrorValue, int * pstatus );
```

DESCRIPTION

   Searches for header 'DataNumber' in 'DataChain'.   In this headers it searches for 'keyword'. If the header or the keyword   do not exists, the return value is 0 and a specific error value   is returned. This error is not fatal and can be used as a test for the   existence of the keyword or the header.   A pointer to the long int value specified by 'keyword' is returned   in 'Value'.

RETURN VALUE

return value FALSE if not found and no other error

  return(int) FALSE : data header not found,

                    *pstatus = status_error;

                    *pErrorValue=(CouldNotFindHeader, RoutineSucceeded);

            TRUE : data header found or error,

                    *pstatus = Success or status_error;

                    *pErrorValue = <any>

  --+*/

edf_read_data_2d_raw

Reads 2d data without type conversion

SYNOPSIS

```
void edf_read_data_2d_raw ( int stream, long int DataNumber,
                            int DataChain, long int Dim[],
                            long int *pDataArraySize, void **ppData,
                            int * pDataType, long *pDataValueOffset,
                            int *pByteOrder, long *pRasterConfiguration,
                            int *pErrorValue, int *pstatus );
```

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'. If it does not exist the routine stops with an error. A data array with the dimension Data[Dim[1],Dim[2]] is read from the file. The pointer &&Data[0,0] is returned in ppData. The data type, the data value offset, the byte order and the raster configuration of the array are returned.

If the stored array is only 1 dimensional, the data is read with Dim[2] set to 1.

The returned array has a length of *pDataArraySize bytes. The data buffer is allocated and must be released explicitly.

Parameters:

long int	Dim[0]	(o) 2
	Dim[1]	(o) dimension 1
	Dim[2]	(o) dimension 2
long int	* pDataArraySize	(o) size of the data array in bytes
void	** pData	(o) pointer to the pointer of the data array
long	* pDataValueOffset	(o) data value offset of the array elements
int	* pDataType	(o) data type of the array elements (DType)

HISTORY

23-Jul-1999 PB pDataValueOffset, pByteOrder and pRasterConfiguration added

edf_read_data_2d_float

Read 2d data and convert to float

SYNOPSIS

void edf_read_data_2d_float (int stream, long int DataNumber,

int DataChain, long int Dim[],

long int * pDataArraySize, float **ppData, int * pErrorValue, int * pstatus);

DESCRIPTION

Searches for header 'DataNumber' in 'DataChain'. If it does not exist the routine stops with an error. A data array is read from the file and converted into a float array of the type float Data[Dim[1],Dim[2]]. The data array is allocated. The pointer &&Data[0,0] is returned in ppData. If the stored array is only 1 dimensional, the data is read with Dim[2] set to 1.

The data are read with the specification given in the header. All read data values are converted into float and the returned data array has a length of *pDataArraySize bytes. The data buffer is allocated and must be released explicitly.

Parameters:

```
long int   Dim[0]         (o) 2
           Dim[1]         (o) dimension 1
           Dim[2]         (o) dimension 2
long int * pDataArraySize (o) size of the data array in bytes
flat    ** pData          (o) pointer to the pointer of the data array
```

File History Routines

DESCRIPTION

The routines of this module are used to read and write history lines.

- 'int edf_history_new (void)' must be called first. It clears and initializes the internal history list and argument list.
- 'int edf_history_skip (void) ' marks the next argument as not required
- 'int edf_history_take (void) ' marks the next argument as required (default)
- 'int edf_history_argv (const char * substring)' is used to store the arguments of the call in an internal list. It can be called several times to pass all arguments.
- 'int edf_read_header_history (int stream, long int DataNumber,
  int DataChain,
  int * pErrorValue, int * pstatus) ' initializes the history list and reads the history strings from the file header.
- 'int edf_write_header_history (int stream, long int DataNumber,
  int DataChain,
  int * pErrorValue, int * pstatus)' writes the history strings into the output file header.
- 'int edf_history_free (void)' releases all internal lists.
- 'void hist_debug (int debug)' sets the module into debug mode.

The length of a history line is limited to MaxHistoryLineSize-1.

HISTORY

1999-06-24 V1.0 Peter Boesecke

edf_history_new

SYNOPSIS

```
int edf_history_new ( void )
```

DESCRIPTION

This routines must be called first. If not already initialized, it initializes all history lists. Eventually existing history lines are removed and a new empty history line with size MaxHistoryLineSize is created. ´edf_history_argv´ adds parameters to this line. ´edf_write_header_history´ appends it with a new key to the history lines that were read with ´edf_read_header_history´.

RETURN VALUES

In case of success the return value is 1, otherwise 0.

edf_history_skip

SYNOPSIS

```
int edf_history_skip ( void )
```

DESCRIPTION

This routines marks the next parameter that is passed to edf_history_argv as not required.

RETURN VALUES

In case of success the return value is 1, otherwise 0.

edf_history_take

SYNOPSIS

```
int edf_history_take ( void )
```

DESCRIPTION

This routines marks the next parameter that is passed to edf_history_argv as required (opposite of edf_history_skip)

RETURN VALUES

In case of success the return value is 1, otherwise 0.

edf_history_free

SYNOPSIS

```
int edf_history_free ( void )
```

DESCRIPTION

Removes all history lines.

RETURN VALUES

In case of success the return value is 1, otherwise 0.

edf_read_header_history

SYNOPSIS

```
int edf_read_header_history  (int stream, long int DataNumber,
                                 int DataChain,
                                 int * pErrorValue, int * pstatus )
```

DESCRIPTION

Reads all history lines from the date file header and copies them to 'hline->key's. History lines have the keyword HISTORY_KEY_PREFIX'u', where 'u' is an unsigned positive integer. A new history line with the key HISTORY_KEY_PREFIX'last+1' and MaxHistoryLineSize bytes is created.

RETURN VALUES

In case of success the return value is 1, otherwise 0.

edf_write_header_history

SYNOPSIS

```
int edf_write_header_history   ( int stream, long int DataNumber,
                                    int DataChain,
                                    int * pErrorValue, int * pstatus )
```

DESCRIPTION

Writes the history strings in 'history_root' into the date file header using the 'hline->key's as keywords. Writes the history string in 'history_argv_root' into the date file header using 'current_history_key' as keyword.

RETURN VALUES

In case of success the return value is 1, otherwise 0.

edf_history_argv

Appends an argument to history line

SYNOPSIS

```
int edf_history_argv ( const char * argument )
```

DESCRIPTION

Appends argument to history line

RETURN VALUES

In case of success the return value is 1, otherwise 0.

Additional Routines

edf_raster_normalization

Conversion to raster orientation 1

SYNOPSIS

```
int edf_raster_normalization ( void * dest, const void * src,
                               const long data_dim[],
                               long raster_configuration, size_t item )
```

DESCRIPTION

Conversion of the multi-dimensional array src with raster configuration number ´raster_configuration´ to the n-dimensional array dest with raster configuration number 1. The length n of data_dim is stored in data_dim[0]. The total length of the array is data_dim[n+1]. data_dim[i] is the length of coordinate i. Input and output array must be different. Sufficient memory must have been allocated for both arrays. The total number of elements in the array is specified in data_dim[n+1].

The raster configuration specifies only the way how data is stored. It does not influence the number of dimensions. Therefore, the dimension array is not changed even if, apparently, horizontal and vertical axes were changed.

ARGUMENTS

```
void * dest                    output array (must be allocated), must
                               be different from source array
  const void * src             source array
  const long data_dim[]        source dimensions
  long raster_configuaration   raster configuration number of source
                               array
  size_t item                  size of an array element
```

GENERAL

An n-dimensional array has ´N = 2^n * (n!)´ different ways of storing its data in a regular raster. Each of the n axes can be stored in two different ways: up and down. This results in 2^n different possibilities of data storage. The n axes can be stored in any of the (n!) possible permutations. This results in N = 2^n * (n!) different ways of storing the data. A two dimensional array (n=2) can be stored in 2^2 * 2! = 8 different ways, a three dimensional array (n=3) can be stored in 2^3 * 3! = 8*6 = 48 different ways, and so on.

The data elements are stored in an array with dim_1 * dim_2 * ... * dim_n cells with identical size. The number of dimension is n. The first index (i_1) is the fastest. A specific raster configuration is given by the n-tupel (k_2, -k_3, -k_1), which means that the fastest index i_1 corresponds to the coordinate k_2, the medium fast index i_2 corresponds to the invers k_3 direction and the slowest coordinate i_3 corresponds to the invers k_1 direction.

RASTER CONFIGURATION

Arrays can be stored in different ways depending on the relationship between offsets and fast and slow array indices. The configuration of the indices is specified by a raster configuration number D.

A unique raster configuration number for multi dimensional arrays is used that is defined on the basis of the following demands. The raster configuration number is called D in the text:

- Array indices are numbered from 1 to N.
- A one dimensional array can be stored from low array indices to high array indices (D=1) or from high array indices to low array indices (D=2).
- The array element Array(k_1, k_2, k_3, ... , k_N) is accessed with an offset I from the first element measured in element size

I = (i_1-1) + (i_2-1) * Dim[1] + (i_3-1) * Dim[2] * Dim[1] + ...
  = Sum[J=1,J<=N,J++](i_J * Product[i=1,i<=J,i++](Dim[i]))
  (Dim[0]==1)

with i_1, i_2, i_3, ... replaced by k_1, k_2, k_3, ... in a special order that is specified by the raster configuration number D.

The raster configuration is described by grouping the array indices k_nn and the offset indices i_nn, e.g. for a 2-dimensional array where the fast and slow indices are interchanged.

Example for N=2 and D=5:

i_n   1, 2
k_nn 2, 1

- - The raster configuration number D is 1, when the array indices are ordered from "fast" (i_1) to "slow" (i_N), e.g. when i_1 = k_1, i_2 = k_2, etc.
- - The raster configuration D of the n-dimensional sub-array

Array[Dim_1, Dim_2, ... Dim_n] is the same as for the (n+1) dimensional array Array[Dim_1, Dim_2, ... Dim_n, Dim_(n+1)] if Dim_(n+1) == 1.

D (Array[Dim_1, Dim_2, ... Dim_n, 1]) == D (Array[Dim_1, Dim_2, ... Dim_n])

These demands give a unique description of multi-dimensional raster conformations with a configuration number D.

The raster configuration is defined as follows:

For the definition it is necessary to distinguish strictly between the array indices k_1, k_2, etc. and the offset indices i_1, i_2, etc.

A offset index with a small number, e.g. i_1 runs faster than an index with a higher number, e.g. i_3. The definition of the raster orientation is based on a standard orthogonal coordinate system with the first coordinate (x_1) horizontal and the second coordinate (x_2) vertical with respect to the observer. The position of the observer must be defined elsewhere. In standard scattering geometry the observer is located at the sample position and is looking against the detector. The origin of the coordinate system is at the lower left corner of the detector, independently of any detector pixel readout coordinate. The direction of the third coordinate x_3 is found with the vector product x_1 X x_2 = x_3. It is pointing against the observer. This might be usedful for representing three dimensional objects. It should be possible to find higher dimensional coordinates accordingly.

An n-dimensional array has ´N = 2^n * (n!)´ different ways of storing its data in a regular raster. Each of the n axes can be stored in two different ways: up and down. This results in 2^n different possibilities of data storage. The n axes can be stored in any of the (n!) possible permutations. This results in N = 2^n * (n!) different ways of storing the data. A two dimensional array (n=2) can be stored in 2^2 * 2! = 8 different ways, a three dimensional array (n=3) can be stored in 2^3 * 3! = 8*6 = 48 different ways, and so on.

The data elements are stored in an array with dim_1 * dim_2 * ... * dim_n cells with identical size. The number of data elements is n. The first index is the fastest. A specific raster configuration is given by the n-tupel (x_2, -x_3, -x_1), which means that the fastest index i_1 corresponds to the coordinate x_2, the medium fast index i_2 corresponds to the invers x_3 direction and the slowest coordinate i_3 corresponds to the invers x_1 direction. If the array has n_1, n_2 and n_3 elements in each direction the data origin X0 = (0,0,0) in the real world corresponds to the array element IX0 = (0,n_2,n_3).

CONFIGURATION NUMBERS

In the standard configuration (D=1) all array indices k_nn are identical to the offset indices i_nn.

In the 1-dimensional case the standard configuration (numbered 1) is the configuration in which the array index increases with the coordinate. The second configuration is where the index is antiparallel to the coordinate (numbered 2).

n=1 has two configurations (A(2) = 2):

raster_configuration D	Configuration
1	1
2	-1

If the number of configurations for n coordinates is A(n) = 2^n * (n!), the number of configurations for n+1 coordinates is given by

(1) 2 * (n+1) * A(n) = 2^(n+1) * (n+1)!

C(n) is the group of all possible configurations for n coordinates. If it is given, the configurations for n+1 coordinates can be built by inserting the configurations of the new coordinate (1 and -1) at each of the n+1 possible positions (n before each coordinate and 1 after the last coordinate). To have a well defined ordering the new coordinate is first added non-inverted after the end of all A(n) configurations and then inverted (A(n)*2). This is repeated subsequently from the end to the start before all n remaining positions.

```
(x1, x2, x3, ... , xn) -> (x1, x2, x3, ... , xn,  xn+1)
                          (x1, x2, x3, ... , xn, -xn+1)       +2
(x1, x2, x3, ... ,  xn+1, xn)
                          (x1, x2, x3, ... , -xn+1, xn)       +2
...
( xn+1, x1, x2, x3, ... , xn)
                          (-xn+1, x1, x2, x3, ... ,-xn)       +2
(n+1)*2
                               A(n) = 2^n * (n!)
```

The total number of configuration for n+1 coordinates is then

(2) 2 * (n+1) * A(n) = 2^(n+1) * (n+1)! ,

which is equal to A(n+1).

EXAMPLE

n	raster_configuration D	Configuration
1	1	1
1	2	-1
2	1	1, 2
2	2	-1, 2
2	3	1,-2

2	4	-1,-2
2	5	2, 1
2	6	2,-1
2	7	-2, 1
2	8	-2,-1
3	1	1, 2, 3
3	2	-1, 2, 3
3	3	1,-2, 3
3	4	-1,-2, 3
3	5	2, 1, 3
3	6	2,-1, 3
3	7	-2, 1, 3
3	8	-2,-1, 3
3	9	1, 2,-3
3	10	-1, 2,-3
3	11	1,-2,-3
3	12	-1,-2,-3
3	13	2, 1,-3
3	14	2,-1,-3
3	15	-2, 1,-3
3	16	-2,-1,-3
3	17	1, 3, 2
	...	
3	32	-2,-3,-1
3	33	3, 1, 2
	...	
3	48	-3,-2,-1

The raster configuration 13 for n=3 (2, 1,-3) means that the first offset index of the array (which is the fastest) corresponds to the coordinate k_2, the second index corresponds to the coordinate k_1 and the third index to the inverted coordinate k_3.

The largest D (D = A(n)) is always the conformation where the direction and order of all array indices are inverted.

RETURN VALUE

int status
0 : success
-1: failed

AUTHOR

13-Mar-1998 Peter Boesecke
17-May-1998 PB calculation of IS for positive order corrected:
IS[n_dim] = 0 changed to IS[n_dim] = 1.

edf_data_sizeof

Returns the size of a data type element

SYNOPSIS

DType data_type;

size_t edf_data_sizeof (int data_type);

DESCRIPTION

The size required for a data_type element is returned. Where data_type is the enum type DType.

RETURN VALUE

NULL : error (e.g. for invalid data type)

>NULL : size of data_type

AUTHOR

03-Mar-1998 PB Specification

edfio_version

SYNOPSIS

```
char *edfio_version         ( void )
```

DESCRIPTION

Returns a pointer to the version string of the module edfio

RETURN VALUE

Pointer to the version string

edf_dataformat_version

Return edf data format version string

SYNOPSIS
```
char * edf_dataformat_version ( void )
```

DESCRIPTION

Returns the edf data format version string.

RETURN VALUE

Pointer to edf data format version string.

edf_print_filetable

SYNOPSIS
```
int edf_print_filetable( FILE * out, int level, int verbose )
```

DESCRIPTION

Prints the current filetable to the file ´out´

RETURN VALUE

0: success

-1: failed

edf_report_data_error

Returns the error message of ErrorValue

SYNOPSIS
```
char * edf_report_data_error ( int ErrorValue );
```

DESCRIPTION

Allocates a buffer and copies the error message corresponding to ´ErrorValue´. It returns a pointer to the allocated buffer.

Error Values

NoDataFormatError, UnknownErrorValue, RoutineFailed, RoutineSucceeded, CouldNotMallocMemory, CouldNotFreeHeaders, CouldNotGetBinaryArray, NoMoreStreamsAvailable, CouldNotOpenFile, EndOfFileDetected, CouldNotFindHeader, CouldNotFindSymbol, BadSizeDefinition, BadDataBlock, CouldNotFindKeyword, WriteDataError, ReadDataError, NoStreamOpen, NotESRFDataFile, NoDataBlocksFound, FileIsNotWritable, FileIsNotOpened, CouldNotCloseFile, CouldNotInsertChain, CouldNotInsertBlock, CouldNotInsertSymbol, MissingKeyDefinition, GeneralBlockNotFirst, ErrorCreatingGeneralBlock, ErrorReadingGeneralBlock, ErrorLocatingBlocks, CouldNotSetBuffer, NumberConversionFailed, DataConversionFailed, MissingArrayDimensions, Not2dData, CouldNotWriteDimension, CouldNotReadDimension, CouldNotWriteBinary, CannotReOpenGeneralBlock, CannotOpenAsBslFile

edfio_debug

Set / reset module into debug mode

SYNOPSIS

```
void edfio_debug ( int debug );
```

DESCRPTION

Sets/resets all sub-modules into debug mode

HISTORY

02-Jan-1996 Peter Boesecke
17-Mar-1998 Peter Boesecke V1.00
19-Mar-1998 Peter Boesecke locate_blocks
20-Mar-1998 Peter Boesecke search_general
21-Mar-1998 file->Buffer = (char *) NULL; in init_file
22-Mar-1998
28-Mar-1998 block->BinaryFilePos only used together with BinaryFileName
17-May-1998 PB V1.002 indexing error in reorder_raster and raster_normalization
corrected
17-May-1998 PB V1.003 open of old files read-only with mode="read"
18-May-1998 PB V1.004 open_data_file and open_as_bsl_file: opening mode must be
either Old or Read
01-Jun-1998 PB V1.01 set_bsl_input_byteorder
20-Jun-1998 PB V1.011 locate_blocks (corrected): if in V1x V1_IMAGE_KEY is
missing, the header id number is used as image number.
22-Jul-1998 PB V1.012 read_header_value: debug info added (keyword, value)
30-Aug-1998 PB V1.013 numbers string: closing '\0' added (was missing)
18-Apr-1999 PB V1.02 all public routines start with edf_ or edfio_
18-Apr-1999 PB V1.021 Return value of edf_read_data_2d_dimensionis only FALSE if
open_read_block terminated withthe error value: CouldNotFindHeader.
30-Apr-1999 PB V1.022 edf_read_data_2d_raw
16-Jun-1999 PB V1.023 write_header_value debug info added
23-Jun-1999 PB V1.024 Restrictions split into PUBLIC and PRIVATE
24-Jun-1999 PB V1.03 Module history added, restrictions again only PRIVATE
26-Jun-1999 PB V1.04 History_Line renamed to Header_Line, now public
edf_read_header, edf_write_header
17-Jul-1999 PB V1.05 edf_history_argv: strlen(argument)+3 (instead +2)

17-Jul-1999 PB V1.06 MaxLinLen increased from 127 to 255 in SaxsInput.h:
InputLineLength increased to 256
23-Jul-1999 PB V1.07 DataValueOffset (EDF_DataFormatVersion = 2.20) "DataKey_"
changed to "DataKey-"
26-Jul-1999 PB V1.08 Block boundary warning only given if binary lenghtis different
from zero, warning on BlockBoundaryKey missing removed, useinternal
default (512 bytes) without notice
08-Nov-1999 PB V1.09 edf_raster_normalization added
08-Nov-1999 PB V1.10 in HSymb: shortlen and required added, new functions:
edf_history_skip(), edf_history_take()
26-Nov-1999 PB V1.11 for cc on dec alpha: - some length values were not defined as
constant and were changed to const.- all statements of the type
*ps++=tolower(*ps); changed to { *ps=tolower(*ps); ps++; }
19-Dec-1999 PB V1.12 reorder_raster corrected, Data2FloatIEEE32 bad size check also
for output data
07-Jan-2000 PB V1.13 for cc on dec alpha: all calls of the form
"c=(char)ic=fgetc(channel);" changed to {ic=fgetc(channel); c=(char)ic;}
otherwise ic == c and EOF cannot be checked;
16-Mar-2000 PB V1.14 Comments revised
17-Mar-2000 PB V1.15 ChainKey generally defined as "Image.<ChainNum>"
18-Mar-2000 PB V1.16 DEFAULT_CHAIN_KEY changed to CLASS_KEY, DBClass
and DBInstance
19-Mar-2000 PB V1.17 locate_block: using SequenceNumber, if IDs are missing