OPERA

Operational Programme for the Exchange of Weather Radar Information

FM94-BUFR Encoding and Decoding Software Library

API Documentation

Version 1.0

For BUFR Software Version 3.0

by

Helmut Paulitsch, Juergen Fuchsberger

December 2007

Table of Contents

OPERA BUFR software Module Index

OPERA BUFR software Modules

Here is a list of all modules:

OPERA BUFR software Data Structure Index

OPERA BUFR software Data Structures

Here are the data structures with brief descriptions:

OPERA BUFR software File Index

OPERA BUFR software File List

Here is a list of all documented files with brief descriptions:

OPERA BUFR software Page Index

OPERA BUFR software Related Pages

Here is a list of all related documentation pages:

OPERA BUFR software Module Documentation

Basic functions for encoding to BUFR

Basic functions for encoding to BUFRFunctions

- int bufr_encode_sections34 (dd *descs, int ndescs, varfl *vals, bufr_t *msg)
  Creates section 3 and 4 of BUFR message from arrays of data and data descriptors.
- int bufr_encode_sections0125 (sect_1_t *s1, bufr_t *msg)
  This function creates sections 0, 1, 2 and 5.
- int bufr_write_file (bufr_t *msg, char *file)
  This functions saves the encoded BUFR-message to a binary file.

Function Documentation

int bufr_encode_sections34 (dd * *descs*, int *ndescs*, varfl * *vals*, bufr_t * *msg*)

This function codes data from an array data descriptors `descs` and an array of varfl-values `vals` to a data section and a data descripor section of a BUFR message. Memory for both sections is allocated in this function and must be freed by the calling functions.

Parameters:

descs Data-descriptors corresponding to `vals` . For each descriptor there must be a data-vaule stored in `vals` . `descs` may also include replication factors and sequence descriptors. In that case there must be a larger number of `vals` then of `descs` .
ndescs Number of data descriptos contained in `descs` .
vals Data-values to be coded in the data section. For each entry in `descs` there must be an entry in `vals` . If there are relication factors in `descs` , of course there must be as much `vals` as definded by the replication factor.
msg The BUFR message where to store the coded descriptor and data sections. The memory-area for both sections is allocated by this function and must be freed by the calling function using bufr_free_data .

Returns:

The return-value is 1 if data was successfully stored, 0 if not.

See also:

bufr_encode_sections0125, bufr_data_from_file, bufr_read_msg

Examlples:

apisample.c

int bufr_encode_sections0125 (sect_1_t * *s1*, bufr_t * *msg*)

This function creates sections 0, 1, 2 and 5 of a BUFR message. Memory for this section is allocated by this function and must be freed by the calling function using bufr_free_data .

The total length of the message is calculeted out of the single section length, thus sections 3 and 4 must already be present in the bufr message when calling this function. The BUFR edition is wrote into section 0 and is taken from the global _bufr_edition parameter.

If section 1 data and time parameters are set to 999 (no value), the current system time is taken for coding date and time information.

Parameters:

s1 sect_1_t structure containing section 1 data
msg BUFR message where the sections are to be stored. Must already contain section 3 and 4.

Returns:

1 on success, 0 on error.

Examlples:

apisample.c

int bufr_write_file (bufr_t * *msg*, char * *file*)

This function takes the encoded BUFR message and writes it to a binary file.

Parameters:

msg The complete BUFR message
file The filename of the destination file

Returns:

1 on success, 0 on error

See also:

bufr_read_file

Examlples:

apisample.c

Basic functions for decoding from BUFR

Basic functions for decoding from BUFRFunctions

- int bufr_read_file (bufr_t *msg, char *file)
  This functions reads the encoded BUFR-message to a binary file.
- int bufr_get_sections (char *bm, int len, bufr_t *msg)
  Calculates the section length of a BUFR message and allocates memory for each section.
- int bufr_decode_sections01 (sect_1_t *s1, bufr_t *msg)
  This function decodes sections 0 and 1.
- int bufr_read_msg (void *datasec, void *ddsec, size_t datasecl, size_t ddescl, dd **descr, int *ndescs, varfl **vals, size_t *nvals)
  Decode BUFR data and descriptor section and write values and descriptors to arrays.

Function Documentation

int bufr_read_file (bufr_t * *msg*, char * *file*)

This function reads the encoded BUFR message from a binary file, calculates the section length and writes each section to a memory block. Memory for the sections is allocated by this function and must be freed by the calling function using bufr_free_data .

Parameters:

msg The complete BUFR message
file The filename of the binary file

Returns:

1 on success, 0 on error

See also:

bufr_write_file

int bufr_get_sections (char * *bm*, int *len*, bufr_t * *msg*)

This function calculates the sections length of a BUFR message and allocates memory for each section. The memory has to be freed by the calling function using bufr_free_data .

Parameters:

bm Pointer to the memory where the raw BUFR message is stored
len Length of `bm`
msg The BUFR message containing the single sections and section length

Returns:

Returns the length of the complete BUFR message or 0 on error.

See also:

bufr_free_data, bufr_read_file

int bufr_decode_sections01 (sect_1_t * *s1*, bufr_t * *msg*)

This function decodes sections 0 and 1 of a BUFR message. The BUFR edition is read from section 0 and is written to the global _bufr_edition parameter.

Parameters:

s1 sect_1_t structure to contain section 1 data
msg BUFR message where the sections are stored.

Returns:

1 on success, 0 on error.

Examlples:

apisample.c

int bufr_read_msg (void * *datasec*, void * *ddsec*, size_t *datasecl*, size_t *ddescl*, dd ** *descr*, int * *ndescs*, varfl ** *vals*, size_t * *nvals*)

This function decodes the data and descriptor sections of a BUFR message and stored them into arrays `descr` and `vals` . Memory for storing descriptor- and data-array is allocated by this function and has to be freed by the calling function.

Parameters:

datasec Is where the data-section is stored.
ddsec Is where the data-descriptor-section is stored.
datasecl Number of bytes of the data-section.
ddescl Number of bytes of the data-descriptor-section.
descr Array where the data-descriptors are stored after reading them from the data-descriptor section. This memory area is allocated by this function and has to be freed by the calling function.
ndescs Number of data-descriptors in `descs`
vals Array where the data corresponding to the data-descriptors is stored.
nvals Number of values in `vals`

Returns:

1 if both sections were decoded successfuly, 0 on error

See also:

bufr_create_msg, bufr_data_to_file

Extended functions for encoding to BUFR

Extended functions for encoding to BUFRFunctions

- int bufr_out_descsec (dd *descp, int ndescs, int desch)
  Write descriptor section of a BUFR message to the bitsream.
- int bufr_open_descsec_w ()
  Open bitstream for section 3 for writing and set default values.

- void bufr_close_descsec_w (bufr_t *bufr, int desch)
  Write length of section 3 and close bitstream.
- int bufr_parse_in (dd *descs, int start, int end, int(*inputfkt)(varfl *val, int ind), int callback_descs)
  Parse data descriptors and call user defined input function for each element or for each descriptor.
- void bufr_sect_1_from_file (sect_1_t *s1, char *file)
  Reads section 1 from a file and stores data read in s1.

Function Documentation

int bufr_out_descsec (dd * *descp*, int *ndescs*, int *desch*)

This function writes the descriptor section of a BUFR message to the section 3 bitstream which has already been opened using bufr_open_descsec_w

Parameters:

descp Array holding the data descriptors
ndescs Number of descriptors
desch Handle to the bitstream

Returns:

1 on success, 0 on error

See also:

bufr_open_descsec_w, bufr_out_descsec

int bufr_open_descsec_w ()

This function opens the bitstream for section 3 and sets default values. The bistream must be closed using bufr_close_descsec_w .

Returns:

Returns handle for the bitstream or -1 on error.

See also:

bufr_close_descsec_w, bufr_out_descsec

void bufr_close_descsec_w (bufr_t * *bufr*, int *desch*)

This function calculates and writes the length of section 3, then closes the bitstream.

Parameters:

bufr BUFR message to hold the section.
desch Handle to the bitstream

See also:

bufr_open_descsec_w, bufr_out_descsec

int bufr_parse_in (dd * *descs*, int *start*, int *end*, int(*)(varfl *val, int ind) *inputfkt*, int *callback_descs*)

This function, derived from bufr_parse_new , parses a descriptor or a sequence of descriptors and calls the user defined function `inputfkt` for reading each data-value corresponding to an element descriptor. In case of CCITT (ASCII) data it calls the user-function for each character of the string.

Data values are wrote out to the global data section bitstream (see bufr_open_datasect_w).

Optionally `inputfkt` is called also for sequence descriptors and ccitt descriptors

Parameters:

descs Pointer to the data-descriptors.
start First data-descriptor for output.

end Last data-descriptor for output.
inputfkt User defined input function to be called for each data-element or descriptor
callback_descs Flag that indictes when the user-functions are to be called:
0 for normal behaviour (call `inputfkt` for each element descriptor and each CCITT character)
1 for extended behaviour (call `inputfkt` also for sequence descriptors and CCITT descriptors)

Returns:

The function returns 1 on success, 0 on error

See also:

bufr_parse, bufr_parse_new, bufr_parse_in, Callback functions for encoding to BUFR , bufr_open_datasect_w

void bufr_sect_1_from_file (sect_1_t * *s1*, char * *file*)

This function reads section 1 from an ASCII file and stores the data read in a structure `s1` . If the file can not be read, `s1` is filled with internally defined default values.

Parameters:

s1 Structure where section 1 data is stored.
file Filename of the input file.

See also:

bufr_sect_1_to_file

Extended functions for decoding from BUFR

Extended functions for decoding from BUFRFunctions

- int bufr_parse_out (dd *descs, int start, int end, int(*outputfkt)(varfl val, int ind), int callback_all_descs)
  Parse data descriptors and call user defined output function for each element or for each descriptor.
- int bufr_sect_1_to_file (sect_1_t *s1, char *file)
  Writes section 1 data to an ASCII file.
- int bufr_in_descsec (dd **descs, int ndescs, int desch)
  Read descriptor section of a BUFR message from the bitsream.
- int bufr_open_descsec_r (bufr_t *msg)
  Open bitstream of section 3 for reading.
- void bufr_close_descsec_r (int desch)
  close bitstream for section 3
- int bufr_get_ndescs (bufr_t *msg)
  Calculate number of data descriptors in a BUFR message.

Function Documentation

int bufr_parse_out (dd * *descs*, int *start*, int *end*, int(*)(varfl val, int ind) *outputfkt*, int *callback_all_descs*)

This function, derived from bufr_parse_new , parses a descriptor or a sequence of descriptors and calls the user defined function `outputfkt` for each data-value corresponding to an element descriptor. In case of CCITT (ASCII) data it calls the user-function for each character of the string.

Data values are read from the global data section bitstream (see bufr_open_datasect_r).

Optionally `outputfkt` is called for all descriptors including sequence descriptors, repetition descriptors, ...

Parameters:

descs Pointer to the data-descriptors.
start First data-descriptor for output.
end Last data-descriptor for output.
outputfkt User defined output function to be called for each data-element or descriptor
callback_all_descs Flag that indictes when the user-functions are to be called:
0 for normal behaviour (call `outputfkt` for each element descriptor and each CCITT character)
1 for extended behaviour (call `outputfkt` for all descriptors)

Returns:

The function returns 1 on success, 0 on error

See also:

bufr_parse, bufr_parse_new, bufr_parse_in, Callback functions for decoding from BUFR , bufr_open_datasect_r

Examlples:

apisample.c

int bufr_sect_1_to_file (sect_1_t * *s1*, char * *file*)

This function writes section 1 data to an ASCII file

Parameters:

s1 Structure where section 1 data is stored.
file Filename of the output file.

See also:

bufr_sect_1_from_file

Examlples:

apisample.c

int bufr_in_descsec (dd ** *descs*, int *ndescs*, int *desch*)

This function reads the descriptor section of a BUFR message from the bitsream which was opened using bufr_open_descsec_r

Parameters:

descs Array to hold the data descriptors
ndescs Number of descriptors
desch Handle to the bitstream

Returns:

1 on success, 0 on error

See also:

bufr_get_ndescs, bufr_open_descsec_r, bufr_out_descsec

Examlples:

apisample.c

int bufr_open_descsec_r (bufr_t * *msg*)

This function opens a bitstream for reading of section 3. It must be closed by bufr_close_descsec_r .

Parameters:

msg The encoded BUFR message

Returns:

Returns handle to the bitstream or -1 on error

See also:

bufr_close_descsec_r, bufr_in_descsec

Examlples:

apisample.c

void bufr_close_descsec_r (int *desch*)

This functin closes the input bitstream of section 3 which was opened by bufr_open_descsec_r .

Parameters:

desch Handle to the bitstream

See also:

bufr_open_descsec_r, bufr_in_descsec

Examlples:

apisample.c

int bufr_get_ndescs (bufr_t * *msg*)

This function calculates the number of data descriptors in a BUFR message.

Parameters:

msg The complete BUFR message

Returns:

Returns the number of data descriptors.

See also:

bufr_in_descsec

Examlples:

apisample.c

BUFR utility functions

BUFR utility functionsFunctions

- int bufr_parse_new (dd *descs, int start, int end, int(*inputfkt)(varfl *val, int ind), int(*outputfkt)(varfl val, int ind), int callback_all_descs)
  Parse data descriptors and call user defined functions for each data element or for each descriptor.
- int bufr_parse (dd *descs, int start, int end, varfl *vals, unsigned *vali, int(*userfkt)(varfl val, int ind))
  Parse data descriptors and call user-function for each element.
- void bufr_free_data (bufr_t *msg)
  Frees memory allocated for a BUFR message.
- int bufr_check_fxy (dd *d, int ff, int xx, int yy)
  Tests equality of descriptor d with (f,x,y).
- int bufr_val_to_array (varfl **vals, varfl v, int *nv)
  Store a value to an array of floats.
- int bufr_desc_to_array (dd *descs, dd d, int *ndescs)
  Store a descriptor to an array.
- void bufr_get_date_time (long *year, long *mon, long *day, long *hour, long *min)
  Recall date/time info of the last BUFR-message created.

Function Documentation

int bufr_parse_new (dd * *descs*, int *start*, int *end*, int(*)(varfl *val, int ind) *inputfkt*, int(*)(varfl val, int ind) *outputfkt*, int *callback_all_descs*)

This function, a more general version of bufr_parse , parses a descriptor or a sequence of descriptors and calls the user defined functions `inputfkt` and `outputfkt` for each data-value

corresponding to an element descriptor. In case of CCITT (ASCII) data it calls the user-functions for each character of the string.

Data values are read using the user-defined function `inputfkt` and wrote out using `outputfkt` .

Optionally the user-defined functions are called for all descriptors, including sequence descriptors and data modification descriptors.

Parameters:

descs Pointer to the data-descriptors.
start First data-descriptor for output.
end Last data-descriptor for output.
inputfkt User defined input function to be called for each data-element or descriptor
outputfkt User defined ouput function to be called for each data-element or descriptor
callback_all_descs Flag that indictes when the user-functions are to be called:
0 for normal behaviour (call user-functions for each element descriptor and each CCITT character)
1 for extended behaviour (call both user-functions also for sequence descriptors and CCITT descriptors,
call `outputfkt` also for replication descriptors and data modification descriptors.)

Returns:

The function returns 1 on success, 0 on error.

See also:

bufr_parse, bufr_parse_in, bufr_parse_out, Callback functions for encoding to BUFR , Callback functions for decoding from BUFR

int bufr_parse (dd * *descs*, int *start*, int *end*, varfl * *vals*, unsigned * *vali*, int(*)(varfl val, int ind) *userfkt*)

This function parses a descriptor or a sequence of descriptors and calls the user defined function `userfkt` for each data-value corresponding to an element descriptor. In case of CCITT (ASCII) data it calls `userfkt` for each character of the string.

Data values are read from an array of floats stored at `vals` .

Parameters:

descs Pointer to the data-descriptors.
start First data-descriptor for output.
end Last data-descriptor for output.
vals Pointer to an array of values.
vali Index for the array `vals` that identifies the values to be used for output. `vali` is increased after data-output.
userfkt User-function to be called for each data-element

Returns:

The function returns 1 on success, 0 if there was an error outputing to the bitstreams.

void bufr_free_data (bufr_t * *msg*)

This function frees all memory allocated for a BUFR message by bufr_data_from_file , bufr_encode_sections0125 , bufr_read_file or bufr_get_sections .

Parameters:

msg The encoded BUFR message

Examlples:

apisample.c

int bufr_check_fxy (dd * *d*, int *ff*, int *xx*, int *yy*)

This functions tests wheter a descriptor equals the given values f, x, y

Parameters:

d The descriptor to be tested

ff,xx,yy The values for testing

Return values:

1 If the descriptor equals the given values
0 If the descriptor is different to the given values

Examlples:

apisample.c

int bufr_val_to_array (varfl ** *vals*, varfl *v*, int * *nv*)

This function stores the value `v` to an array of floats `vals` . The memory-block for `vals` is allocated in this function and has to be freed by the calling function. The number of values is used to calculate the size of the array and reallocate memory if necessary.

Parameters:

vals The array containing the values
v The value to be put into the array
nv Current number of values in the array

Returns:

1 on success, 0 on error.

int bufr_desc_to_array (dd * *descs*, dd *d*, int * *ndescs*)

This function stores the descriptor `d` to an array of descriptors `descs` . The array descs must be large enough to hold `ndescs` + 1 descriptors.

Parameters:

descs The array containing the descriptors
d The descriptor to be put into the array
ndescs Current number of descriptors in the array

Returns:

1 on success, 0 on error.

void bufr_get_date_time (long * *year*, long * *mon*, long * *day*, long * *hour*, long * *min*)

This function can be called to recall the data/time-info of the last BUFR-message created, if the appropiate data descriptors have been used.

Parameters:

year 4 digit year if _bufr_edition is set to 4, year of century (2 digit) if _bufr_edition is < 4.
mon Month (1 - 12)
day (1 - 31)
hour
min

Examlples:

apisample.c

Functions for data descriptor management

Functions for data descriptor managementFunctions

- int read_tables (char *dir, int vmtab, int vltab, int subcent, int gencent)
  Reads bufr tables from csv-files.
- void show_desc (int f, int x, int y)
  Prints the specified descriptor or all if f = 999.
- int get_index (int typ, dd *descr)
  Returns the index for the given descriptor and typ.

- int read_tab_d (char *fname)
  Reads bufr table d from a csv-files.
- int read_tab_b (char *fname)
  Reads bufr table b from a csv-files.
- void free_descs (void)
  Frees all memory that has been allocated for data descriptors.
- char * get_unit (dd *d)
  Returns the unit for a given data descriptor.

Function Documentation

int read_tables (char * *dir*, int *vmtab*, int *vltab*, int *subcent*, int *gencent*)

This function reads the descriptor tables from csv-files and stores the descriptors in a global array des . Memory for the descriptors is allocated by this function and has to be freed using free_descs .

The filenames are generated by this function and have the form bufrtab{b|d}_Y.csv or loctab{b|d}_X_Y.csv where X is a value calculated of the originating center and subcenter. (X = `subcent` * 256 + `gencent`) Y is the table version.

Parameters:

dir The directory where to search for tables, if NULL the function uses the current directory
vmtab Master table version number
vltab Local table version number.
subcent Originating/generating subcenter
gencent Originating/generating center

Returns:

Returns 0 on success or -1 on errors.

Note:

The local tables are optional

Examlples:

apisample.c

void show_desc (int *f*, int *x*, int *y*)

This function prints all information on the specified descriptor or all descriptors if f = 999

Parameters:

f,x,y The descriptor to display.

int get_index (int *typ*, dd * *descr*)

This function returns the index into the global des array of a descriptor given by parameters `typ` and `descr` .

Parameters:

typ The type of descriptor (ELDESC or SEQDESC).
descr The descriptor.

Returns:

The index of the descriptor in des or -1 on error.

int read_tab_d (char * *fname*)

This function reads a sequence descriptor table (d) from a csv-file and stores the descriptors in a global array des . Memory for the descriptors is allocated by this function and has to be freed using free_descs .

Parameters:

fname The name of a csv-file.

Returns:

Returns 1 on success or 0 on error.

See also:

read_tables, read_tab_b

int read_tab_b (char * *fname*)

This function reads an element descriptor table (b) from a csv-file and stores the descriptors in a global array des . Memory for the descriptors is allocated by this function and has to be freed using free_descs .

Parameters:

fname The name of the csv-file.

Returns:

Returns 1 on success or 0 on error.

See also:

read_tables, read_tab_d

void free_descs (void)

This function frees all memory that has been allocated for data descriptors

See also:

read_tables, read_tab_b, read_tab_d

Examlples:

apisample.c

char* get_unit (dd * *d*)

This function searches the global des array and returns the unit for a data descriptor.

Parameters:

d The descriptor.

Returns:

Pointer to a string containing the unit or NULL if the descriptor is not found in the global des array.

Functions for run length encoding

Functions for run length encodingFunctions

- int rlenc_from_file (char *infile, int nrows, int ncols, varfl **vals, int *nvals, int depth)
  Runlength-encodes a radar image from a file to an array.
- int rlenc_from_mem (unsigned short *img, int nrows, int ncols, varfl **vals, int *nvals)
  This function encodes a radar image to BUFR runlength-code.
- int rlenc_compress_line_new (int line, unsigned int *src, int ncols, varfl **dvals, int *nvals)
  Encodes one line of a radar image to BUFR runlength-code.

Function Documentation

int rlenc_from_file (char * *infile*, int *nrows*, int *ncols*, varfl ** *vals*, int * *nvals*, int *depth*)

This function encodes a radar image file with `depth` bytes per pixel to BUFR runlength-code and stores the resulting values into an array `vals` by a call to bufr_val_to_array .

Currently depth can be one or two bytes per pixel. In case of two bytes per pixel data is read in "High byte - low byte order". So pixel values 256 257 32000 are represented by 0100 0101 7D00 hex.

Note:

In difference to the old rlenc function the initial length of `vals` must be given in the parameter `nvals` in order to prevent bufr_val_to_array from writing to an arbitrary position.

Parameters:

infile File holding the radar image.
ncols Number of columns of the image.
nrows Number of rows of the image.
depth Image depth in bytes
vals Float-array holding the coded image.
nvals Number of values in VALS.

Returns:

The return-value ist 1 on success, 0 on a fault.

See also:

rlenc_from_mem, rldec_to_file, rlenc_compress_line_new

int rlenc_from_mem (unsigned short * *img*, int *nrows*, int *ncols*, varfl ** *vals*, int * *nvals*)

This function encodes a radar image in memory to BUFR runlength-code and stores the resulting values into an array `vals` by a call to bufr_val_to_array .

Note:

In difference to the old rlenc function the initial length of `vals` must given in the parameter `nvals` in order to prevent bufr_val_to_array from writing to an arbitrary position.

Parameters:

img Array holding the uncompressed radar image.
ncols Number of columns of the image.
nrows Number of rows of the image.
vals Float-array holding the coded image.
nvals Number of values in `vals` .

Returns:

The return-value ist 1 on success, 0 on a fault.

See also:

rlenc_from_file, rldec_to_mem, rlenc_compress_line_new

Examlples:

apisample.c

int rlenc_compress_line_new (int *line*, unsigned int * *src*, int *ncols*, varfl ** *dvals*, int * *nvals*)

This function encodes one line of a radar image to BUFR runlength-code and stores the resulting values to array `dvals` by a call to bufr_val_to_array .

Note:

In difference to the old rlenc_compress_line function the initial length of `vals` must given in the parameter `nvals` in order to prevent bufr_val_to_array from writing to an arbitrary position.

Parameters:

line Line number.
src Is where the uncompressed line is stored.

ncols Number of pixels per line.
dvals Float-array holding the coded image.
nvals Number of values in VALS.

Returns:

The function returns 1 on success, 0 on a fault.

See also:

rldec_decompress_line

Functions for run length decoding

Functions for run length decodingFunctions

- int rldec_to_file (char *outfile, varfl *vals, int depth, int *nvals)
  Decodes a BUFR-runlength-encoded radar image to a file.
- int rldec_to_mem (varfl *vals, unsigned short **img, int *nvals, int *nrows, int *ncols)
  Decodes a BUFR-runlength-encoded radar image to memory.
- void rldec_decompress_line (varfl *vals, unsigned int *dest, int *ncols, int *nvals)
  Decodes one line of a radar image from BUFR runlength-code.
- void rldec_get_size (varfl *vals, int *nrows, int *ncols)
  Gets the number of rows and columns of a runlength compressed image.

Function Documentation

int rldec_to_file (char * *outfile*, varfl * *vals*, int *depth*, int * *nvals*)

This function decodes a BUFR-runlength-encoded radar image stored at `vals` . The decoded image is stored in a "depth byte-per-pixel-format" at the file `outfile` .

Currently depth can be one or two bytes per pixel. In case of two bytes per pixel data is stored in "High byte - low byte order". So pixel values 256 257 32000 are represented by 0100 0101 7D00 hex.

Parameters:

outfile Destination-file for the radar image.
vals Float-array holding the coded image.
depth Number of bytes per pixel
nvals Number of varfl values needed for the compressed radar image.

Returns:

The return-value ist 1 on success, 0 on a fault.

See also:

rldec_to_mem, rldec_decompress_line, rlenc_from_file

int rldec_to_mem (varfl * *vals*, unsigned short ** *img*, int * *nvals*, int * *nrows*, int * *ncols*)

This function decodes a BUFR-runlength-encoded radar image stored at `vals` . The decoded image is stored in an array `img` [] which will be allocated by this function if `img` [] = NULL. The memory for the image must be freed by the calling function!

Parameters:

vals Float-array holding the coded image.
img Destination-array for the radar image.
nvals Number of varfl values needed for the compressed radar image.

nrows Number of lines in image
ncols Number of pixels per line

Returns:

The return-value ist 1 on success, 0 on a fault.

See also:

rlenc_from_mem, rldec_to_file, rldec_decompress_line

Examlples:

apisample.c

void rldec_decompress_line (varfl * *vals*, unsigned int * *dest*, int * *ncols*, int * *nvals*)

This function decodes one line of a radar image from BUFR runlength-code and stores the resulting values to array `dest` which has to be large enough to hold a line.

Parameters:

vals Float-array holding the coded image.
dest Is where the uncompressed line is stored.
ncols Number of pixels per line.
nvals Number of values needed for compressed line.

See also:

rlenc_compress_line_new

void rldec_get_size (varfl * *vals*, int * *nrows*, int * *ncols*)

This function gets the number of rows and colums of a runlength compressed image stored at array `vals`

Parameters:

vals Float-array holding the coded image.
nrows Number of lines in image.
ncols Number of pixels per line.

See also:

rldec_to_file, rldec_decompress_line

Functions for encoding/decoding from/to OPERA ASCII Files

Functions for encoding/decoding from/to OPERA ASCII FilesFunctions

- int bufr_data_from_file (char *file, bufr_t *msg)
  read data and descriptors from ASCII file and code them into sections 3 and 4
- int bufr_data_to_file (char *file, char *imgfile, bufr_t *msg)
  Decode data and descriptor sections of a BUFR message and write them to an ASCII file.

Function Documentation

int bufr_data_from_file (char * *file*, bufr_t * *msg*)

This function reads descriptors and data from an ASCII file and codes them into a BUFR data descriptor and data section (section 3 and 4). Memory for both sections is allocated in this function and must be freed by the calling functions using bufr_free_data .

Parameters:

file Name of the input ASCII file
msg BUFR message to contain the coded sections

Returns:

1 on succes, 0 on error

See also:

bufr_data_to_file, bufr_create_msg, bufr_free_data

int bufr_data_to_file (char * *file*, char * *imgfile*, bufr_t * *msg*)

This functions decodes data and descriptor sections of a BUFR message and writes them into an ASCII file. If there is an OPERA bitmap (currently descriptors 3 21 192 to 3 21 197, 3 21 200 and 3 21 202) it is written to a seperate file.

Parameters:

file Name of the output ASCII file
imgfile Name of the output bitmap file(s)
msg BUFR message to contain the coded sections

Returns:

1 on succes, 0 on error

See also:

bufr_data_from_file, bufr_read_msg

Examlples:

apisample.c

Callback functions for encoding to BUFR

Callback functions for encoding to BUFRFunctions

- int bufr_val_from_global (varfl *val, int ind)
  Get one value from global array of values.

Function Documentation

int bufr_val_from_global (varfl * *val*, int *ind*)

This functions gets the next value from the global array of values.

Parameters:

val The received value
ind Index to the global array des [] holding the description of known data-descriptors.

Returns:

1 on success, 0 on error.

See also:

bufr_open_val_array, bufr_close_val_array

Callback functions for decoding from BUFR

Callback functions for decoding from BUFRFunctions

- int bufr_val_to_global (varfl val, int ind)
  Write one value to global array of values.

Function Documentation

int bufr_val_to_global (varfl *val*, int *ind*)

This functions writes one value to the global array of values.

Parameters:

val The value to store
ind Index to the global array des [] holding the description of known data-descriptors.

Returns:

1 on success, 0 on error.

See also:

bufr_open_val_array, bufr_close_val_array

Examlples:

apisample.c

Utilities for encoding callback functions

Utilities for encoding callback functionsFunctions

- int bufr_open_datasect_w ()
  Opens bitstream for section 4 writing.
- void bufr_close_datasect_w (bufr_t *msg)
  Closes bitstream for section 4 and adds data to BUFR message.
- bufrval_t * bufr_open_val_array ()
  Opens global array of values for read/write.
- void bufr_close_val_array ()
  Closes global array of values and frees all memory.

Function Documentation

int bufr_open_datasect_w ()

This function opens the data section bitstream for writing and returns its handle.

Returns:

Returns the handle to the data section bitstream or -1 on error.

See also:

bufr_close_datasect_w, bufr_parse_in

void bufr_close_datasect_w (bufr_t * *msg*)

This function closes the data section bitstream and appends it to a BUFR message, also stores the length in the BUFR message.

Parameters:

msg BUFR message where the data has to be stored

See also:

bufr_open_datasect_w, bufr_parse_in

bufrval_t* bufr_open_val_array ()

This function opens the global array of values for use by bufr_val_from_global and bufr_val_to_global and returns its pointer.

Returns:

Pointer to the array of values or NULL on error.

See also:

bufr_close_val_array, bufr_val_to_global, # bufr_val_from_global

Examlples:

apisample.c

void bufr_close_val_array ()

This function closes the global array of values used by bufr_val_from_global and bufr_val_to_global and frees all allocated memory.

See also:

bufr_open_val_array, bufr_val_to_global, bufr_val_from_global

Examlples:

apisample.c

Utilities for decoding callback functions

Utilities for decoding callback functionsFunctions

- int bufr_open_datasect_r (bufr_t *msg)
  Opens bitstream for reading section 4.
- void bufr_close_datasect_r ()
  Closes bitstream for section 4.

Function Documentation

int bufr_open_datasect_r (bufr_t * *msg*)

This function opens the data section bitstream at for reading and returns its handle.

Parameters:

msg The BUFR message containing the data section.

Returns:

Returns the handle to the data section bitstream or -1 on error.

See also:

bufr_close_datasect_r, bufr_parse_out

Examlples:

apisample.c

void bufr_close_datasect_r ()

This function closes the data section bitstream.

See also:

bufr_open_datasect_r, bufr_parse_out

Examlples:

apisample.c

Functions for input and output to/from a bitstream

Functions for input and output to/from a bitstreamFunctions

- int bitio_i_open (void *buf, size_t size)
  This function opens a bitstream for input.
- int bitio_i_input (int handle, unsigned long *val, int nbits)
  This function reads a value from a bitstream.
- void bitio_i_close (int handle)
  Closes an bitstream that was opened for input.
- int bitio_o_open ()
  Opens a bitstream for output.
- long bitio_o_append (int handle, unsigned long val, int nbits)
  This function appends a value to a bitstream.
- void bitio_o_outp (int handle, unsigned long val, int nbits, long bitpos)
  This function outputs a value to a specified position of a bitstream.
- size_t bitio_o_get_size (int handle)
  Returns the size of an output-bitstream (number of bytes).
- void * bitio_o_close (int handle, size_t *nbytes)
  This function closes an output-bitstream.

Function Documentation

int bitio_i_open (void * *buf*, size_t *size*)

This function opens a bitstream for input.

Parameters:

buf Buffer to be used for input
size Size of buffer.

Returns:

the function returns a handle by which the bitstream can be identified for all subsequent actions or -1 if the maximum number of opened bitstreams exceeds.

See also:

bitio_i_close, bitio_i_input, bitio_o_open

int bitio_i_input (int *handle*, unsigned long * *val*, int *nbits*)

This function reads a value from a bitstream. The bitstream must have been opened by bitio_i_open .

Parameters:

handle Identifies the bitstream.
val Is where the input-value is stored.
nbits Number of bits the value consists of.

Returns:

Returns 1 on success or 0 on a fault (number of bytes in the bitstream exceeded).

See also:

bitio_i_open, bitio_i_close, bitio_o_outp

void bitio_i_close (int *handle*)

Closes an bitstream that was opened for input

Parameters:

handle Handle that identifies the bitstream.

See also:

bitio_i_open, bitio_i_input

int bitio_o_open ()

This function opens a bitstream for output.

Returns:

The return-vaule is a handle by which the bit-stream can be identified for all subesquent actions or -1 if there is no unused bitstream available.

long bitio_o_append (int *handle*, unsigned long *val*, int *nbits*)

This function appends a value to a bitstream which was opened by bitio_o_open .

Parameters:

handle Indicates the bitstream for appending.
val Value to be output.
nbits Number of bits of `val` to be output to the stream.

Note:

`nbits` must be less than sizeof (`long`)

Returns:

The return-value is the bit-position of the value in the bit-stream, or -1 on a fault.

See also:

bitio_o_open, bitio_o_close, bitio_o_outp

void bitio_o_outp (int *handle*, unsigned long *val*, int *nbits*, long *bitpos*)

This function outputs a value to a specified position of a bitstream.

Parameters:

handle Indicates the bitstream for output.
val Value to be output.
nbits Number of bits of `val` to be output to the stream.
bitpos bitposition of the value in the bitstream.

Note:

`nbits` must be less then sizeof (`long`)

See also:

bitio_o_open, bitio_o_close, bitio_o_append, bitio_i_input

size_t bitio_o_get_size (int *handle*)

This function returns the size of an output-bitstream (number of bytes)

Parameters:

handle Identifies the bitstream

Returns:

Size of the bitstream.

See also:

bitio_o_open, bitio_o_outp, bitio_o_append

void* bitio_o_close (int *handle*, size_t * *nbytes*)

This function closes an output-bitstream identified by `handle` and returns a pointer to the memory-area holding the bitstream.

Parameters:

handle Bit-stream-handle

nbytes number of bytes in the bitstream.

Returns:

The funcion returns a pointer to the memory-area holding the bit-stream or NULL if an invalid handle was specified. The memory area must be freed by the calling function.

See also:

bitio_o_open, bitio_o_outp, bitio_o_append, bitio_i_close

Deprecated functions

Deprecated functionsFunctions

- void bufr_clean (void)
- int setup_sec0125 (char *sec[], size_t secl[], sect_1_t s1)
- int save_sections (char **sec, size_t *secl, char *buffile)
- int val_to_array (varfl **vals, varfl v, size_t *nvals)
- int rlenc (char *infile, int nrows, int ncols, varfl **vals, size_t *nvals)
  Runlength-encodes a radar image.
- int rlenc_compress_line (int line, unsigned char *src, int ncols, varfl **dvals, size_t *nvals)
  Encodes one line of a radar image to BUFR runlength-code.
- int rldec (char *outfile, varfl *vals, size_t *nvals)
  Decodes a BUFR-runlength-encoded radar image.

Function Documentation

void bufr_clean (void)

Deprecated:

use free_descs instead

This function frees all memory-blocks allocated by read_tables

int setup_sec0125 (char * *sec*[], size_t *secl*[], sect_1_t *s1*)

Deprecated:

use bufr_encode_sections0125 instead

Sets up section 0,1,2,5 in a rather easy fashion and takes Section 1 data from structure s1.

Parameters:

sec Sections 0 - 5
secl Lengths of sections 0 - 5
s1 Data to be put into Section 1

int save_sections (char ** *sec*, size_t * *secl*, char * *buffile*)

Deprecated:

Use bufr_write_file instead.

Write BUFR message to a binary file.

Parameters:

sec Poiter-Array to the 6 sections.
secl Length of the sections.
buffile Output-File

Returns:

The function returns 1 on success, 0 on a fault.

int val_to_array (varfl ** *vals*, varfl *v*, size_t * *nvals*)

Deprecated:
use bufr_val_to_array instead.

This function stores the value V to an array of floats VALS. The memory- block for VALS is allocated in this function and has to be freed by the calling function.

Parameters:

vals The array containing the values
v The value to be put into the array
nvals Number of values in the array

Returns:

1 on success, 0 on error.

int rlenc (char * *infile*, int *nrows*, int *ncols*, varfl ** *vals*, size_t * *nvals*)

Deprecated:
Use rlenc_from_file instead.

This function encodes a "one byte per pixel" radar image to BUFR runlength- code and stores the resulting values by a call to VAL_TO_ARRAY.

Parameters:

infile File holding the "one byte per pixel" radar image.
ncols Number of columns of the image.
nrows Number of rows of the image.
vals Float-array holding the coded image.
nvals Number of values in VALS.

Returns:

The return-value ist 1 on success, 0 on a fault.

int rlenc_compress_line (int *line*, unsigned char * *src*, int *ncols*, varfl ** *dvals*, size_t * *nvals*)

Deprecated:
Use rlenc_compress_line_new instead.

This function encodes one line of a radar image to BUFR runlength-code and stores the resulting values by a call to val_to_array .

Parameters:

line Line number.
src Is where the uncompressed line is stored.
ncols Number of pixels per line.
dvals Float-array holding the coded image.
nvals Number of values in VALS.

Returns:

The function returns 1 on success, 0 on a fault.

int rldec (char * *outfile*, varfl * *vals*, size_t * *nvals*)

Deprecated:
Use rldec_to_file instead.

This function decodes a BUFR-runlength-encoded radar image stored at `VALS` . The decoded image is stored in a one "byte-per-pixel-format" at the file `OUTFILE` .

Parameters:

outfile Destination-file for the "one byte per pixel" radar image.
vals Float-array holding the coded image.
nvals Number of values needed for the radar image.

Returns:

The return-value ist 1 on success, 0 on a fault.

API examples

API examplesFunctions

- void bufr_encoding_sample (radar_data_t *src_data, bufr_t *bufr_msg)
  Sample for encoding a BUFR message.
- void bufr_decoding_sample (bufr_t *msg, radar_data_t *data)
  Sample for decoding a BUFR message.

Function Documentation

void bufr_encoding_sample (radar_data_t * *src_data*, bufr_t * *bufr_msg*)

This function encodes sample data to a BUFR message and saves the results to a file apisample.bfr, also returns the encoded message.

Parameters:

src_data Our source data.
bufr_msg Our encoded BUFR message.

See also:

bufr_decoding_sample

Examlples:

apisample.c

void bufr_decoding_sample (bufr_t * *msg*, radar_data_t * *data*)

This function decodes a BUFR message and stores the values in our sample radar data structure. Also saves the result to a file.

Parameters:

msg Our encoded BUFR message.
data Our source data.

See also:

bufr_encoding_sample

Examlples:

apisample.c

OPERA BUFR software Data Structure Documentation

bufr_t Struct Reference

bufr_tStructure that holds the encoded bufr message.

`#include <bufr.h>`

Data Fields

- char * sec [6]
  pointers to sections

- int secl [6]
  length of sections

Detailed Description

Examlples:

apisample.c

The documentation for this struct was generated from the following file:

- bufr.h

---

bufrval_t Struct Reference

bufrval_tStructure holding values for callbacks bufr_val_from_global and bufr_val_to_global .

`#include <bufr.h>`

Data Fields

- varfl * vals
  array of values
- int vali
  current index into array of values
- int nvals
  number of values

Detailed Description

Examlples:

apisample.c

The documentation for this struct was generated from the following file:

- bufr.h

---

dd Struct Reference

ddDescribes one data descriptor.

`#include <desc.h>`

Data Fields

- int f
  f
- int x

x

- int y
  y

Detailed Description

Examlples:

apisample.c

The documentation for this struct was generated from the following file:

- desc.h

---

del Struct Reference

delDefines an element descriptor.

`#include <desc.h>`

Data Fields

- dd d
  Descriptor ID.
- char * unit
  Unit.
- int scale
  Scale.
- varfl refval
  Reference Value.
- int dw
  Data width (number of bits).
- char * elname
  element name

The documentation for this struct was generated from the following file:

- desc.h

---

desc Struct Reference

descStructure that defines one descriptor. This can be an element descriptor or a sequence descriptor.

`#include <desc.h>`

Data Fields

- int id
  Can be SEQDESC or ELDESC .

- del * el
  Element descriptor.
- dseq * seq
  Sequence descriptor.
- int key
  search key
- int nr
  serial number (insert position)

The documentation for this struct was generated from the following file:

- desc.h

---

dseq Struct Reference

dseqStructure that defines a sequence of descriptors.

`#include <desc.h>`

Data Fields

- dd d
  sequence-descriptor ID
- int nel
  Number of elements.
- dd * del
  list of element descriptors

The documentation for this struct was generated from the following file:

- desc.h

---

sect_1_t Struct Reference

sect_1_tHolds the information contained in section 1.

`#include <desc.h>`

Data Fields

- int mtab
  BUFR master table.
- int subcent
  Originating/generating subcenter.
- int gencent
  Originating/generating center.
- int updsequ
  Update sequence number.
- int opsec

optional section

- int dcat
  Data Category type (BUFR Table A).
- int dcatst
  Data Category sub-type.
- int idcatst
  International Data Category sub-type.
- int vmtab
  Version number of master tables used.
- int vltab
  Version number of local tables used.
- int year
  Year of century.
- int mon
  Month.
- int day
  Day.
- int hour
  Hour.
- int min
  Minute.
- int sec
  Second (used as of BUFR edition 4).

Detailed Description

Holds the information contained in section 1

See also:

bufr_sect_1_from_file, bufr_sect_1_to_file, bufr_encode_sections0125, bufr_decode_sections01

Examlples:

apisample.c

Field Documentation

int sect_1_t::mtab

BUFR master table 0 for standard WMO BUFR tables

Examlples:

apisample.c

int sect_1_t::updsequ

Update sequence number zero for original BUFR messages; incremented for updates

Examlples:

apisample.c

int sect_1_t::opsec

Bit 1 = 0 No optional section = 1 Optional section included Bits 2 - 8 set to zero (reserved)

Examlples:

apisample.c

int sect_1_t::dcatst

Data Category sub-type defined by local ADP centres

Examlples:

apisample.c

int sect_1_t::idcatst

International Data Category sub-type Common Table C-13, used as of BUFR edition 4

int sect_1_t::year

Year of century 2 digit for BUFR edition < 4, 4 digit year as of BUFR edition 4

Examlples:

apisample.c

The documentation for this struct was generated from the following file:

- desc.h

OPERA BUFR software File Documentation

apisample.c File Reference

apisample.cSample application for encoding and decoding BUFR using OPERA BUFR software as a library.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "bufrlib.h"
#include "apisample.h"
#include "bufr_io.h"
```

Functions

- void bufr_encoding_sample (radar_data_t *src_data, bufr_t *bufr_msg)
  Sample for encoding a BUFR message.
- void bufr_decoding_sample (bufr_t *msg, radar_data_t *data)
  Sample for decoding a BUFR message.

Detailed Description

This sample application uses the OPERA BUFR software api for encoding and decoding a sample radar image to/from a BUFR message.

---

bitio.c File Reference

bitio.cfunctions for input and output to/from a bitstream

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <assert.h>
#include <memory.h>
#include "desc.h"
#include "bufr.h"
#include "bitio.h"
```

Functions

- int bitio_i_open (void *buf, size_t size)
  This function opens a bitstream for input.
- int bitio_i_input (int handle, unsigned long *val, int nbits)
  This function reads a value from a bitstream.
- void bitio_i_close (int handle)
  Closes an bitstream that was opened for input.
- int bitio_o_open ()
  Opens a bitstream for output.
- long bitio_o_append (int handle, unsigned long val, int nbits)
  This function appends a value to a bitstream.
- void bitio_o_outp (int handle, unsigned long val, int nbits, long bitpos)
  This function outputs a value to a specified position of a bitstream.
- size_t bitio_o_get_size (int handle)
  Returns the size of an output-bitstream (number of bytes).
- void * bitio_o_close (int handle, size_t *nbytes)
  This function closes an output-bitstream.

Detailed Description

The functions in this file can be used for input and output to/from a bitstream as needed for BUFR-messages. Data is stored on/read from a bitstream as follows: For example if you wan to store a 12 bit-value VAL on a bit-stream, consisting of a character-array C, the bits are assigned (bit 0 is the least segnificant bit).

VAL bit 00 -> C[0] bit 00

VAL bit 01 -> C[0] bit 01

VAL bit 02 -> C[0] bit 02

VAL bit 03 -> C[0] bit 03

VAL bit 04 -> C[0] bit 04

VAL bit 05 -> C[0] bit 05

VAL bit 06 -> C[0] bit 06

VAL bit 07 -> C[1] bit 07

VAL bit 08 -> C[1] bit 00

VAL bit 09 -> C[1] bit 01

VAL bit 10 -> C[1] bit 02

VAL bit 11 -> C[1] bit 03

if you append another 2-bit value VAL1 to the stream:

VAL bit 00 -> C[1] bit 04

VAL bit 01 -> C[1] bit 05

Functions for output of data to a bit-stream are named bitio_o_*, those for inputing from a bitstream bitio_i_*.

Output to a bit-stream must be as follows:

h = bitio_o_open (); open a bitstrem, handle H is returned to identify for subsequent calls.

bitio_o_append (h, val, nbits); Append VAL to the bitstream.

bitio_o_close (h, nbytes); close bitstream.from a bit-stream must be as follows:

h = bitio_i_open (); open a bitstream for input

bitio_i_input (); read a value from the bitstream

bitio_i_close (); close the bitstream

More details can be found at the description of the functions. Note that the buffer holding the bitstream is organized as an array of characters. So the functions are independent from the computer-architecture (byte-swapping).

---

bitio.h File Reference

bitio.hFunction definitions for bitstream input and output.

Functions

- int bitio_i_open (void *buf, size_t size)
  This function opens a bitstream for input.
- int bitio_i_input (int handle, unsigned long *val, int nbits)
  This function reads a value from a bitstream.
- size_t bitio_o_get_size (int handle)
  Returns the size of an output-bitstream (number of bytes).
- void bitio_i_close (int handle)
  Closes an bitstream that was opened for input.
- int bitio_o_open ()
  Opens a bitstream for output.
- long bitio_o_append (int handle, unsigned long val, int nbits)
  This function appends a value to a bitstream.
- void bitio_o_outp (int handle, unsigned long val, int nbits, long bitpos)
  This function outputs a value to a specified position of a bitstream.
- void * bitio_o_close (int handle, size_t *nbytes)
  This function closes an output-bitstream.

Detailed Description

This file defines all functions for input and output to/from a bitstream.

bufr.c File Reference

bufr.cMain OPERA BUFR library functions.

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <time.h>
#include "desc.h"
#include "bufr.h"
#include "bitio.h"
#include "rlenc.h"
```

Functions

- void bufr_clean (void)
- int bufr_create_msg (dd *descs, int ndescs, varfl *vals, void **datasec, void **ddsec, size_t *datasecl, size_t *ddescl)
  Creates section 3 and 4 of BUFR message from arrays of data and data descriptors.
- int bufr_encode_sections34 (dd *descs, int ndescs, varfl *vals, bufr_t *msg)
  Creates section 3 and 4 of BUFR message from arrays of data and data descriptors.
- int bufr_read_file (bufr_t *msg, char *file)
  This functions reads the encoded BUFR-message to a binary file.
- int bufr_get_sections (char *bm, int len, bufr_t *msg)
  Calculates the section length of a BUFR message and allocates memory for each section.
- int bufr_out_descsec (dd *descp, int ndescs, int desch)
  Write descriptor section of a BUFR message to the bitsream.
- int bufr_open_descsec_w ()
  Open bitstream for section 3 for writing and set default values.
- void bufr_close_descsec_w (bufr_t *bufr, int desch)
  Write length of section 3 and close bitstream.
- int setup_sec0125 (char *sec[], size_t secl[], sect_1_t s1)
- int save_sections (char **sec, size_t *secl, char *buffile)
- int bufr_parse_new (dd *descs, int start, int end, int(*inputfkt)(varfl *val, int ind), int(*outputfkt)(varfl val, int ind), int callback_all_descs)
  Parse data descriptors and call user defined functions for each data element or for each descriptor.
- int bufr_parse (dd *descs, int start, int end, varfl *vals, unsigned *vali, int(*userfkt)(varfl val, int ind))
  Parse data descriptors and call user-function for each element.
- int bufr_parse_in (dd *descs, int start, int end, int(*inputfkt)(varfl *val, int ind), int callback_descs)
  Parse data descriptors and call user defined input function for each element or for each descriptor.
- int bufr_parse_out (dd *descs, int start, int end, int(*outputfkt)(varfl val, int ind), int callback_all_descs)
  Parse data descriptors and call user defined output function for each element or for each descriptor.
- void bufr_sect_1_from_file (sect_1_t *s1, char *file)
  Reads section 1 from a file and stores data read in s1.
- int bufr_encode_sections0125 (sect_1_t *s1, bufr_t *msg)
  This function creates sections 0, 1, 2 and 5.

- int bufr_write_file (bufr_t *msg, char *file)
  This functions saves the encoded BUFR-message to a binary file.
- void bufr_free_data (bufr_t *msg)
  Frees memory allocated for a BUFR message.
- int bufr_check_fxy (dd *d, int ff, int xx, int yy)
  Tests equality of descriptor d with (f,x,y).
- int bufr_decode_sections01 (sect_1_t *s1, bufr_t *msg)
  This function decodes sections 0 and 1.
- int bufr_sect_1_to_file (sect_1_t *s1, char *file)
  Writes section 1 data to an ASCII file.
- int bufr_read_msg (void *datasec, void *ddsec, size_t datasecl, size_t ddescl, dd **descr, int *ndescs, varfl **vals, size_t *nvals)
  Decode BUFR data and descriptor section and write values and descriptors to arrays.
- int bufr_in_descsec (dd **descs, int ndescs, int desch)
  Read descriptor section of a BUFR message from the bitsream.
- int bufr_open_descsec_r (bufr_t *msg)
  Open bitstream of section 3 for reading.
- void bufr_close_descsec_r (int desch)
  close bitstream for section 3
- int val_to_array (varfl **vals, varfl v, size_t *nvals)
- int bufr_val_to_array (varfl **vals, varfl v, int *nv)
  Store a value to an array of floats.
- int bufr_desc_to_array (dd *descs, dd d, int *ndescs)
  Store a descriptor to an array.
- int bufr_get_ndescs (bufr_t *msg)
  Calculate number of data descriptors in a BUFR message.
- void bufr_get_date_time (long *year, long *mon, long *day, long *hour, long *min)
  Recall date/time info of the last BUFR-message created.
- int bufr_open_datasect_w ()
  Opens bitstream for section 4 writing.
- int bufr_open_datasect_r (bufr_t *msg)
  Opens bitstream for reading section 4.
- void bufr_close_datasect_w (bufr_t *msg)
  Closes bitstream for section 4 and adds data to BUFR message.
- void bufr_close_datasect_r ()
  Closes bitstream for section 4.
- int bufr_val_from_global (varfl *val, int ind)
  Get one value from global array of values.
- int bufr_val_to_global (varfl val, int ind)
  Write one value to global array of values.
- bufrval_t * bufr_open_val_array ()
  Opens global array of values for read/write.
- void bufr_close_val_array ()
  Closes global array of values and frees all memory.

Detailed Description

This file contains all functions used for encoding and decoding data to BUFR format.

Function Documentation

int bufr_create_msg (dd * *descs*, int *ndescs*, varfl * *vals*, void ** *datasec*, void ** *ddsec*, size_t * *datasecl*, size_t * *ddescl*)

Deprecated:

Use bufr_encode_sections34 instead.

This function codes data from an array data descriptors `descs` and an array of varfl-values `vals` to a data section and a data descripor section of a BUFR message. Memory for both sections is allocated in this function and must be freed by the calling functions.

Parameters:

descs Data-descriptors corresponding to `vals` . For each descriptor there must be a data-vaule stored in `vals` . `descs` may also include replication factors and sequence descriptors. In that case there must be a larger number of `vals` then of `descs` .
ndescs Number of data descriptos contained in `descs` .
vals Data-values to be coded in the data section. For each entry in `descs` there must be an entry in `vals` . If there are relication factors in `descs` , of course there must be as much `vals` as definded by the replication factor.
datasec Is where the data-section (section 4) is stored. The memory-area for the data-section is allocated by this function and must be freed by the calling function.
ddsec Is where the data-descriptor-section (section 3) in stored. The memory needed is allocated by this function and must be freed by the calling function.
datasecl Number of bytes in `datasec` .
ddescl Number of bytes in `ddsec` .

Returns:

The return-value is 1 if data was successfully stored, 0 if not.

See also:

bufr_read_msg, bufr_data_from_file

---

bufr.h File Reference

bufr.hDefinitions of main OPERA BUFR library functions.

Data Structures

- struct bufr_t
  Structure that holds the encoded bufr message.
- struct bufrval_t
  Structure holding values for callbacks bufr_val_from_global and bufr_val_to_global .

Defines

- #define MAX_DESCS 1000
  Maximum number of data descriptors in a BUFR message.

Typedefs

- typedef char * bd_t
  one bufr data element is a string

Functions

- int bufr_create_msg (dd *descs, int ndescs, varfl *vals, void **datasec, void **ddsec, size_t *datasecl, size_t *ddescl)
  Creates section 3 and 4 of BUFR message from arrays of data and data descriptors.
- int bufr_encode_sections34 (dd *descs, int ndescs, varfl *vals, bufr_t *msg)
  Creates section 3 and 4 of BUFR message from arrays of data and data descriptors.
- int bufr_encode_sections0125 (sect_1_t *s1, bufr_t *msg)
  This function creates sections 0, 1, 2 and 5.
- int bufr_write_file (bufr_t *msg, char *file)
  This functions saves the encoded BUFR-message to a binary file.
- int bufr_read_file (bufr_t *msg, char *file)
  This functions reads the encoded BUFR-message to a binary file.
- int bufr_get_sections (char *bm, int len, bufr_t *msg)
  Calculates the section length of a BUFR message and allocates memory for each section.
- int bufr_decode_sections01 (sect_1_t *s1, bufr_t *msg)
  This function decodes sections 0 and 1.
- int bufr_read_msg (void *datasec, void *ddsec, size_t datasecl, size_t ddescl, dd **desc, int *ndescs, varfl **vals, size_t *nvals)
  Decode BUFR data and descriptor section and write values and descriptors to arrays.
- void bufr_sect_1_from_file (sect_1_t *s1, char *file)
  Reads section 1 from a file and stores data read in s1.
- int bufr_open_descsec_w ()
  Open bitstream for section 3 for writing and set default values.
- int bufr_out_descsec (dd *descp, int ndescs, int desch)
  Write descriptor section of a BUFR message to the bitsream.
- void bufr_close_descsec_w (bufr_t *bufr, int desch)
  Write length of section 3 and close bitstream.
- int bufr_parse_in (dd *descs, int start, int end, int(*inputfkt)(varfl *val, int ind), int callback_descs)
  Parse data descriptors and call user defined input function for each element or for each descriptor.
- int bufr_open_descsec_r (bufr_t *msg)
  Open bitstream of section 3 for reading.
- int bufr_get_ndescs (bufr_t *msg)
  Calculate number of data descriptors in a BUFR message.
- int bufr_in_descsec (dd **descs, int ndescs, int desch)
  Read descriptor section of a BUFR message from the bitsream.
- void bufr_close_descsec_r (int desch)
  close bitstream for section 3
- int bufr_parse_out (dd *descs, int start, int end, int(*outputfkt)(varfl val, int ind), int callback_all_descs)
  Parse data descriptors and call user defined output function for each element or for each descriptor.
- int bufr_sect_1_to_file (sect_1_t *s1, char *file)
  Writes section 1 data to an ASCII file.
- void bufr_free_data (bufr_t *d)
  Frees memory allocated for a BUFR message.
- int bufr_check_fxy (dd *d, int ff, int xx, int yy)
  Tests equality of descriptor d with (f,x,y).
- void bufr_get_date_time (long *year, long *mon, long *day, long *hour, long *min)
  Recall date/time info of the last BUFR-message created.

- int bufr_val_to_array (varfl **vals, varfl v, int *nvals)
  Store a value to an array of floats.
- int bufr_desc_to_array (dd *descs, dd d, int *ndescs)
  Store a descriptor to an array.
- int bufr_parse_new (dd *descs, int start, int end, int(*inputfkt)(varfl *val, int ind), int(*outputfkt)(varfl val, int ind), int callback_all_descs)
  Parse data descriptors and call user defined functions for each data element or for each descriptor.
- int bufr_parse (dd *descs, int start, int end, varfl *vals, unsigned *vali, int(*userfkt)(varfl val, int ind))
  Parse data descriptors and call user-function for each element.
- bufrval_t * bufr_open_val_array ()
  Opens global array of values for read/write.
- void bufr_close_val_array ()
  Closes global array of values and frees all memory.
- int bufr_open_datasect_w ()
  Opens bitstream for section 4 writing.
- void bufr_close_datasect_w (bufr_t *msg)
  Closes bitstream for section 4 and adds data to BUFR message.
- int bufr_open_datasect_r (bufr_t *msg)
  Opens bitstream for reading section 4.
- void bufr_close_datasect_r ()
  Closes bitstream for section 4.
- int bufr_val_from_global (varfl *val, int ind)
  Get one value from global array of values.
- int bufr_val_to_global (varfl val, int ind)
  Write one value to global array of values.
- void bufr_clean ()
- int val_to_array (varfl **vals, varfl v, size_t *nvals)
- int setup_sec0125 (char *sec[], size_t secl[], sect_1_t s1)

Variables

- int _bufr_edition
  global bufr edition number
- int _replicating
  global replication indicator

Detailed Description

This file contains declaration of functions used for encoding and decoding data to BUFR format.

Function Documentation

int bufr_create_msg (dd * *descs*, int *ndescs*, varfl * *vals*, void ** *datasec*, void ** *ddsec*, size_t * *datasecl*, size_t * *ddescl*)

Deprecated:

Use bufr_encode_sections34 instead.

This function codes data from an array data descriptors `descs` and an array of varfl-values `vals` to a data section and a data descripor section of a BUFR message. Memory for both sections is allocated in this function and must be freed by the calling functions.

Parameters:

descs Data-descriptors corresponding to `vals` . For each descriptor there must be a data-vaule stored in `vals` . `descs` may also include replication factors and sequence descriptors. In that case there must be a larger number of `vals` then of `descs` .
ndescs Number of data descriptos contained in `descs` .
vals Data-values to be coded in the data section. For each entry in `descs` there must be an entry in `vals` . If there are relication factors in `descs` , of course there must be as much `vals` as definded by the replication factor.
datasec Is where the data-section (section 4) is stored. The memory-area for the data-section is allocated by this function and must be freed by the calling function.
ddsec Is where the data-descriptor-section (section 3) in stored. The memory needed is allocated by this function and must be freed by the calling function.
datasecl Number of bytes in `datasec` .
ddescl Number of bytes in `ddsec` .

Returns:

The return-value is 1 if data was successfully stored, 0 if not.

See also:

bufr_read_msg, bufr_data_from_file

Variable Documentation

int _bufr_edition

The bufr edition number is stored in section 0 of the BUFR message. It is used by the software for determining the format of section 1.

See also:

bufr_get_date_time, bufr_encode_sections0125, bufr_decode_sections01, bufr_parse_new, bufr_val_from_datasect, bufr_val_to_datasect

int _replicating

This flag is used to indicate an ongoing data replication and is set by bufr_parse_new . It can be used for different output formating when a replication occurs.

See also:

bufr_parse_new, bufr_file_out

---

bufr_io.c File Reference

bufr_io.cFunctions for reading/writing to/from OPERA format ASCII BUFR files.

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <errno.h>
#include "desc.h"
#include "bufr.h"
#include "bitio.h"
#include "rlenc.h"
```

Defines

- #define BUFR_OUT_BIN 0

Output to binary format for flag tables.

Functions

- int bufr_data_from_file (char *file, bufr_t *msg)
  read data and descriptors from ASCII file and code them into sections 3 and 4
- int bufr_data_to_file (char *file, char *imgfile, bufr_t *msg)
  Decode data and descriptor sections of a BUFR message and write them to an ASCII file.

Detailed Description

This file contains functions for reading/writing to/from OPERA format ASCII BUFR files.

---

bufr_io.h File Reference

bufr_io.hIncludes functions for reading/writing to/from OPERA format ASCII BUFR files.

Functions

- int bufr_data_from_file (char *file, bufr_t *msg)
  read data and descriptors from ASCII file and code them into sections 3 and 4
- int bufr_data_to_file (char *file, char *imgfile, bufr_t *msg)
  Decode data and descriptor sections of a BUFR message and write them to an ASCII file.

Detailed Description

This file includes functions for reading/writing to/from OPERA format ASCII BUFR files.

---

bufrlib.h File Reference

bufrlib.hIncludes all functions for the OPERA BUFR software library.

```
#include "desc.h"
#include "bufr.h"
#include "bitio.h"
#include "rlenc.h"
```

Detailed Description

This file includes all header files used by the OPERA BUFR software library.

---

decbufr.c File Reference

decbufr.cReads a BUFR-file, decodes it and stores decoded data in a text-file.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "bufrlib.h"
```

```
#include "bufr_io.h"
```

Detailed Description

This function reads a BUFR-file, decodes it and stores decoded data in a text-file. Decoded bitmaps are stored in a seperate file.

---

desc.c File Reference

desc.cFunctions for reading the descriptor tables.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>
#include "desc.h"
```

Functions

- int read_tables (char *dir, int vmtab, int vltab, int subcent, int gencent)
  Reads bufr tables from csv-files.
- void show_desc (int f, int x, int y)
  Prints the specified descriptor or all if f = 999.
- int get_index (int typ, dd *descr)
  Returns the index for the given descriptor and typ.
- int read_tab_d (char *fname)
  Reads bufr table d from a csv-files.
- int read_tab_b (char *fname)
  Reads bufr table b from a csv-files.
- void free_descs (void)
  Frees all memory that has been allocated for data descriptors.
- int desc_is_flagtable (int ind)
- int desc_is_codetable (int ind)
- void trim (char *buf)
  Deletes all terminating blanks in a string.
- char * get_unit (dd *d)
  Returns the unit for a given data descriptor.

Detailed Description

This file contains all functions used for reading the decriptor tables and utilites for managing the data descriptors.

Function Documentation

int desc_is_flagtable (int *ind*)

Checks if a descriptor is a flag-table.

Parameters:

ind Index to the global array des [] holding the description of known data-descriptors.

Returns:

1 if descriptor is a flag-table, 0 if not.

See also:

desc_is_codetable

int desc_is_codetable (int *ind*)

Checks if a descriptor is a code-table.

Parameters:

ind Index to the global array des [] holding the description of known data-descriptors.

Returns:

1 if descriptor is a code-table, 0 if not.

See also:

desc_is_flagtable

void trim (char * *buf*)

This functions deletes all terminating blanks in a string.

Parameters:

buf Our string.

---

desc.h File Reference

desc.hData structures needed for holding the supported data-descriptors.

Data Structures

- struct sect_1_t
  Holds the information contained in section 1.
- struct dd
  Describes one data descriptor.
- struct del
  Defines an element descriptor.
- struct dseq
  Structure that defines a sequence of descriptors.
- struct desc
  Structure that defines one descriptor. This can be an element descriptor or a sequence descriptor.

Defines

- #define MISSVAL 99999.999999
- #define SEQDESC 0
  Identifier for a sequence descriptor.
- #define ELDESC 1
  Identifier for an element descriptor.
- #define MAXDESC 2000
  Max. number of descriptors in the global descriptor-array (des).

Typedefs

- typedef double varfl
  Defines the internal float-variable type.

Functions

- int read_tab_b (char *fname)
  Reads bufr table b from a csv-files.
- int read_tab_d (char *fname)
  Reads bufr table d from a csv-files.
- char * get_unit (dd *d)
  Returns the unit for a given data descriptor.
- int get_index (int typ, dd *d)
  Returns the index for the given descriptor and typ.
- void free_descs (void)
  Frees all memory that has been allocated for data descriptors.
- void trim (char *buf)
  Deletes all terminating blanks in a string.
- int read_tables (char *dir, int vm, int vl, int subcenter, int gencenter)
  Reads bufr tables from csv-files.
- void show_desc (int f, int x, int y)
  Prints the specified descriptor or all if f = 999.
- int desc_is_codetable (int ind)
- int desc_is_flagtable (int ind)

Variables

- int ndes
  Total number of descriptors found.
- desc * des [MAXDESC+OPTDESC]
  Array holding all data descriptors.
- int dw
  Current data width modification factor (default: 128).
- int sc
  Current scale modification factor (default: 128).
- int addfields
  Number of associated fields to be added to any data-item.
- int ccitt_special
  Special index for ccitt characters.
- int add_f_special
  Special index for associated fields.
- int _desc_special
  Special index for descriptors without data.

Detailed Description

This file defines the data-structures needed to hold the supported data-descriptors. Also defines all functions used for reading the decriptor tables and utilites for managing the data descriptors.

Define Documentation

#define MISSVAL 99999.999999

This is the internal missing value indicator. Missing values are indicated as "missing" and if we find such a value we set it internally to MISSVAL

Examlples:

apisample.c

Typedef Documentation

typedef double varfl

Defines the internal float-variable type. This can be float or double. Float needs less memory than double. Double-floats need not to be converted by your machine before operation (software runs faster). The default is double.

Note:

The format-string in all scanf-calls must be changed for `varfl-values !`

Examlples:

apisample.c

Function Documentation

void trim (char * *buf*)

This functions deletes all terminating blanks in a string.

Parameters:

buf Our string.

int desc_is_codetable (int *ind*)

Checks if a descriptor is a code-table.

Parameters:

ind Index to the global array des [] holding the description of known data-descriptors.

Returns:

1 if descriptor is a code-table, 0 if not.

See also:

desc_is_flagtable

int desc_is_flagtable (int *ind*)

Checks if a descriptor is a flag-table.

Parameters:

ind Index to the global array des [] holding the description of known data-descriptors.

Returns:

1 if descriptor is a flag-table, 0 if not.

See also:

desc_is_codetable

Variable Documentation

desc* des[MAXDESC+OPTDESC]

Array holding all data descriptors. The descriptors are read from the descriptor table files using read_tables or read_tab_b and read_tab_d

See also:

read_tables, read_tab_b, read_tab_d, get_index

Examlples:

apisample.c

int dw

Current data width modification factor (default: 128) Add dw - 128 to the data-width (dw can be optionally set by 2 01 YYY)

int sc

Current scale modification factor (default: 128). Add sc - 128 to the scale-factor (sc can be optionally set by 2 02 YYY)

int addfields

Number of associated fields to be added to any data-item. `addfields` can be set by 2 04 YYY and canceled by 2 04 000

int ccitt_special

This index is used by bufr_parse_new and its derivates to indicate that a value is a CCITT character

See also:

bufr_parse_new, Callback functions for encoding to BUFR, Callback functions for decoding from BUFR

int add_f_special

This index is used by bufr_parse_new and its derivates to indicate that a value is an associated field.

See also:

bufr_parse_new, Callback functions for encoding to BUFR, Callback functions for decoding from BUFR

int _desc_special

This index is used by bufr_parse_new and its derivates to indicate that we have a descriptor without value for output.

See also:

bufr_parse_new, Callback functions for decoding from BUFR

Examlples:

apisample.c

---

encbufr.c File Reference

encbufr.cReads source-data from a textfile and codes it into a BUFR-file.

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include "bufrlib.h"
#include "bufr_io.h"
```

Detailed Description

This function reads source-data from a textfile and codes is into a BUFR-file. Bitmaps are read from a seperate file.

---

rlenc.c File Reference

rlenc.cFunctions for run-length encoding and decoding.

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "desc.h"
#include "bufr.h"
#include "rlenc.h"
```

Defines

- #define LBUFLEN 5000
  Size of the internal buffer holding one uncompressed line.
- #define ENCBUFL 5000
  Size of the internal buffer holding one compressed line.

Functions

- int rlenc (char *infile, int nrows, int ncols, varfl **vals, size_t *nvals)
  Runlength-encodes a radar image.
- int rlenc_compress_line (int line, unsigned char *src, int ncols, varfl **dvals, size_t *nvals)
  Encodes one line of a radar image to BUFR runlength-code.
- int rldec (char *outfile, varfl *vals, size_t *nvals)
  Decodes a BUFR-runlength-encoded radar image.
- int rlenc_from_file (char *infile, int nrows, int ncols, varfl **vals, int *nvals, int depth)
  Runlength-encodes a radar image from a file to an array.
- int rldec_to_file (char *outfile, varfl *vals, int depth, int *nvals)
  Decodes a BUFR-runlength-encoded radar image to a file.
- int rlenc_from_mem (unsigned short *img, int nrows, int ncols, varfl **vals, int *nvals)
  This function encodes a radar image to BUFR runlength-code.
- int rldec_to_mem (varfl *vals, unsigned short **img, int *nvals, int *nrows, int *ncols)
  Decodes a BUFR-runlength-encoded radar image to memory.
- int rlenc_compress_line_new (int line, unsigned int *src, int ncols, varfl **dvals, int *nvals)
  Encodes one line of a radar image to BUFR runlength-code.
- void rldec_decompress_line (varfl *vals, unsigned int *dest, int *ncols, int *nvals)
  Decodes one line of a radar image from BUFR runlength-code.
- void rldec_get_size (varfl *vals, int *nrows, int *ncols)
  Gets the number of rows and columns of a runlength compressed image.

Detailed Description

This file contains all functions used for run-length encoding and decoding of image files.

---

rlenc.h File Reference

rlenc.hFunction definitions for run-length encoding and decoding.

Functions

- int rlenc_from_file (char *infile, int nrows, int ncols, varfl **vals, int *nvals, int depth)
  Runlength-encodes a radar image from a file to an array.
- int rlenc_from_mem (unsigned short *img, int nrows, int ncols, varfl **vals, int *nvals)
  This function encodes a radar image to BUFR runlength-code.
- int rldec_to_file (char *outfile, varfl *vals, int depth, int *nvals)
  Decodes a BUFR-runlength-encoded radar image to a file.
- int rldec_to_mem (varfl *vals, unsigned short **img, int *nvals, int *nrows, int *ncols)
  Decodes a BUFR-runlength-encoded radar image to memory.
- int rlenc_compress_line_new (int line, unsigned int *src, int ncols, varfl **dvals, int *nvals)
  Encodes one line of a radar image to BUFR runlength-code.
- void rldec_decompress_line (varfl *vals, unsigned int *dest, int *ncols, int *nvals)
  Decodes one line of a radar image from BUFR runlength-code.
- void rldec_get_size (varfl *vals, int *nrows, int *ncols)
  Gets the number of rows and columns of a runlength compressed image.
- int rlenc (char *infile, int nrows, int ncols, varfl **vals, size_t *nvals)
  Runlength-encodes a radar image.
- int rldec (char *outfile, varfl *vals, size_t *nvals)
  Decodes a BUFR-runlength-encoded radar image.
- int rlenc_compress_line (int line, unsigned char *src, int ncols, varfl **dvals, size_t *nvals)
  Encodes one line of a radar image to BUFR runlength-code.

Detailed Description

This file contains all functions used for run-length encoding and decoding of image files.

OPERA BUFR software Example Documentation

apisample.c

This is an example for encoding and decoding a BUFR massage.

```
/*---------------------------------------------------------------------

     B U F R   E N C O D I N G   A N D   D E C O D I N G   S O F T W A R E

FILE:          APISAMPLE.C
IDENT:         $Id: apisample.c,v 1.0 2007-12-07 09:44:49+01 fuxi Exp fuxi $

AUTHOR:        Juergen Fuchsberger
               Institute of Broadband Communication,
               Technical University Graz, Austria
```

```
VERSION NUMBER:3.0

DATE CREATED:  4-DEC-2007

STATUS:        DEVELOPMENT FINISHED

AMENDMENT RECORD:

$Log: apisample.c,v $
Revision 1.0  2007-12-07 09:44:49+01  fuxi
Initial revision


------------------------------------------------------------------------ */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "bufrlib.h"
#include "apisample.h"
#include "bufr_io.h"

/*==========================================================================*/
/* internal function definitons                                             */
/*==========================================================================*/

static void create_source_msg (dd* descs, int* nd, varfl** vals,
                               radar_data_t* d);
static int our_callback (varfl val, int ind);
static void create_sample_data (radar_data_t* d);

/*==========================================================================*/
/* internal data                                                            */
/*==========================================================================*/

radar_data_t our_data; /* sturcture holding our decoded data */
char *version = "apisample V3.0, 5-Dec-2007\n";

/*==========================================================================*/

void bufr_encoding_sample (radar_data_t* src_data, bufr_t* bufr_msg) {

    sect_1_t s1;          /* structure holding information from section 1 */
    dd descs[MAX_DESCS];  /* array of data descriptors, must be large enough
                             to hold all required descriptors */
    int nd = 0;           /* current number of descriptors in descs */
    varfl* vals = NULL;   /* array of data values */
    int ok;

    long year, mon, day, hour, min;

    memset (&s1, 0, sizeof (sect_1_t));

    /* first let's create our source message */

    create_source_msg (descs, &nd, &vals, src_data);

    /* Prepare data for section 1 */

    s1.year = 999;
    s1.mon  = 999;
    s1.day = 999;
    s1.hour = 999;
    s1.min  = 999;
    s1.mtab = 0;                    /* master table used */
    s1.subcent = 255;               /* originating subcenter */
    s1.gencent = 255;               /* originating center */
    s1.updsequ = 0;                 /* original BUFR message */
    s1.opsec = 0;                   /* no optional section */
    s1.dcat = 6;                    /* message type */
    s1.dcatst = 0;                  /* message subtype */
    s1.vmtab = 11;                  /* version number of master table used */
    s1.vltab = 4;                   /* version number of local table used */

    /* read supported data descriptors from tables */
```

```
    ok = (read_tables (NULL, s1.vmtab, s1.vltab, s1.subcent, s1.gencent) >= 0);

    /* encode our data to a data-descriptor- and data-section */

    if (ok) ok = bufr_encode_sections34 (descs, nd, vals, bufr_msg);

    /* setup date and time if necessary */

    if (ok && s1.year == 999) {
        bufr_get_date_time (&year, &mon, &day, &hour, &min);
        s1.year = (int) year;
        s1.mon = (int) mon;
        s1.day = (int) day;
        s1.hour = (int) hour;
        s1.min = (int) min;
        s1.sec = 0;
    }

    /* encode section 0, 1, 2, 5 */

    if (ok) ok = bufr_encode_sections0125 (&s1, bufr_msg);

    /* Save coded data */

    if (ok) ok = bufr_write_file (bufr_msg, "apisample.bfr");

    if (vals != NULL)
        free (vals);
    free_descs ();

    if (!ok) exit (EXIT_FAILURE);
}

/*=========================================================================*/
void bufr_decoding_sample (bufr_t* msg, radar_data_t* data) {

    sect_1_t s1;
    int ok, desch, ndescs;
    dd* dds = NULL;

    /* initialize variables */

    memset (&s1, 0, sizeof (sect_1_t));

    /* Here we could also read our BUFR message from a file */
    /* bufr_read_file (msg, buffile); */

    /* decode section 1 */

    ok = bufr_decode_sections01 (&s1, msg);

    /* Write section 1 to ASCII file */

    bufr_sect_1_to_file (&s1, "section.1.out");

    /* read descriptor tables */

    if (ok) ok = (read_tables (NULL, s1.vmtab, s1.vltab, s1.subcent,
                               s1.gencent) >= 0);

    /* decode data descriptor and data-section now */

    /* open bitstreams for section 3 and 4 */

    desch = bufr_open_descsec_r(msg);
    ok = (desch >= 0);
    if (ok) ok = (bufr_open_datasect_r(msg) >= 0);

    /* calculate number of data descriptors  */

    ndescs = bufr_get_ndescs (msg);

    /* allocate memory and read data descriptors from bitstream */

    if (ok) ok = bufr_in_descsec (&dds, ndescs, desch);

    /* output data to our global data structure */
```

```
    if (ok) ok = bufr_parse_out (dds, 0, ndescs - 1, our_callback, 1);

    /* get data from global */

    data = &our_data;

    /* close bitstreams and free descriptor array */

    if (dds != (dd*) NULL)
        free (dds);
    bufr_close_descsec_r (desch);
    bufr_close_datasect_r ();

    /* decode data to file also */

    if (ok) ok = bufr_data_to_file ("apisample.src", "apisample.img", msg);

    bufr_free_data (msg);
    free_descs();
    exit (EXIT_SUCCESS);


}

/*=============================================================================*/
/*
Sample for encoding and decoding a BUFR message

*/

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

    bufr_t bufr_msg ;   /* structure holding encoded bufr message */

    /* initialize variables */

    memset (&bufr_msg, 0, sizeof (bufr_t));
    memset (&our_data, 0, sizeof (radar_data_t));

    /* check command line parameters */

    while (argc > 1 && *argv[1] == '-')
    {
        if (*(argv[1] + 1) == 'v')
            fprintf (stderr, "%s", version);
    }

    /* sample for encoding to BUFR */

    create_sample_data (&our_data);
    bufr_encoding_sample (&our_data, &bufr_msg);

    /* sample for decoding from BUFR */

    memset (&our_data, 0, sizeof (radar_data_t));
    bufr_decoding_sample (&bufr_msg, &our_data);
    bufr_free_data (&bufr_msg);

    free (our_data.img.data);

    exit (EXIT_SUCCESS);
}


/*=============================================================================*/
#define fill_desc(ff,xx,yy) {\
        dd.f=ff; dd.x=xx; dd.y=yy; \
        bufr_desc_to_array (descs, dd, nd);}
#define fill_v(val) bufr_val_to_array (vals, val, &nv);

static void create_source_msg (dd* descs, int* nd, varfl** vals,
                               radar_data_t* d) {

    dd dd;
    int nv = 0, i;

    fill_desc(3,1,1);           /* WMO block and station number */
    fill_v(d->wmoblock);
```

```
    fill_v(d->wmostat);

    fill_desc(3,1,192);         /* Meta information about the product */
    fill_v(d->meta.year);       /* Date */
    fill_v(d->meta.month);
    fill_v(d->meta.day);
    fill_v(d->meta.hour);       /* Time */
    fill_v(d->meta.min);
    fill_v(d->img.nw.lat);      /* Lat. / lon. of NW corner */
    fill_v(d->img.nw.lon);
    fill_v(d->img.ne.lat);      /* Lat. / lon. of NE corner */
    fill_v(d->img.ne.lon);
    fill_v(d->img.se.lat);      /* Lat. / lon. of SE corner */
    fill_v(d->img.se.lon);
    fill_v(d->img.sw.lat);      /* Lat. / lon. of SW corner */
    fill_v(d->img.sw.lon);
    fill_v(d->proj.type);            /* Projection type */
    fill_v(d->meta.radar.lat);       /* Latitude of radar */
    fill_v(d->meta.radar.lon);       /* Longitude of radar */
    fill_v(d->img.psizex);           /* Pixel size along x coordinate */
    fill_v(d->img.psizey);           /* Pixel size along y coordinate */
    fill_v(d->img.nrows);            /* Number of pixels per row */
    fill_v(d->img.ncols);            /* Number of pixels per column */

    fill_desc(3,1,22);          /* Latitude, longitude and height of station */
    fill_v(d->meta.radar.lat);
    fill_v(d->meta.radar.lon);
    fill_v(d->meta.radar_height);

                                /* Projection information (this will be
                                   a sequence descriptor when using tables 6 */
    fill_desc(0,29,199);        /* Semi-major axis or rotation ellipsoid */
    fill_v(d->proj.majax);
    fill_desc(0,29,200);        /* Semi-minor axis or rotation ellipsoid */
    fill_v(d->proj.minax);
    fill_desc(0,29,193);        /* Longitude Origin */
    fill_v(d->proj.orig.lon);
    fill_desc(0,29,194);        /* Latitude Origin */
    fill_v(d->proj.orig.lat);
    fill_desc(0,29,195);        /* False Easting */
    fill_v(d->proj.xoff);
    fill_desc(0,29,196);        /* False Northing */
    fill_v(d->proj.yoff);
    fill_desc(0,29,197);        /* 1st Standard Parallel */
    fill_v(d->proj.stdpar1);
    fill_desc(0,29,198);        /* 2nd Standard Parallel */
    fill_v(d->proj.stdpar2);

    fill_desc(0,30,31);         /* Image type */
    fill_v(d->img.type);

    fill_desc(0,29,2);          /* Co-ordinate grid */
    fill_v(d->img.grid);

    fill_desc(0,33,3);          /* Quality information */
    fill_v(d->img.qual);


    /* level slicing table note the use of change of datawith in order to
       encode our values, also values are converted to integer, loosing
       precision
    */

    fill_desc(2,1,129);          /* change of datawidth because 0 21 1
                                    only codes to 7 bit */
    fill_desc(3,13,9);          /* Reflectivity scale */
    fill_v(d->img.scale.vals[0]);   /* scale[0] */
    fill_v(d->img.scale.nvals -1);     /* number of scale values - 1 */
    for (i = 1; i < d->img.scale.nvals; i++) {
        fill_v(d->img.scale.vals[i]);
    }
    fill_desc(2,1,0);          /* cancel change of datawidth */

    /* another possibility for the level slicing table withour using
       datawidth and scale change and without loosing precision */

    fill_desc(0,21,198);        /* dBZ Value offset */
    fill_v(d->img.scale.offset);
```

```
    fill_desc(0,21,199);        /* dBZ Value increment */
    fill_v(d->img.scale.increment);


    fill_desc(3,21,193);        /* 8 bit per pixel pixmap */

    /* run length encode our bitmap */
    rlenc_from_mem (d->img.data, d->img.nrows, d->img.ncols, vals, &nv);

    free(d->img.data);
}

/*=========================================================================*/

static int our_callback (varfl val, int ind) {

    radar_data_t* b = &our_data;   /* our global data structure */
    bufrval_t* v;                  /* array of data values */
    varfl* vv;
    int i = 0, nv, nr, nc;
    dd* d;

    /* do nothing if data modifictaon descriptor or replication descriptor */

    if (ind == _desc_special) return 1;

    /* sequence descriptor */

    if (des[ind]->id == SEQDESC) {

        /* get descriptor */

        d = &(des[ind]->seq->d);

        /* open array for values */

        v = bufr_open_val_array ();
        if (v == (bufrval_t*) NULL) return 0;

        /* WMO block and station number */

        if  (bufr_check_fxy (d, 3,1,1)) {

            /* decode sequence to global array */

            bufr_parse_out (des[ind]->seq->del, 0, des[ind]->seq->nel - 1,
                            bufr_val_to_global, 0);

            /* get our data from the array */

            b->wmoblock = (int) v->vals[i++];
            b->wmostat = (int) v->vals[i];

        }
        /* Meta information */

        else if (bufr_check_fxy (d, 3,1,192)) {

            bufr_parse_out (des[ind]->seq->del, 0, des[ind]->seq->nel - 1,
                            bufr_val_to_global, 0);
            vv = v->vals;
            i = 0;
            b->meta.year = (int) vv[i++];       /* Date */
            b->meta.month = (int) vv[i++];
            b->meta.day = (int) vv[i++];
            b->meta.hour = (int) vv[i++];       /* Time */
            b->meta.min = (int) vv[i++];
            b->img.nw.lat = vv[i++];      /* Lat. / lon. of NW corner */
            b->img.nw.lon = vv[i++];
            b->img.ne.lat = vv[i++];      /* Lat. / lon. of NE corner */
            b->img.ne.lon = vv[i++];
            b->img.se.lat = vv[i++];      /* Lat. / lon. of SE corner */
            b->img.se.lon = vv[i++];
            b->img.sw.lat = vv[i++];      /* Lat. / lon. of SW corner */
            b->img.sw.lon = vv[i++];
            b->proj.type = (int) vv[i++];       /* Projection type */
            b->meta.radar.lat = vv[i++];        /* Latitude of radar */
            b->meta.radar.lon = vv[i++];        /* Longitude of radar */
```

```
            b->img.psizex = vv[i++];      /* Pixel size along x coordinate */
            b->img.psizey = vv[i++];      /* Pixel size along y coordinate */
            b->img.nrows = (int) vv[i++];     /* Number of pixels per row */
            b->img.ncols = (int) vv[i++];     /* Number of pixels per column */

        }
        /* Latitude, longitude and height of station */

        else if (bufr_check_fxy (d, 3,1,22)) {

            bufr_parse_out (des[ind]->seq->del, 0, des[ind]->seq->nel - 1,
                            bufr_val_to_global, 0);
            vv = v->vals;
            i = 0;
            b->meta.radar.lat = vv[i++];
            b->meta.radar.lon = vv[i++];
            b->meta.radar_height = vv[i];
        }
        /* Reflectivity scale */

        else if (bufr_check_fxy (d, 3,13,9)) {
            int j;

            bufr_parse_out (des[ind]->seq->del, 0, des[ind]->seq->nel - 1,
                            bufr_val_to_global, 0);
            vv = v->vals;
            i = 0;

            b->img.scale.vals[0] = vv[i++];
            b->img.scale.nvals = (int) vv[i++] + 1;  /* number of scale values */
            assert(b->img.scale.nvals < 256);
            for (j = 1; j < b->img.scale.nvals; j++) {
                b->img.scale.vals[j] = vv[i++];
            }
        }

        /* our bitmap */

        else if (bufr_check_fxy (d, 3,21,193)) {

            /* read bitmap and run length decode */

            if (!bufr_parse_out (des[ind]->seq->del, 0, des[ind]->seq->nel - 1,
                                 bufr_val_to_global, 0)) {
                bufr_close_val_array ();
                return 0;
            }

            if (!rldec_to_mem (v->vals, &(b->img.data), &nv, &nr, &nc)) {
                bufr_close_val_array ();
                fprintf (stderr, "Error during runlength-compression.\n");
                return 0;
            }
        }

        else {
            fprintf (stderr,
                     "Unknown sequence descriptor %d %d %d", d->f, d->x, d->y);
        }
        /* close the global value array */

        bufr_close_val_array ();

    }

    /* element descriptor */

    else if (des[ind]->id == ELDESC) {

        d = &(des[ind]->el->d);

        if (bufr_check_fxy (d, 0,29,199))
            /* Semi-major axis or rotation ellipsoid */
            b->proj.majax = val;
        else if (bufr_check_fxy (d, 0,29,200))
            /* Semi-minor axis or rotation ellipsoid */
            b->proj.minax = val;
        else if (bufr_check_fxy (d, 0,29,193))
```

```
            /* Longitude Origin */
            b->proj.orig.lon = val;
        else if (bufr_check_fxy (d, 0,29,194))
            /* Latitude Origin */
            b->proj.orig.lat = val;
        else if (bufr_check_fxy (d, 0,29,195))
            /* False Easting */
            b->proj.xoff = (int) val;
        else if (bufr_check_fxy (d, 0,29,196))
            /* False Northing */
            b->proj.yoff = (int) val;
        else if (bufr_check_fxy (d, 0,29,197))
            /* 1st Standard Parallel */
            b->proj.stdpar1 = val;
        else if (bufr_check_fxy (d, 0,29,198))
            /* 2nd Standard Parallel */
            b->proj.stdpar2 = val;
        else if (bufr_check_fxy (d, 0,30,31))
            /* Image type */
            b->img.type = (int) val;
        else if (bufr_check_fxy (d, 0,29,2))
            /* Co-ordinate grid */
            b->img.grid = (int) val;
        else if (bufr_check_fxy (d, 0,33,3))
            /* Quality information */
            b->img.qual = val;
        else if (bufr_check_fxy (d, 0,21,198))
            /* dBZ Value offset */
            b->img.scale.offset = val;
        else if (bufr_check_fxy (d, 0,21,199))
            /* dBZ Value increment */
            b->img.scale.increment = val;
        else {
            fprintf (stderr,
                     "Unknown element descriptor %d %d %d", d->f, d->x, d->y);
            return 0;
        }
    }
    return 1;
}

/*=============================================================================*/
#define NROWS 200   /* Number of rows for our sample radar image */
#define NCOLS 200   /* Number of columns for our sample radar image */

static void create_sample_data (radar_data_t* d) {

    int i;

    /* create a sample radar image */

    d->img.data = (unsigned short*) calloc (NROWS * NCOLS,
                                            sizeof (unsigned short));

    if (d->img.data == NULL) {
        fprintf (stderr, "Could not allocate memory for sample image!\n");
        exit (EXIT_FAILURE);
    }

    /* fill image with random data (assuming 8 bit image depth -> max
       value = 254; 255 is missing value) */

#ifdef VERBOSE
    fprintf (stderr, "RAND_MAX = %d\n", RAND_MAX);
#endif

    for (i = 0; i < NROWS * NCOLS; i++) {
        d->img.data[i] = (unsigned short) ((float) rand() / RAND_MAX * 254);
#ifdef VERBOSE
        fprintf (stderr, "Value: %d\n", d->img.data[i]);
#endif
    }

    /* create our source data */

    d->wmoblock = 11;
    d->wmostat  = 164;
```

```
    d->meta.year = 2007;
    d->meta.month = 12;
    d->meta.day = 5;
    d->meta.hour = 12;
    d->meta.min = 5;
    d->meta.radar.lat = 47.06022;
    d->meta.radar.lon = 15.45772;
    d->meta.radar_height = 355;

    d->img.nw.lat = 50.4371;
    d->img.nw.lon = 8.1938;
    d->img.ne.lat = 50.3750;
    d->img.ne.lon = 19.7773;
    d->img.se.lat = 44.5910;
    d->img.se.lon = 19.1030;
    d->img.sw.lat = 44.6466;
    d->img.sw.lon = 8.7324;
    d->img.psizex = 1000;
    d->img.psizey = 1000;
    d->img.nrows = NROWS;
    d->img.ncols = NCOLS;
    d->img.type = 2;
    d->img.grid = 0;
    d->img.qual = MISSVAL;

    /* create level slicing table */

    d->img.scale.nvals = 255;

    for (i = 0; i < 255; i++) {
        d->img.scale.vals[i] = i * 0.5 - 31.0;
    }
    d->img.scale.offset = -31;
    d->img.scale.increment = 0.5;

    d->proj.type = 2;
    d->proj.majax = 6378137;
    d->proj.minax = 6356752;
    d->proj.orig.lon = 13.333333;
    d->proj.orig.lat = 47.0;
    d->proj.xoff = 458745;
    d->proj.yoff = 364548;
    d->proj.stdpar1 = 46.0;
    d->proj.stdpar2 = 49.0;
}

/* end of file */
```

OPERA BUFR software Page Documentation

Deprecated List**Deprecated List**

Global bufr_create_msg

Use bufr_encode_sections34 instead.

Global bufr_clean

use free_descs instead

Global setup_sec0125

use bufr_encode_sections0125 instead

Global save_sections

Use bufr_write_file instead.

Global val_to_array

use bufr_val_to_array instead.

Global rlenc

Use rlenc_from_file instead.

Global rlenc_compress_line

Use rlenc_compress_line_new instead.

Global rldec

Use rldec_to_file instead.