NAME

BS – BS file data library

DESCRIPTION

/usr/lib/libbs.a is a collection of functions which allow a programmer to manipulate Hawaii Mapping Research Group (HMRG) BS data files.

USAGE

Following is a description of all available subroutines in the BS library. An application referencing any of these routines must be linked to the library at compile time by specifying the **-lbs** flag. The memory allocation library may also need to be referenced via the **-lmem** flag if the application calls certain of the routines described below.

Most of the subroutines described below which return an integer will return either **BS_SUCCESS** or a defined failure code such as **BS_READ**, **BS_MEMALLOC**, etc. (See **ERROR CODES** below.) Return values for all other functions will be explicitly described.

Many of the subroutines which relate to input or output require a pointer to an open XDR stream as one of the arguments. Such a stream will generally be obtained by calling **xdrstdio_create()** on an open file pointer. All of the input functions assume that the XDR stream is appropriately positioned at the time of the function call, e.g., a function which attempts to read a particular type of header will succeed only if the XDR stream is currently positioned at the beginning of such a header.

Note that the file reading functions described below are capable of reading files stored in certain obsolete versions of the file format as well as files stored in the current format. The file writing functions, however, will always write output files in the current format regardless of the value of the **bsf_version** field of the file header which defines the format version.

#include <rpc/rpc.h>
#include <sys/time.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <local/bs.h>

int
bs_rdbsfhdr(BSFile *_bsf_, XDR *_xdrs_)

reads the next BS file header from the XDR input stream *xdrs* and stores it into the user-allocated structure *bsf*.

int
bs_wrbsfhdr(BSFile *_bsf_ , XDR *_xdrs_)

writes the BS file header *bsf* to the XDR output stream *xdrs*.

int
bs_freebsfmem(BSFile *_bsf_)

frees all memory referenced by the structure members of *bsf* which was allocated by **bs_rdbsfhdr()**. It does not free the memory addressed by *bsf* itself. The function should generally be used between successive calls to **bs_rdbsfhdr()**.

int
bs_rdversion(FILE *_fp_, int *_version_)

reads the file format version of *fp* and stores it into *version*. The function should be called only with *fp* at the very beginning of the input stream, which in this case is a file pointer rather than the more common XDR stream. It is normally used only when an application needs to determine the format version of a dataset and no subsequent reading is intended. Nearly all callers should instead use **bs_rdbsfhdr()** as described above to read the entire BS file header (including the format version) and leave the input stream positioned in a more useful location, i.e., at the beginning of the first ping header.

int
bs_rdpnghdr(Ping *_png_, XDR *_xdrs_, int _version_)

reads the next BS ping header from the XDR input stream *xdrs* and stores it into the user-allocated structure *png*. The version of the file format being read must be indicated by *version*, which should normally be the value of the *bsf_version* field of the input file's file header record.

int
bs_wrpnghdr(Ping ****png*, XDR ****xdrs***)**
writes the ping header *png* to the XDR output stream *xdrs*.

int
bs_pngdatabufsz(Ping ****png*, unsigned long long ****pngsz***)**
writes into *pngsz* the size in bytes of the smallest buffer capable of holding the various data (e.g., sensor, bathymetry, sidescan, auxiliary beam information, etc.) associated with *png*. The function itself returns **BS_SUCCESS** or an error code in the event that a valid buffer size cannot be determined due to negative or otherwise invalid sample count values as described by *png*. A reasonable (but possibly not airtight and definitely not precise) effort is made to guard against pings with overly large sample counts, as the original format implementation implicitly limited the maximum ping buffer size to be no greater than the number of bytes which could be described by a signed 32-bit integer, i.e., **BS_MAXSIGNEDINT32** (2147483647). This routine has been coded to enforce that limitation, but will attempt if possible to determine the actual size of a ping buffer even when it would be greater than this limit. In any case where the size of the ping is known to be over the limit, or it is believed that it might be over the limit under circumstances which diminish the precision of the size computation due to overflow issues, the function will return **BS_HUGEPING**. In such cases the value written into *pngsz* will be accurate if and only if the host architecture supports 8-byte (or larger) unsigned long long integers.

MemType *
bs_pngmemalloc(Ping ****png*)**
allocates enough memory to store the data associated with the ping header *png*. A pointer to the allocated memory is returned. (The pointer may be a null pointer if the function fails for any reason.) The quantity of memory allocated is dependent upon the number of data samples and padding samples indicated by *png*. Padding samples, i.e., meaningless placeholder samples that are immediately contiguous and subsequent to valid data samples, are never stored to files, but it is sometimes convenient to allocate this additional sample memory at the time a ping is read in order to perform an operation that may result in an increase in the number of meaningful samples associated with that ping. An arbitrary amount of such additional memory can be allocated with this function by setting *png*→**png_snspad** and the **ps_btypad** and **ps_sspad** fields of the *png*→**png_sides[ACP_PORT]** and *png*→**png_sides[ACP_STBD]** substructures to appropriate values before the function is called.

int
bs_pngrealloc(Ping ****png*, MemType *****data***, unsigned int ****datasz)***
allocates memory similarly to **bs_pngmemalloc()** as described above but is generally more convenient to use. Its second and third arguments are the addresses of a buffer pointer and an integer describing the size of the buffer. The buffer pointer and the integer should be set to the null pointer and 0, respectively, before the first call to this routine. Each time the routine is called it will determine the smallest buffer size sufficient to hold the data samples and padding samples indicated by *png*. If the buffer pointed to by **data* is large enough (as described by **datasz*) to hold those samples then it is zeroed and nothing else is done, otherwise the existing buffer (if any) is freed, a new buffer is allocated, and **data* and **datasz* are updated to reflect the new buffer and its size. Note that, unlike **bs_pngmemalloc()**, the return value of this function is an error code (e.g., **BS_SUCCESS**) and not a pointer to the buffer.

int
bs_rdpngdata(Ping ****png*, MemType ****data***, XDR ****xdrs***)**
reads ping data from the XDR input stream *xdrs* and stores it into the memory pointed to by *data*. The number of data samples to be read is obtained from the header structure *png*. This routine assumes that the *data* memory has already been allocated (e.g., by **bs_pngrealloc()**). Note that each bathymetry sample consists of either two or three consecutive floating point values depending upon the value of *png*→**png_flags**. If the **PNG_XYZ** bit of the latter is set then each sample is an x/y/z triplet with the first value representing across-

track distance, the second value representing along-track distance and the third value representing depth. If the **PNG_XYZ** bit is not set then the data are in x/z format, i.e., the along-track distance is not present. Compass samples are read and stored into the first *png*→**png_compass.sns_nsamps*sizeof(float)** bytes of *data*. Towfish depth samples are read and stored into the next *png*→**png_depth.sns_nsamps*sizeof(float)** bytes. Pitch samples are read and stored into the next *png*→**png_pitch.sns_nsamps*sizeof(float)** bytes. Roll samples are read and stored into the next *png*→**png_roll.sns_nsamps*sizeof(float)** bytes. Port bathymetry samples are read and stored into the next *bsi*png*→**png_sides[ACP_PORT].ps_btycount*sizeof(float)** bytes, where *bsi* is 2 or 3 depending on whether the samples are in x/z or x/y/z format. Port bathymetry flags are read and stored into the next *png*→**png_sides[ACP_PORT].ps_btycount*sizeof(unsigned int)** bytes. Port sidescan samples are read and stored into the next *png*→**png_sides[ACP_PORT].ps_sscount*sizeof(float)** bytes. Port sidescan flags are read and stored into the next *png*→**png_sides[ACP_PORT].ps_sscount*sizeof(unsigned char)** bytes. Starboard bathymetry samples are read and stored into the next *bsi*png*→**png_sides[ACP_STBD].ps_btycount*sizeof(float)** bytes (where *bsi* is 2 or 3 as described above). Starboard bathymetry flags are read and stored into the next *png*→**png_sides[ACP_STBD].ps_btycount*sizeof(unsigned int)** bytes. Starboard sidescan samples are read and stored into the next *png*→**png_sides[ACP_STBD].ps_sscount*sizeof(float)** bytes. Starboard sidescan flags are read and stored into the next *png*→**png_sides[ACP_STBD].ps_sscount*sizeof(unsigned char)** bytes. Port auxiliary beam information is read and stored into the next *png*→**png_sides[ACP_PORT].ps_btycount*sizeof(AuxBeamInfo)** bytes. Finally, starboard auxiliary beam information is read and stored into the last *png*→**png_sides[ACP_STBD].ps_btycount*sizeof(AuxBeamInfo)** bytes. If sample padding has been specified by the *png* header as described above, each group of samples as described above may be separated from the preceding group by a byte offset corresponding to the amount of padding applied to the previous group. For instance, if the port bathymetry has been padded, then (i) the port bathymetry flags will be offset from the end of the port bathymetry samples by a gap which is *bsi*png*→**png_sides[ACP_PORT].ps_btypad*sizeof(float)** bytes in length (where *bsi* is 2 or 3 as described above), (ii) the port sidescan samples will be offset from the end of the port bathymetry flags by a gap which is *png*→**png_sides[ACP_PORT].ps_btypad*sizeof(unsigned int)** bytes in length, and (iii) the starboard auxiliary beam information will be offset from the end of the port auxiliary beam information by a gap which is *png*→**png_sides[ACP_PORT].ps_btypad*sizeof(AuxBeamInfo)** bytes in length. Sensor sample padding as specified by *png*→**png_snspad** is applied between the last group of sensor samples and the port bathymetry, not between each group (e.g., compass and depth) of sensor samples. Note that the port auxiliary beam information is constrained to begin on a byte which is offset from *data* by a multiple of **PNG_BYTEALIGNSZ** bytes.

int
bs_wrpngdata(Ping *_png_, MemType *_data_, XDR *_xdrs_)

writes the ping data pointed to by *data* and associated with the ping header *png* to the XDR output stream *xdrs*. The number of data samples to be written is obtained from *png*. (See **bs_rdpngdata()** above for a detailed description of the organization of the contents of *data*.) Padding samples, if any, are not written, but their presence as indicated by the **png_snspad**, **ps_btypad** and **ps_sspad** fields of *png* will affect the offsets from *data* at which the various bathymetry samples, bathymetry flags, sidescan samples and auxiliary beam information are presumed to be located.

int
bs_getpngdataptrs(Ping *_png_, MemType *_data_, PingData *_pngdata_)

returns into the fields of *pngdata* pointers to the various components (e.g., sensor samples, bathymetry samples, bathymetry flags, sidescan samples and auxiliary beam information) of the *data* buffer associated with *png*. Null pointers may be returned into fields when certain components are not present, e.g., auxiliary beam information.

int
bs_rdpngpddata(Ping *_png_, PingData *_pngdata_, XDR *_xdrs_)

reads the ping data associated with the ping header *png* from the XDR input stream *xdrs*, storing the

various components of the data into the memory buffers pointed to by the fields of *pngdata*. This routine assumes that these buffers have already been allocated. The number of data samples to be read is obtained from *png*.

int
bs_wrpngpddata(Ping ****png*, PingData ****pngdata***, XDR ****xdrs***)**
writes the ping data whose various components are pointed to by the fields of *pngdata* and are associated with the ping header *png* to the XDR output stream *xdrs*. The number of data samples to be written is obtained from *png*.

int
bs_rdpng(Ping ****png*, MemType *****data***, XDR ****xdrs***, int** *version***)**
reads a ping from the XDR input stream *xdrs*. The header will be stored into *png*, and the data will be stored into **data*. This routine allocates the memory pointed to by **data*. (The *data* parameter should be passed as the address of a memory pointer variable, which will be set to point to the newly allocated memory.) The version of the file format being read must be indicated by *version*, which should normally be the value of the **bsf_version** field of the input file's file header record. This function performs the same operation as calling **bs_rdpnghdr(), bs_pngmemalloc()** and **bs_rdpngdata()** in succession. Note that it is not possible to allocate sample padding with this routine.

int
bs_wrpng(Ping ****png*, MemType ****data***, XDR ****xdrs***)**
writes both the ping header pointed to by *png* and the data pointed to by *data* to the XDR output stream *xdrs*.

int
bs_seekpng(int *n***, XDR ****xdrs***, int** *version***)**
skips over the next *n* pings in the XDR input stream *xdrs*, leaving the stream positioned at the beginning of the next ping. The version of the file format being read must be indicated by *version*, which should normally be the value of the **bsf_version** field of the input file's file header record.

int
bs_seekpngdata(Ping ****png*, XDR ****xdrs***)**
skips over a ping data segment (whose size is described by *png*) in the XDR input stream *xdrs*, leaving the stream positioned at the beginning of the next ping.

int
bs_copypng(int *n***, XDR ****xdris***, XDR ****xdros***, int** *version***)**
copies the next *n* pings from the XDR input stream *xdris* to the XDR output stream *xdros*, leaving the input stream positioned at the beginning of the next ping. The version of the file format being read must be indicated by *version*, which should normally be the value of the **bsf_version** field of the input file's file header record.

The stream-oriented nature of the I/O routines described above dictates that BS datafiles will generally be processed by reading an input file and then writing a new output file, where the latter is written in full from beginning to end. It is convenient in some circumstances, however, to modify an existing file in place rather than create a new file, particularly in the case where only the file header flags and/or a small number of ping header field values or ping sample values or flags must be altered, e.g., ping flags, navigation data, individual bathymetry or sidescan sample flags, etc. A crude mechanism is provided to enable this via the publicly accessable global variable

unsigned long bs_iobytecnt
and a small number of write functions. The **bs_iobytecnt** variable is always set by all of the above I/O routines to the exact number of bytes transferred from/to an input/output file by any particular call to such a routine. (The **bs_copypng()** function which both reads and writes data stores the number of written output bytes to **bs_iobytecnt**.) A calling program can therefore monitor this variable carefully and retain knowledge of the exact file byte offsets (from the beginning of the file) of each ping header in the file. These

remembered ping header byte offsets, which must take into account the number of bytes used to store the initial file header as well as each ping header and each ping data segment, can then be passed to the functions

int
bs_wrpflags(int *version***, FILE** ***fp*****, long** *phoffset***, unsigned int** *flags***)**
int
bs_wrsllc(int *version***, FILE** ***fp*****, long** *phoffset***, double** *slon***, double** *slat***, float** *scourse***)**
and

int
bs_wrtllc(int *version***, FILE** ***fp*****, long** *phoffset***, double** *tlon***, double** *tlat***, float** *tcourse***)**
to directly rewrite the ping flags (via the first function), the longitude, latitude and course of the ship (via the second function) and the longitude, latitude and course of the towfish (via the third function), where *version* is the BS file format version as recorded in the file's **bsf_version** file header field. The function

int
bs_wrtll(int *version***, FILE** ***fp*****, long** *phoffset***, double** *tlon***, double** *tlat***)**
directly rewrites only the longitude and latitude of the towfish.

int
bs_wrfflagssetbits(FILE ***fp*****, unsigned int** *bitmask***)**
and

int
bs_wrfflagsclrbits(FILE ***fp*****, unsigned int** *bitmask***)**
are similarly used to set and/or clear the bits of *bitmask* to and/or from the file header flags while preserving the state of all other bit flags, while

int
bs_wrpflagssetbits(int *version***, FILE** ***fp*****, long** *phoffset***, unsigned int** *bitmask***)**
and

int
bs_wrpflagsclrbits(int *version***, FILE** ***fp*****, long** *phoffset***, unsigned int** *bitmask***)**
may be used to set and/or clear the bits of *bitmask* to and/or from the ping flags while preserving the state of all other bit flags.

int
bs_setswradius(int *version***, FILE** ***fp*****, long** *phoffset***, int** *side***, unsigned int** *datatypemask***, float** *swradius***)**
flags all samples of any data type whose mask bit is present in *datatypemask* (which must contain either or both of the mask bits **BS_DTM_BATHYMETRY** and/or **BS_DTM_SIDESCAN**) on the named *side* (either **ACP_PORT** or **ACP_STBD**) at across-track distances greater than *swradius* with **{BTYD,SSD}_SWEDGE** for the ping whose header is located at the named file byte offset, thus effectively trimming the swath radius of that *side* of the ping to *swradius*.

Note that a file pointer rather than an XDR stream is passed to all of these file and ping header field and sample flag rewrite functions, which will internally seek to the specified file byte offset *phoffset* marking the start of some particular ping header and write XDR-formatted data at appropriate offsets from that point. The file pointer will be positioned just after the modified bytes when these routines return. Note that these functions are exceedingly dangerous insofar as the use of an incorrect *phoffset* which does not actually reference the exact beginning of a ping header will certainly result in a fatally corrupted datafile.

int
bs_xdrstring(XDR ***xdrs*****, char** ****cpp*****, unsigned long** ***bytecnt***)
was originally created only for internal use by the various BS I/O routines described above, but has since been made publicly available due to its more generally useful performance of XDR character string encoding and decoding. It is not typically used by any calling application to access BS datafiles, but rather to

access other files used by HMRG software which employ a similar style of XDR character string storage where the string is stored as an integer (the string length) followed by the bytes of the string (if the length is greater than 0). The routine returns 1 if successful and 0 otherwise, also recording the total number of bytes transferred (including the leading integer) into **bytecnt*.

int
bs_appendstr(char *****field*, char ****string***)**

appends the specified *string* to any character string *field* of an existing BS header. Note that the *field* parameter must be the address of the header's character string field, and not the string itself. This routine will allocate new memory for the appended string and deallocate the memory consumed by the previous string where appropriate.

int
bs_replacestr(char *****field*, char ****string***)**

replaces an existing character string *field* of an BS header with the specified *string*. Note that the *field* parameter must be the address of the header's character string field, and not the string itself. This routine will allocate new memory for the replacement string and deallocate the memory consumed by the previous string where appropriate. (A copy is made of the character string pointed to by *string*, so *string* may be safely deallocated, overwritten, etc., after the function returns.)

int
bs_striptail(char ****string*, char** *c***)**

strips all consecutive instances of *c* from the end of *string*.

int
bs_appendlog(BSFile ****bsf*, char *****argv***)**

appends the specified argument vector *argv* to the log field of the named BS header, inserting a blank space between each of the character strings pointed to by *argv* and appending a trailing semicolon to the final string. The routine will also append a newline to the pre-existing log field before appending *argv* if that pre-existing log field is non-empty. The last element of the *argv* array of character pointers must be a null pointer. This routine will allocate new memory for the modified log field and deallocate the memory consumed by the previous log field.

Two routines are provided for the generation of single- and double-precision IEEE NaN (not-a-number) quantities which are used by the **bsfile**(4) format to note that the value of a certain parameter (e.g., the towfish pulse length as described by the **ps_pulse** field of the **PingSide** data structure) is unknown.

float
bs_nanf()

and

double
bs_nand()

respectively generate these single- and double-precision NaN quantities. Each of the routines

int
bs_isnanf(float *f***)**

and

int
bs_isnand(double *d***)**

will return 1 if its argument is a NaN quantity and 0 otherwise.

A group of routines are provided for the manipulation of ping marks, which are used to flag pings either within a single program or between cooperating applications. A ping mark will have an integer value which is either **BS_NULLMARK** or some bitwise combination of the bitflags **BS_LOWMARK** and/or **BS_HIGHMARK**. Each side of a ping, **ACP_PORT** and **ACP_STBD**, is marked separately.

void *

bs_mrkmemalloc(int *size***)**

allocates enough memory to maintain ping marks for a group of *size* pings, sets all of those marks to **BS_NULLMARK** and returns a pointer to that memory. (The pointer may be a null pointer if the function fails for any reason.)

int
bs_mrkget(void *****markers*, **int** *side*, **int** *pingid***)**

returns the mark value of the specified *pingid* on the declared *side* from the ping mark memory buffer *markers*.

void
bs_mrkset(void *****markers*, **int** *side*, **int** *pingid*, **int** *value***)**

sets the mark value of the specified *pingid* on the declared *side* in the ping mark memory buffer *markers* to the stated *value*.

The pre-processor macro

int
bs_pngvisible(*flags***)**

returns zero if either of the **PNG_HIDE** or **PNG_LOWQUALITY** bits of *flags* (which should be the **png_flags** field of a **Ping** structure) is set, and non-zero otherwise. The

int
bs_pngmscvisible(*flags***)**

macro returns zero if any of the **PNG_MSCHIDE**, **PNG_HIDE** or **PNG_LOWQUALITY** bits of *flags* (which should be the **png_flags** field of a **Ping** structure) is set, and non-zero otherwise.

The routine

int
bs_splitfile(char *****dirnm*, **char *****bsfnm0*, **char *****bsfnm1*, **int *pngid*, **char *****logprefix***)**

splits the existing BS datafile *bsfnm0* located in directory *dirnm* into two pieces, leaving the initial *pngid* pings in *bsfnm0* and creating a new file *bsfnm1* to contain the remaining pings. (Note that a new *bsfnm0* will actually be recreated from the original *bsfnm0* which is then removed.) The *dirnm* argument may be set to a null pointer to indicate that the operation should be performed within the current directory. The *logprefix* argument, which may also be a null pointer, should point to a short string (typically just the name of the calling program) which will be incorporated into both output files' log entries along with *pngid* by the routine.

Finally, the routines

int
bs_tmparse(char ***str*, **int *mode*, **double ***tmval***)**

and

int
bs_tmparsegmttz(char ***str*, **int *mode*, **double ***tmval***)**

parse a character string *str* of the form

year/*julianday*-*hour*:*minute*:*second*

(when *mode* is **TM_JULIAN**) or

year/*month*/*day*-*hour*:*minute*:*second*

(when *mode* is **TM_CALENDAR**), setting **tmval* equal to the number of seconds since January 1, 1970, represented by the time described within *str*. **bs_tmparsegmttz()** should be used only when the calling application's environment is using the GMT timezone, but is considerably more efficient in terms of memory usage than the more general **bs_tmparse()** when the routine is to be called a large number of times. The routines will allow any of the field separation characters '/', '**-**', '**:**' and/or ' ' to be used interchangeably within *str*, e.g., '92-67-1-23-56' and '92/67 1:23:56' denote the same time value. All fields except the *year* are optional and, if not specified, will be set to appropriate minimum values. If any particular field other

than the *year* is specified, however, then all other fields which would normally precede that field within the string must also be specified. The *year* will be interpreted explicitly unless it is (i) between 0 and 49, in which case it will be interpreted as 2000+*year*, or (ii) between 50 and 99, in which case it will be interpreted as 1900+*year*. Month and day values are specified in normal human (rather than Unix) format, meaning that the *julianday* may range from 1 to 366, while the calendar *month* and *day* may range from 1 to 12 and 1 to 31 respectively. The *second* field may include a decimal fraction if so desired, while all other fields must be non-negative integers. The length of *str* may not exceed **TM_MAXSTRLEN**.

ERROR CODES

The following error codes are defined by /usr/local/bs.h.

```
#define BS_SUCCESS          (0)
#define BS_FAILURE          (1)
#define BS_FILTERWAIT       (2)
#define BS_MISC             (3)
#define BS_BADARG           (4)
#define BS_MEMALLOC         (5)
#define BS_OPEN             (6)
#define BS_READ             (7)
#define BS_WRITE            (8)
#define BS_SYSVIPC          (9)
#define BS_X11              (10)
#define BS_SIGNAL           (11)
#define BS_PIPE             (12)
#define BS_FCNTL            (13)
#define BS_FORK             (14)
#define BS_DUP2             (15)
#define BS_CHDIR            (16)
#define BS_EXEC             (17)
#define BS_PDB              (18)
#define BS_EOF              (19)
#define BS_BADDATA          (20)
#define BS_FSEEK            (21)
#define BS_ACCESS           (22)
#define BS_RENAME           (23)
#define BS_BADARCH          (24)
#define BS_HUGEPING         (25)
#define BS_GTK              (26)
#define BS_CAIRO            (27)
```

SEE ALSO

bsfile(4)

AUTHOR

Roger Davis, July 2005.