

NAME

mbio – Format independent input/output library for swath mapping sonar data.

VERSION

Version 5.0

DESCRIPTION

MBIO (**M**ulti**B**eam **I**nterface **O**utput) is a library of functions used for reading and writing swath mapping sonar data files. **MBIO** supports a large number of data formats associated with different institutions and different sonar systems. The purpose of **MBIO** is to allow users to write processing and display programs which are independent of particular data formats and to provide a standard approach to swath mapping sonar data i/o.

MB-SYSTEM AUTHORSHIP

David W. Caress
Monterey Bay Aquarium Research Institute
Christian do Santos Ferreira
Center for Marine Environmental Sciences (MARUM)
University of Bremen
Dale N. Chayes
Center for Coastal and Ocean Mapping
University of New Hampshire

DATA TERMINOLOGY

MBIO handles three types of swath mapping data: beam bathymetry, beam amplitude, and sidescan. Both amplitude and sidescan represent measures of backscatter strength. Beam amplitudes are backscatter values associated with the same preformed beams used to obtain bathymetry; **MBIO** assumes that a bathymetry value exists for each amplitude value and uses the bathymetry beam location for the amplitude. Sidescan is generally constructed with a higher spatial resolution than bathymetry, and carries its own location parameters. In the context of **MB-System** documentation, the discrete values of bathymetry and amplitude are referred to as "beams", and the discrete values of sidescan are referred to as "pixels". An additional difference between "beam" and "pixel" data involves data flagging. An array of "beamflags" is carried by **MBIO** functions which allows the bathymetry (and by extension the amplitude) data to be flagged as bad. The details of the beamflagging scheme are presented below.

OVERVIEW

MBIO opens and initializes sonar data files for reading and writing using the functions **mb_read_init** and **mb_write_init**, respectively. These functions return a pointer to a data structure including all relevant information about the opened file, the control parameters which determine how data is read or written, and the arrays used for processing the data as it is read or written. This pointer is then passed to the functions used for reading or writing. There is no limit on the number of files which may be opened for reading or writing at any given time in a program.

The **mb_read_init** and **mb_write_init** functions also return initial maximum numbers of bathymetry beams, amplitude beams, and sidescan pixels that can be used to allocate data storage arrays of the appropriate sizes. However, for some data formats there are no specified maximum numbers of beams and pixels, and so in general the required dimensions may increase as data are read. Applications must pass appropriately dimensioned arrays into data extraction routines such as **mb_read**, **mb_get**, and **mb_get_all**. In order to enable dynamic memory management of these application arrays, the application must first register each array by passing the array pointer location to the function **mb_register_array**.

Data files are closed using the function **mb_close**. All internal and registered arrays are deallocated as part

of closing the file.

When it comes to actually reading and writing swath mapping sonar data, **MBIO** has three levels of i/o functionality:

- 1: Simple reading of swath data files. The primary functions are:

mb_read()

mb_get()

The positions of individual beams and pixels are returned in longitude and latitude by

mb_read() and in acrosstrack and alongtrack distances by **mb_get()**. Only a limited set of navigation information is returned. Comments are also returned. These functions can be used without any special include files or any knowledge of the actual data structures used by the data formats or **MBIO**.

- 2: Complete reading and writing of data structures containing all of the available information. Data records may be read or written without extracting any of the information, or the swath data may be passed with the data structure. Several functions exist to extract information from or insert information into the data structures; otherwise, special include files are required to make sense of the sonar-specific data structures passed by level 2 i/o functions. The basic read and write functions that only pass pointers to internal data structures are:

mb_read_ping()

mb_write_ping()

The read and write routines which also extract or insert information are:

mb_get_all()

mb_put_all()

mb_put_comment()

The information extraction and insertion functions are:

mb_insert()

mb_extract()

mb_extract_lonlat()

mb_extract_nav()

mb_insert_nav()

mb_extract_altitude()

mb_insert_altitude()

mb_ttimes()

mb_copyrecord()

- 3: Buffered reading and writing of data structures containing all of the available information.

The primary functions are:

mb_buffer_init()

mb_buffer_close()

mb_buffer_load()

mb_buffer_dump()

```

mb_buffer_info()
mb_buffer_get_next_data()
mb_buffer_extract()
mb_buffer_insert()
mb_buffer_get_next_nav()
mb_buffer_extract_nav()
mb_buffer_insert_nav()

```

The level 1 **MBIO** functions allow users to read sonar data independent of format, with the limitation that only a limited set of navigation information is passed. Thus, some of the information contained in certain data formats (e.g. the "heave" value in Hydrosweep DS data) is not passed by **mb_read()** or **mb_get()**. In general, the level 1 functions are useful for applications such as graphics which require only the navigation and the depth and/or backscatter values.

The level 2 functions (**mb_get_all()** and **mb_put_all()**) read and write the complete data structures, translate the data to internal data structures associated with each of the supported sonar systems, and pass pointers to these internal data structures. Additional functions allow a variety of information to be extracted from or inserted into the data structures (e.g. **mb_extract()** and **mb_insert()**). Additional information may be accessed using special include files to decode the data structures. The great majority of processing programs use level 2 functions.

The level 3 functions provide buffered reading and writing which is useful for applications that generate output files and need access to multiple pings at a time. In addition to reading (**mb_buffer_load()**) and writing (**mb_buffer_dump()**), functions exist for extracting information from the buffer (**mb_buffer_extract()**) and inserting information into the buffer (**mb_buffer_insert()**).

MBIO supports swath data in a number of different formats, each specified by a unique id number. The function **mb_format()** determines if a format id is valid. A set of similar functions returns information about the specified format (e.g. **mb_format_description()**, **mb_format_system()**, **mb_format_description()**, **mb_format_dimensions()**, **mb_format_flags()**, **mb_format_source()**, **mb_format_beamwidth()**).

Some **MB-System** programs can process multiple data files specified in "datalist" files. Each line of a datalist file contains a file path and the corresponding **MBIO** format id. Datalist files can be recursive and can contain comments. The functions used to extract input swath data file paths from datalist files includes **mb_datalist_open()**, **mb_datalist_read()**, and **mb_datalist_close()**.

A number of other **MBIO** functions dealing with default values for important parameters, error messages, memory management, and time conversions also exist and are discussed below.

SUPPORTED SWATH SONAR SYSTEMS

Each swath mapping sonar system outputs a data stream which includes some values or parameters unique to that system. In general, a number of different data formats have come into use for data from each of the sonar systems; many of these formats include only a subset of the original data stream. Internally, **MBIO** recognizes which sonar system each data format is associated with and uses a data structure including the complete data stream for that sonar. Consequently, it is possible to read and write the complete data stream when using the level 2 or 3 **MBIO** functions. The sonars and other sensors associated with supported formats include:

```

SeaBeam "classic" 16 beam multibeam sonar
Hydrosweep DS 59 beam multibeam sonar
Hydrosweep MD 40 beam mid-depth multibeam sonar
SeaBeam 2000 multibeam sonar
SeaBeam 2112, 2120, and 2130 multibeam sonars
Simrad EM12, EM121, EM950, and EM1000 multibeam sonars

```

Simrad EM120, EM300, EM1002, and EM3000 multibeam sonars
 Kongsberg EM122, EM302, EM710, EM2040
 Kongsberg EM124, EM304, EM712, EM2040
 Hawaii MR-1 shallow tow interferometric sonar
 ELAC Bottomchart 1180 and 1050 multibeam sonars
 ELAC/SeaBeam Bottomchart Mk2 1180 and 1050 multibeam sonars
 Reson Seabat 9001/9002 multibeam sonars
 Reson Seabat 8101 multibeam sonars
 Simrad/Mesotech SM2000 multibeam sonars
 WHOI DSL AMS-120 deep tow interferometric sonar AMS-60 interferometric sonar
 WASSP multibeams (old format)
 Reson 7125 multibeam
 Teledyne T20, T50 multibeams
 R2Sonic multibeam
 3D at Depth subsea lidars (SL-1, SL-2, WiSSL)

SUPPORTED FORMATS

The following swath mapping sonar data formats are supported in this version of **MBIO**:

MBIO Data Format ID: 11
 Format name: MBF_SBSIOMRG
 Informal Description: SIO merge Sea Beam
 Attributes: Sea Beam, bathymetry, 16 beams, binary, uncentered,
 SIO.

MBIO Data Format ID: 12
 Format name: MBF_SBSIOCEN
 Informal Description: SIO centered Sea Beam
 Attributes: Sea Beam, bathymetry, 19 beams, binary, centered,
 SIO.

MBIO Data Format ID: 13
 Format name: MBF_SBSIOLSI
 Informal Description: SIO LSI Sea Beam
 Attributes: Sea Beam, bathymetry, 19 beams, binary, centered,
 obsolete, SIO.

MBIO Data Format ID: 14
 Format name: MBF_SBURICEN
 Informal Description: URI Sea Beam
 Attributes: Sea Beam, bathymetry, 19 beams, binary, centered,
 URI.

MBIO Data Format ID: 15
 Format name: MBF_SBURIVAX
 Informal Description: URI Sea Beam from VAX
 Attributes: Sea Beam, bathymetry, 19 beams, binary, centered,
 VAX byte order, URI.

MBIO Data Format ID: 16
 Format name: MBF_SBSIOSWB
 Informal Description: SIO Swath-bathy SeaBeam
 Attributes: Sea Beam, bathymetry, 19 beams, binary, centered,
 SIO.

MBIO Data Format ID: 17

Format name: MBF_SBIFREMR

Informal Description: IFREMER Archive SeaBeam

Attributes: Sea Beam, bathymetry, 19 beams, ascii, centered,
IFREMER.

MBIO Data Format ID: 21

Format name: MBF_HSATLRAW

Informal Description: Raw Hydrosweep

Attributes: Hydrosweep DS, bathymetry and amplitude, 59 beams,
ascii, Atlas Elektronik.

MBIO Data Format ID: 22

Format name: MBF_HSLDEDMB

Informal Description: EDMB Hydrosweep

Attributes: Hydrosweep DS, bathymetry, 59 beams, binary, NRL.

MBIO Data Format ID: 23

Format name: MBF_HSURICEN

Informal Description: URI Hydrosweep

Attributes: Hydrosweep DS, 59 beams, bathymetry, binary, URI.

MBIO Data Format ID: 24

Format name: MBF_HSLDEOIH

Informal Description: L-DEO in-house binary Hydrosweep

Attributes: Hydrosweep DS, 59 beams, bathymetry and amplitude,
binary, centered, L-DEO.

MBIO Data Format ID: 25

Format name: MBF_HSURIVAX

Informal Description: URI Hydrosweep from VAX

Attributes: Hydrosweep DS, 59 beams, bathymetry, binary,
VAX byte order, URI.

MBIO Data Format ID: 26

Format name: MBF_HSUNKNWN

Informal Description: Unknown Hydrosweep

Attributes: Hydrosweep DS, bathymetry, 59 beams, ascii, unknown origin, SOPAC.

MBIO Data Format ID: 32

Format name: MBF_SB2000SB

Informal Description: SIO Swath-bathy SeaBeam 2000 format

Attributes: SeaBeam 2000, bathymetry, 121 beams,
binary, SIO.

MBIO Data Format ID: 33

Format name: MBF_SB2000SS

Informal Description: SIO Swath-bathy SeaBeam 2000 format

Attributes: SeaBeam 2000, sidescan,
1000 pixels for 4-bit sidescan,
2000 pixels for 12+-bit sidescan,
binary, SIO.

MBIO Data Format ID: 41

Format name: MBF_SB2100RW
Informal Description: SeaBeam 2100 series vender format
Attributes: SeaBeam 2100, bathymetry, amplitude
and sidescan, 151 beams and 2000 pixels, ascii
with binary sidescan, SeaBeam Instruments.

MBIO Data Format ID: 42
Format name: MBF_SB2100B1
Informal Description: SeaBeam 2100 series vender format
Attributes: SeaBeam 2100, bathymetry, amplitude
and sidescan, 151 beams bathymetry,
2000 pixels sidescan, binary,
SeaBeam Instruments and L-DEO.

MBIO Data Format ID: 43
Format name: MBF_SB2100B2
Informal Description: SeaBeam 2100 series vender format
Attributes: SeaBeam 2100, bathymetry and amplitude,
151 beams bathymetry,
binary,
SeaBeam Instruments and L-DEO.

MBIO Data Format ID: 51
Format name: MBF_EMOLDRAW
Informal Description: Old Simrad vendor multibeam format
Attributes: Simrad EM1000, EM12S, EM12D,
and EM121 multibeam sonars,
bathymetry, amplitude, and sidescan,
60 beams for EM1000, 81 beams for EM12S/D,
121 beams for EM121, variable pixels,
ascii + binary, Simrad.

MBIO Data Format ID: 53
Format name: MBF_EM12IFRM
Informal Description: IFREMER TRISMUS format for Simrad EM12
Attributes: Simrad EM12S and EM12D,
bathymetry, amplitude, and sidescan
81 beams, variable pixels, binary,
read-only, IFREMER.

MBIO Data Format ID: 54
Format name: MBF_EM12DARW
Informal Description: Simrad EM12S RRS Darwin processed format
Attributes: Simrad EM12S, bathymetry and amplitude,
81 beams, binary, Oxford University.

MBIO Data Format ID: 56
Format name: MBF_EM300RAW
Informal Description: Simrad current multibeam vendor format
Attributes: Simrad EM120, EM300, EM1002, EM3000,
bathymetry, amplitude, and sidescan,
up to 254 beams, variable pixels, ascii + binary, Simrad.

MBIO Data Format ID: 57

Format name: MBF_EM300MBA
Informal Description: Simrad multibeam processing format
Attributes: Old and new Simrad multibeams,
EM12S, EM12D, EM121, EM120, EM300,
EM100, EM1000, EM950, EM1002, EM3000,
bathymetry, amplitude, and sidescan,
up to 254 beams, variable pixels, ascii + binary, MBARI.

MBIO Data Format ID: 58
Format name: MBF_EM710RAW
Informal Description: Kongsberg current multibeam vendor format
Attributes: Kongsberg EM122, EM302, EM710,
bathymetry, amplitude, and sidescan,
up to 400 beams, variable pixels, binary, Kongsberg.

MBIO Data Format ID: 59
Format name: MBF_EM710MBA
Informal Description: Kongsberg current multibeam processing format
Attributes: Kongsberg EM122, EM302, EM710,
bathymetry, amplitude, and sidescan,
up to 400 beams, variable pixels, binary, MBARI.

MBIO Data Format ID: 61
Format name: MBF_MR1PRHIG
Informal Description: Obsolete SOEST MR1 post processed format
Attributes: SOEST MR1, bathymetry and sidescan,
variable beams and pixels, xdr binary,
SOEST, University of Hawaii.

MBIO Data Format ID: 62
Format name: MBF_MR1ALDEO
Informal Description: L-DEO MR1 post processed format with travel times
Attributes: L-DEO MR1, bathymetry and sidescan,
variable beams and pixels, xdr binary,
L-DEO.

MBIO Data Format ID: 63
Format name: MBF_MR1BLDEO
Informal Description: L-DEO small MR1 post processed format with travel times
Attributes: L-DEO MR1, bathymetry and sidescan,
variable beams and pixels, xdr binary,
L-DEO.

MBIO Data Format ID: 64
Format name: MBF_MR1PRVR2
Informal Description: SOEST MR1 post processed format
Attributes: SOEST MR1, bathymetry and sidescan,
variable beams and pixels, xdr binary,
SOEST, University of Hawaii.

MBIO Data Format ID: 71
Format name: MBF_MBLDEOIH
Informal Description: L-DEO in-house generic multibeam
Attributes: Data from all sonar systems, bathymetry,

amplitude and sidescan, variable beams and pixels,
binary, centered, L-DEO.

MBIO Data Format ID: 72

Format name: MBF_MBARIMB1

Informal Description: MBARI TRN swath bathymetry

Attributes: Downsampled bathymetry from multibeam sonars,
bathymetry only, variable beams, binary, MBARI

MBIO Data Format ID: 75

Format name: MBF_MBNETCDF

Informal Description: CARAIBES CDF multibeam

Attributes: Data from all sonar systems, bathymetry only,
variable beams, netCDF, IFREMER.

MBIO Data Format ID: 76

Format name: MBF_MBNCDFXT

Informal Description: CARAIBES CDF multibeam extended

Attributes: Superset of MBF_MBNETCDF, includes (at least SIMRAD EM12) amplitude,
variable beams, netCDF, IFREMER.

MBIO Data Format ID: 81

Format name: MBF_CBAT9001

Informal Description: Reson SeaBat 9001 shallow water multibeam

Attributes: 60 beam bathymetry and amplitude,
binary, University of New Brunswick.

MBIO Data Format ID: 82

Format name: MBF_CBAT8101

Informal Description: Reson SeaBat 8101 shallow water multibeam

Attributes: 101 beam bathymetry and amplitude,
binary, SeaBeam Instruments.

MBIO Data Format ID: 83

Format name: MBF_HYPC8101

Informal Description: Reson SeaBat 8101 shallow water multibeam

Attributes: 101 beam bathymetry,
ASCII, read-only, Coastal Oceanographics.

MBIO Data Format ID: 84

Format name: MBF_XTFR8101

Informal Description: XTF format Reson SeaBat 81XX

Attributes: 240 beam bathymetry and amplitude,
1024 pixel sidescan
binary, read-only,
Triton-Elis.

MBIO Data Format ID: 88

Format name: MBF_RESON7KR

Informal Description: Reson 7K multibeam vendor format

Attributes: Reson 7K series multibeam sonars,
bathymetry, amplitude, three channels sidescan, and subbottom
up to 254 beams, variable pixels, binary, Reson.

MBIO Data Format ID: 89

Format name: MBF_RESON7K3

Informal Description: Reson 7K multibeam vendor format

Attributes: Reson 7K series multibeam sonars,
bathymetry, amplitude, three channels sidescan, and subbottom
up to 254 beams, variable pixels, binary, Reson.

MBIO Data Format ID: 91

Format name: MBF_BCHRTUNB

Informal Description: Elac BottomChart shallow water multibeam

Attributes: 56 beam bathymetry and amplitude,
binary, University of New Brunswick.

MBIO Data Format ID: 92

Format name: MBF_ELMK2UNB

Informal Description: Elac BottomChart MkII shallow water multibeam

Attributes: 126 beam bathymetry and amplitude,
binary, University of New Brunswick.

MBIO Data Format ID: 93

Format name: MBF_BCHRXUNB

Informal Description: Elac BottomChart shallow water multibeam

Attributes: 56 beam bathymetry and amplitude,
binary, University of New Brunswick.

MBIO Data Format ID: 94

Format name: MBF_L3XSERAW

Informal Description: ELAC/SeaBeam XSE vendor format

Attributes: Bottomchart MkII 50 kHz and 180 kHz multibeam,
SeaBeam 2120 20 KHz multibeam,
bathymetry, amplitude and sidescan,
variable beams and pixels, binary,
L3 Communications (Elac Nautik
and SeaBeam Instruments).

MBIO Data Format ID: 101

Format name: MBF_HSMDARAW

Informal Description: Atlas HSMD medium depth multibeam raw format

Attributes: 40 beam bathymetry, 160 pixel sidescan,
XDR (binary), STN Atlas Elektronik.

MBIO Data Format ID: 102

Format name: MBF_HSMDLDIH

Informal Description: Atlas HSMD medium depth multibeam processed format

Attributes: 40 beam bathymetry, 160 pixel sidescan,
XDR (binary), L-DEO.

MBIO Data Format ID: 111

Format name: MBF_DSL120PF

Informal Description: WHOI DSL AMS-120 processed format

Attributes: 2048 beam bathymetry, 8192 pixel sidescan,
binary, parallel bathymetry and amplitude files, WHOI DSL.

MBIO Data Format ID: 112

Format name: MBF_DSL120SF
Informal Description: WHOI DSL AMS-120 processed format
Attributes: 2048 beam bathymetry, 8192 pixel sidescan,
binary, single files, WHOI DSL.

MBIO Data Format ID: 121
Format name: MBF_GSFGENMB
Informal Description: Leidos Generic Sensor Format (GSF) version GSF-v03.09
Attributes: variable beams, bathymetry and amplitude,
binary, single files, Leidos (formerly SAIC).

MBIO Data Format ID: 131
Format name: MBF_MSTIFFSS
Informal Description: MSTIFF sidescan format
Attributes: variable pixels, sidescan,
binary TIFF variant, single files, Sea Scan.

MBIO Data Format ID: 132
Format name: MBF_EDGJSTAR
Informal Description: Edgetech Jstar format
Attributes: variable pixels, dual frequency sidescan and subbottom,
binary SEG Y variant, single files,
low frequency sidescan returned as
survey data, Edgetech.

MBIO Data Format ID: 133
Format name: MBF_EDGJSTR2
Informal Description: Edgetech Jstar format
Attributes: variable pixels, dual frequency sidescan and subbottom,
binary SEG Y variant, single files,
high frequency sidescan returned as
survey data, Edgetech.

MBIO Data Format ID: 141
Format name: MBF_OICGEODA
Informal Description: OIC swath sonar format
Attributes: variable beam bathymetry and
amplitude, variable pixel sidescan, binary,
Oceanic Imaging Consultants

MBIO Data Format ID: 142
Format name: MBF_OICMBARI
Informal Description: OIC-style extended swath sonar format
Attributes: variable beam bathymetry and
amplitude, variable pixel sidescan, binary,
MBARI

MBIO Data Format ID: 151
Format name: MBF_OMGHDCSJ
Informal Description: UNB OMG HDCS format (the John Hughes Clarke format)
Attributes: variable beam bathymetry and
amplitude, variable pixel sidescan, binary,
UNB

MBIO Data Format ID: 160

Format name: MBF_SEGYSEGY

Informal Description: SEG Y seismic data format

Attributes: seismic or subbottom trace data,
single beam bathymetry, nav,
binary, SEG (SIOSEIS variant)

MBIO Data Format ID: 161

Format name: MBF_MGD77DAT

Informal Description: NGDC MGD77 underway geophysics format

Attributes: single beam bathymetry, nav, magnetics,
gravity, ascii, NOAA NGDC

MBIO Data Format ID: 162

Format name: MBF_ASCIIXYZ

Informal Description: Generic XYZ sounding format

Attributes: XYZ (lon lat depth) ASCII soundings, generic

MBIO Data Format ID: 163

Format name: MBF_ASCIIYXZ

Informal Description: Generic YXZ sounding format

Attributes: YXZ (lat lon depth) ASCII soundings, generic

MBIO Data Format ID: 164

Format name: MBF_HYDROB93

Informal Description: NGDC binary hydrographic sounding format

Attributes: XYZ (lon lat depth) binary soundings

MBIO Data Format ID: 165

Format name: MBF_MBARIROV

Informal Description: MBARI ROV navigation format

Attributes: ROV navigation, MBARI

MBIO Data Format ID: 166

Format name: MBF_MBPRONAV

Informal Description: MB-System simple navigation format

Attributes: navigation, MBARI

MBIO Data Format ID: 167

Format name: MBF_NVNETCDF

Informal Description: CARAIBES CDF navigation

Attributes: netCDF, IFREMER.

MBIO Data Format ID: 168

Format name: MBF_ASCIIXYT

Informal Description: Generic XYT sounding format

Attributes: XYT (lon lat topography) ASCII soundings, generic

MBIO Data Format ID: 169

Format name: MBF_ASCIIYXT

Informal Description: Generic YXT sounding format

Attributes: YXT (lat lon topography) ASCII soundings, generic

MBIO Data Format ID: 170

Format name: MBF_MBARROV2
Informal Description: MBARI ROV navigation format
Attributes: ROV navigation, MBARI

MBIO Data Format ID: 171
Format name: MBF_HS10JAMS
Informal Description: Furuno HS-10 multibeam format,
Attributes: 45 beams bathymetry and amplitude,
ascii, JAMSTEC

MBIO Data Format ID: 172
Format name: MBF_HIR2RNAV
Informal Description: SIO GDC R2R navigation format
Attributes: R2R navigation, ascii, SIO

MBIO Data Format ID: 173
Format name: MBF_MGD77TXT
Informal Description: NGDC MGD77 underway geophysics format
Attributes: single beam bathymetry, nav, magnetics, gravity,
122 byte ascii records with CRLF line breaks, NOAA NGDC

MBIO Data Format ID: 174
Format name: MBF_MGD77TAB
Informal Description: NGDC MGD77 underway geophysics format
Attributes: single beam bathymetry, nav, magnetics, gravity,
122 byte ascii records with CRLF line breaks, NOAA NGDC

MBIO Data Format ID: 181
Format name: MBF_SAMESURF
Informal Description: SAM Electronics SURF format.
Attributes: variable beams, bathymetry, amplitude, and sidescan,
binary, single files, SAM Electronics (formerly Krupp-Atlas Elektronik).

MBIO Data Format ID: 182
Format name: MBF_HSDS2RAW
Informal Description: STN Atlas raw multibeam format
Attributes: STN Atlas multibeam sonars,
Hydrosweep DS2, Hydrosweep MD,
Fansweep 10, Fansweep 20,
bathymetry, amplitude, and sidescan,
up to 1440 beams and 4096 pixels,
XDR binary, STN Atlas.

MBIO Data Format ID: 183
Format name: MBF_HSDS2LAM
Informal Description: L-DEO HSDS2 processing format
Attributes: STN Atlas multibeam sonars,
Hydrosweep DS2, Hydrosweep MD,
Fansweep 10, Fansweep 20,
bathymetry, amplitude, and sidescan,
up to 1440 beams and 4096 pixels,
XDR binary, L-DEO.

MBIO Data Format ID: 191

Format name: MBF_IMAGE83P
Informal Description: Imagenex DeltaT Multibeam
Attributes: Multibeam, bathymetry, 480 beams, ascii + binary, Imagenex.

MBIO Data Format ID: 192

Format name: MBF_IMAGEMBA
Informal Description: MBARI DeltaT Multibeam
Attributes: Multibeam, bathymetry, 480 beams, ascii + binary, MBARI.

MBIO Data Format ID: 201

Format name: MBF_HYSWEEP1
Informal Description: HYSWEEP multibeam data format
Attributes: Many multibeam sonars,
bathymetry, amplitude
variable beams, ascii, HYPACK.

MBIO Data Format ID: 211

Format name: MBF_XTFB1624
Informal Description: XTF format Benthos Sidescan SIS1624
Attributes: variable pixels, dual frequency sidescan and subbottom,
xtf variant, single files,
low frequency sidescan returned as
survey data, Benthos.

MBIO Data Format ID: 221

Format name: MBF_SWPLSSXI
Informal Description: SEA interferometric sonar vendor intermediate format
Attributes: SEA SWATHplus,
bathymetry and amplitude,
variable beams, binary, SEA.

MBIO Data Format ID: 222

Format name: MBF_SWPLSSXP
Informal Description: SEA interferometric sonar vendor processed data format
Attributes: SEA SWATHplus,
bathymetry and amplitude,
variable beams, binary, SEA.

MBIO Data Format ID: 231

Format name: MBF_3DDEPTH
Informal Description: 3DatDepth prototype binary swath mapping LIDAR format
Attributes: 3DatDepth LIDAR, variable pulses, bathymetry and amplitude,
binary, 3DatDepth.

MBIO Data Format ID: 232

Format name: MBF_3DWISSLR
Informal Description: 3D at Depth Wide Swath Subsea Lidar (WiSSL) raw format
Attributes: 3D at Depth lidar, variable pulses, bathymetry and amplitude,
binary, 3D at Depth.

MBIO Data Format ID: 233

Format name: MBF_3DWISSLP
Informal Description: 3D at Depth Wide Swath Subsea Lidar (WiSSL) processing format
Attributes: 3D at Depth lidar, variable pulses, bathymetry and amplitude,

binary, MBARI.

MBIO Data Format ID: 241

Format name: MBF_WASSPENL

Informal Description: WASSP Multibeam Vendor Format

Attributes: WASSP multibeams,
bathymetry and amplitude,
122 or 244 beams, binary, Electronic Navigation Ltd.

MBIO Data Format ID: 251

Format name: MBF_PHOTGRAM

Informal Description: Example format

Attributes: Name the relevant sensor(s),
what data types are supported
how many beams and pixels, file type (ascii, binary, netCDF), Organization that defined
this format.

MBIO Data Format ID: 261

Format name: MBF_KEMKMALL

Informal Description: Kongsberg multibeam echosounder system kmall datagram format

Attributes: Kongsberg fourth generation multibeam sonars (EM2040, EM712,
EM304, EM124), bathymetry, amplitude, backscatter, variable beams,
binary datagrams, Kongsberg.

The institutional acronyms used above have the following meanings:

L-DEO Lamont-Doherty Earth Observatory
MBARI Monterey Bay Aquarium Research Institute
SIO Scripps Institution of Oceanography
WHOI Woods Hole Oceanographic Institution
URI University of Rhode Island
NRL Naval Research Laboratory
UNB University of New Brunswick
UH University of Hawaii
NOAA National Oceans and Atmospheres Agency
NGDC National Geophysical Data Center
USGS United States Geological Survey
IFREMER French government agency responsible
for operation of French oceanographic
research fleet.

FUNCTION STATUS AND ERROR CODES

All of the **MBIO** functions return an integer status value with the convention that:

status = 1: success
status = 0: failure

All **MBIO** functions also pass an error value argument which gives somewhat more information about problems than the status value. The full suite of possible error values and the associated error messages are:

error = 0: "No error",
error = -1: "Time gap in data",
error = -2: "Data outside specified location
bounds",
error = -3: "Data outside specified time interval",
error = -4: "Ship speed too small",
error = -5: "Comment record",

```

error = -6:    "Neither a data record nor a comment
               record",
error = -7:    "Unintelligible data record",
error = -8:    "Ignore this data",
error = -9:    "No data requested for buffer load",
error = -10:   "Data buffer is full",
error = -11:   "No data was loaded into the buffer",
error = -12:   "Data buffer is empty",
error = -13:   "No data was dumped from the buffer"
error = -14:   "No more survey data records in buffer"
error = -15:   "Data inconsistencies prevented
               inserting data into storage structure"

error = 1:     "Unable to allocate memory,
               initialization failed",
error = 2:     "Unable to open file,
               initialization failed",
error = 3:     "Illegal format identifier,
               initialization failed",
error = 4:     "Read error, probably end-of-file",
error = 5:     "Write error",
error = 6:     "No data in specified location bounds",
error = 7:     "No data in specified time interval",
error = 8:     "Invalid MBIO descriptor",
error = 9:     "Inconsistent usage of MBIO descriptor",
error = 10:    "No pings binned but no fatal error
               - this should not happen!",
error = 11:    "Invalid data record type specified
               for writing",
error = 12:    "Invalid control parameter specified
               by user",
error = 13:    "Invalid buffer id",
error = 14:    "Invalid system id – this should
               not happen!"
error = 15:    "This data file is not in the specified format!"

```

In general, programs should treat negative error values as non-fatal (reading and writing can continue) and positive error values as fatal (the data files should be closed and the program terminated).

FUNCTION VERBOSITY

All of the **MBIO** functions are passed a *verbose* parameter which controls how much debugging information is output to standard error. If *verbose* is 0 or 1, the **MBIO** functions will be silent. If *verbose* is 2, then each function will output information as it is entered and as it returns, along with the parameter values passed into and returned out of the function. Greater values of *verbose* will cause additional information to be output, including values at various stages of data processing during read and write operations. In general, programs using **MBIO** functions should adopt the following verbosity conventions:

```

verbose = 0:    "silent" or near-"silent" execution
verbose = 1:    simple output including
                program name, version
                and simple progress updates
verbose >= 2:  debug mode with copious output
                including every function call
                and status listings

```

INITIALIZATION AND CLOSING FUNCTIONS

```

int mb_read_init(
    int verbose,
    char *file,

```

```

    int format,
    int pings,
    int lonflip,
    double bounds[4],
    int btime_i[7],
    int etime_i[7],
    double speedmin,
    double timegap,
    char **mbio_ptr,
    double *btime_d,
    double *etime_d,
    int *beams_bath,
    int *beams_amp,
    int *pixels_ss,
    int *error);

```

The function **mb_read_init** initializes the data file to be read and the data structures required for reading the data. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

<i>file</i> :	input filename
<i>format</i> :	input MBIO data format id
<i>pings</i> :	ping averaging
<i>lonflip</i> :	longitude flipping
<i>bounds</i> :	location bounds of acceptable data
<i>btime_i</i> :	beginning time of acceptable data
<i>etime_i</i> :	ending time of acceptable data
<i>speedmin</i> :	minimum ship speed of acceptable data
<i>timegap</i> :	maximum time allowed before data gap

The format identifier *format* specifies which of the supported data formats is being read or written; the currently supported formats are listed in the "SUPPORTED FORMATS" section.

The *pings* parameter determines whether and how pings are averaged as part of data input. This parameter is used only by the functions **mb_read** and **mb_get**; **mb_get_all** and **mb_buffer_load** do not average pings. If *pings* = 1, then no ping averaging will be done and each ping read will be returned unaltered by the reading function. If *pings* > 1, then the navigation and beam data for *pings* pings will be read, averaged, and returned as the data for a single ping. If *pings* = 0, then the ping averaging will be varied so that the along-track distance between averaged pings is as close as possible to the across-track distance between beams.

The *lonflip* parameter determines the range in which longitude values are returned:

```

lonflip = -1 : -360 to 0
lonflip = 0 : -180 to 180
lonflip = 1 : 0 to 360

```

The *bounds* array sets the area within which data are desired. Data which lie outside the area specified by *bounds* will be returned with an error by the reading function. The functions **mb_read**, **mb_get** and **mb_get_all** use the *bounds* array; the function **mb_buffer_load** does no location checking.

```

bounds[0] : minimum longitude
bounds[1] : maximum longitude
bounds[2] : minimum latitude
bounds[3] : maximum latitude

```


The *btime_i* array sets the desired beginning time for the data and the *etime_i* array sets the desired ending time. If the beginning time is earlier than the ending time, then any data with a time stamp before the beginning time or after the ending time will be returned with an `MB_ERROR_OUT_TIME` error by the reading function. If the beginning time is after the ending time, then data with time stamps between the ending and beginning time are returned with an error. This scheme allows time windowing outside or inside a specified interval. The functions `mb_read`, `mb_get` and `mb_get_all` use the *btime_i* and *etime_i* arrays; the function `mb_buffer_load` does no time checking.

```

btime[0] : year
btime[1] : month
btime[2] : day
btime[3] : hour
btime[4] : minute
btime[5] : second
btime[6] : microsecond
etime[0] : year
etime[1] : month
etime[2] : day
etime[3] : hour
etime[4] : minute
etime[5] : second
etime[6] : microsecond

```

The *speedmin* parameter sets the minimum acceptable ship speed for the data. If the ship speed associated with any ping is less than *speedmin*, then that data will be returned with an error by the reading function. This is used to eliminate data collected while a ship is on station in a simple way. The functions `mb_read`, `mb_get` and `mb_get_all` use the *speedmin* value; the function `mb_buffer_load` does no speed checking.

The *timegap* parameter sets the threshold at which the time interval between sonar pings (or lidar scans, etc.) is regarded as a data gap. Swath data with ping intervals less than *timegap* are regarded as continuous; if the ping interval exceeds *timegap* then a data gap is declared. Ping averaging is not done across data gaps; an error is returned when time gaps are encountered. The functions `mb_read` and `mb_get` use the *timegap* value; the functions `mb_get_all` and `mb_buffer_load` do no ping averaging and thus have no need to check for time gaps.

The returned values are:

```

mbio_ptr:  pointer to an MBIO descriptor structure
btime_d:   desired beginning time in seconds
           since 1/1/70 00:00:0
etime_d:   desired ending time in seconds
           since 1/1/70 00:00:0
beams_bath: maximum number of bathymetry beams
beams_amp:  maximum number of amplitude beams
pixels_ss:  maximum number of sidescan pixels
error:      error value

```

The structure pointed to by *mbio_ptr* holds the file descriptor and all of the control parameters which govern how the data is read; this pointer must be provided to the functions `mb_read`, `mb_get`, `mb_get_all`, or `mb_buffer_load` to read data. The values *beams_bath*, *beams_amp*, and *pixels_ss* return initial estimates of the maximum number of bathymetry and amplitude beams and sidescan pixels, respectively, that the specified data format may contain. In general, *beams_amp* will either be zero or equal to *beams_bath*. The values *btime_d* and *etime_d* give the desired beginning and end times of the data converted to seconds since 00:00:00 on January 1, 1970; **MBIO** uses these units to calculate time internally.

For most data formats, the initial maximum beam and pixel dimensions will not change. However, a few formats support both variable and arbitrarily large numbers of beams and/or pixels, and so applications

must be capable of handling dynamic changes in the numbers of beams and pixels. The arrays allocated internally in the *mbio_ptr* structure are automatically increased when necessary. However, in order to successfully extract swath data using *mb_get*, *mb_get_all*, *mb_read*, or *mb_extract*, an application must also provide pointers to arrays large enough to hold the current maximum numbers of bathymetry beams, amplitude beams, and sidescan pixels. The function *mb_register_array* allows applications to register array pointers so that these arrays are also dynamically allocated by *MBIO*. Registered arrays will be managed as data are read and then freed when **mb_close** is called.

A status value indicating success or failure is returned; an error value argument passes more detailed information about initialization failures.

```
-----
int mb_write_init(
    int verbose,
    char *file,
    int format,
    char **mbio_ptr,
    int *beams_bath,
    int *beams_amp,
    int *pixels_ss,
    int *error);
```

The function **mb_write_init** initializes the data file to be written and the data structures required for writing the data. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

<i>file</i> :	output filename
<i>format</i> :	output MBIO data format id

The returned values are:

<i>mbio_ptr</i> :	pointer to a structure describing the output file
<i>beams_bath</i> :	maximum number of bathymetry beams
<i>beams_back</i> :	maximum number of backscatter beams
<i>error</i> :	error value

The structure pointed to by *mbio_ptr* holds the output file descriptor; this pointer must be provided to the functions **mb_write**, **mb_put**, **mb_put_all**, or **mb_buffer_dump** to write data. The values *beams_bath*, *beams_amp*, and *pixels_ss* return the maximum number of bathymetry and amplitude beams and sidescan pixels, respectively, that the specified data format may contain. In general, *beams_amp* will either be zero or equal to *beams_bath*. In order to successfully write data, the calling program must provide pointers to arrays large enough to hold *beams_bath* bathymetry values, *beams_amp* amplitude values, and *pixels_ss* sidescan values.

For most data formats, the initial maximum beam and pixel dimensions will not change. However, a few formats support both variable and arbitrarily large numbers of beams and/or pixels, and so applications must be capable of handling dynamic changes in the numbers of beams and pixels. The arrays allocated internally in the *mbio_ptr* structure are automatically increased when necessary. However, in order to successfully insert modified swath data using *mb_put*, *mb_put_all*, or *mb_insert*, an application must also provide pointers to arrays large enough to hold the current maximum numbers of bathymetry beams, amplitude beams, and sidescan pixels. The function *mb_register_array* allows applications to register array pointers so that these arrays are also dynamically allocated by *MBIO*. Registered arrays will be managed as data are read and written and then freed when **mb_close** is called.

A status value indicating success or failure is returned; an error value argument passes more detailed information about initialization failures.

```
-----
int mb_register_array(
    int verbose,
    void *mbio_ptr,
    int type,
    int size,
    void **handle,
    int *error)
```

Registers an array pointer **handle* so that the size of the allocated array can be managed dynamically by **MBIO**. Note that the location ***handle* of the array pointer must be supplied, not the pointer value **handle*. The pointer value **handle* should initially be *NULL*. The type value indicates whether this array is to be dimensioned according to the maximum number of bathymetry beams (*type = 1*), amplitude beams (*type = 2*), or sidescan pixels (*type = 3*). The size value indicates the size of each element array in bytes (e.g. a char array has *size = 1*, a short array has *size = 2*, an int array or a float array have *size = 4*, and a double array has *size = 8*). The array is associated with the **MBIO** descriptor *mbio_ptr*, and is freed when *mb_close* is called for this particular *mbio_ptr*.

```
-----
int mb_close(
    int verbose,
    char *mbio_ptr,
    int *error)
```

Closes the data file listed in the **MBIO** descriptor pointed to by *mbio_ptr* and releases all specially allocated memory, including all application arrays registered using **mb_register_array**. The verbose value controls the standard error output verbosity of the function. A status value indicating success or failure is returned; an error value argument passes more detailed information about failures.

LEVEL 1 FUNCTIONS

```
-----
int mb_read(
    int verbose,
    char *mbio_ptr,
    int *kind,
    int *pings,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    double *distance,
    double *altitude,
    double *sonardepth,
    int *nbath,
    int *namp,
    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathlon,
```

```

double *bathlat,
double *ss,
double *sslon,
double *sslats,
char *comment,
int *error);

```

The function **mb_read** reads, processes, and returns sonar data according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. A number of different data record types are recognized by **MB-System**, but **mb_read()** only returns survey and comment data records. The *kind* value indicates which type of record has been read. The data is in the form of bathymetry, amplitude, and sidescan values combined with the longitude and latitude locations of the bathymetry and sidescan measurements (amplitudes are coincident with the bathymetry).

The return values are:

<i>kind:</i>	kind of data record read
	1 survey data
	2 comment
	>=3 other data that cannot be passed by mb_read
<i>pings:</i>	number of pings averaged to give current data
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d:</i>	time of current ping in seconds since 1/1/70 00:00:00
<i>navlon:</i>	longitude
<i>navlat:</i>	latitude
<i>speed:</i>	ship speed in km/s
<i>heading:</i>	ship heading in degrees
<i>distance:</i>	distance along shiptrack since last ping in km
<i>altitude:</i>	altitude of sonar above seafloor in m
<i>sonardepth:</i>	depth of sonar in m
<i>nbath:</i>	number of bathymetry values
<i>namp:</i>	number of amplitude values
<i>nss:</i>	number of sidescan values
<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters
<i>amp:</i>	array of amplitude values in unknown units
<i>bathlon:</i>	array of longitude values corresponding to bathymetry
<i>bathlat:</i>	array of latitude values corresponding to bathymetry
<i>ss:</i>	array of sidescan values in unknown units
<i>sslon:</i>	array of longitude values corresponding to sidescan
<i>sslats:</i>	array of latitude values corresponding

comment: to sidescan
 comment string
error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about read failures.

```

-----
int mb_get(
    int verbose,
    char *mbio_ptr,
    int *kind,
    int *pings,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    double *distance,
    double *altitude,
    double *sonardepth,
    int *nbath,
    int *namp,
    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathacrosstrack,
    double *bathalongtrack,
    double *ss,
    double *ssacrosstrack,
    double *ssalongtrack,
    char *comment,
    int *error);
  
```

The function **mb_get** reads, processes, and returns sonar data according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. A number of different data record types are recognized by **MB-System**, but **mb_get()** only returns survey and comment data records. The *kind* value indicates which type of record has been read. The data is in the form of bathymetry, amplitude, and sidescan values combined with the acrosstrack and alongtrack distances relative to the navigation of the bathymetry and sidescan measurements (amplitudes are coincident with the bathymetry values).

The return values are:

<i>kind:</i>	kind of data record read
	1 survey data
	2 comment
	>=3 other data that cannot be passed by mb_get
<i>pings:</i>	number of pings averaged to give current data
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month

	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d</i> :	time of current ping in seconds since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>distance</i> :	distance along shiptrack since last ping in km
<i>altitude</i> :	altitude of sonar above seafloor in m
<i>sonardepth</i> :	depth of sonar in m
<i>nbath</i> :	number of bathymetry values
<i>namp</i> :	number of amplitude values
<i>nss</i> :	number of sidescan values
<i>beamflag</i> :	array of bathymetry flags
<i>bath</i> :	array of bathymetry values in meters
<i>amp</i> :	array of amplitude values in unknown units
<i>bathacrosstrack</i> :	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack</i> :	array of alongtrack distances in meters corresponding to bathymetry
<i>ss</i> :	array of sidescan values in unknown units
<i>ssacrosstrack</i> :	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack</i> :	array of alongtrack distances in meters corresponding to sidescan
<i>comment</i> :	comment string
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about read failures.

LEVEL 2 FUNCTIONS

```
int mb_read_ping(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int *error);
```

The function **mb_read_ping** reads and returns sonar data according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The data is returned one record at a time; no averaging is performed. A pointer to a data structure containing all of the data read is returned as *store_ptr*; the form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record has been read.

The return values are:

store_ptr: pointer to complete data structure

<i>kind:</i>	kind of data record read
	1 survey data
	2 comment
	3 header
	4 calibrate
	5 mean sound speed
	6 SVP
	7 standby
	8 nav source
	9 parameter
	10 start
	11 stop
	12 nav
	13 run parameter
	14 clock
	15 tide
	16 height
	17 heading
	18 attitude
	19 ssv
	20 angle
	21 event
	22 history
	23 summary
	24 processing parameters
	25 sensor parameters
	26 navigation error
	27 uninterpretable line
<i>error:</i>	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about read failures.

```
-----
int mb_write_ping(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *error);
```

The function **mb_write_ping** writes sonar data to the file listed in the **MBIO** descriptor pointed to by *MBIO_ptr*. The *verbose* value controls the standard error output verbosity of the function. A pointer to a data structure containing all of the data read is passed as *store_ptr*; the form of the data structure is determined by the sonar system associated with the format of the data being written. The values to be output are:

store_ptr: pointer to complete data structure

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about write failures.

```
int mb_get_store(
    int verbose,
    char *mbio_ptr,
    char **store_ptr,
    int *error);
```

The function **mb_get_store()** returns a pointer *store_ptr* to the data storage structure associated with a particular **MBIO** descriptor *mbio_ptr*. The **mb_read_init()** and **mb_write_init()** functions both allocate one of these internal storage structures. The form of the data structure is determined by the sonar system associated with the format of the data being written. Storage structure pointers must be passed to level two **MBIO** functions such as **mb_write_ping()** and **mb_insert()**. The *verbose* value controls the standard error output verbosity of the function.

The return values are:

<i>store_ptr</i> :	pointer to complete data structure
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_get_all(
    int verbose,
    char *mbio_ptr,
    char **store_ptr,
    int *kind,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    double *distance,
    double *altitude,
    double *sonardepth,
    int *nbath,
    int *namp,
    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathacrosstrack,
    double *bathalongtrack,
    double *ss,
    double *ssacrosstrack,
    double *ssalongtrack,
    char *comment,
    int *error);
```

The function **mb_get_all** reads and returns sonar data according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The data is returned one record at a time; no averaging is performed. A pointer to a data structure containing all of the data read is returned as *store_ptr*; the form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-**

System; the *kind* value indicates which type of record has been read. Additional data is returned if the data record is survey data (navigation, bathymetry, amplitude, and sidescan), navigation data (navigation only), or comment data (comment only).

The return values are:

<i>store_ptr:</i>	pointer to complete data structure
<i>kind:</i>	kind of data record read
	1 survey data
	2 comment
	3 header
	4 calibrate
	5 mean sound speed
	6 SVP
	7 standby
	8 nav source
	9 parameter
	10 start
	11 stop
	12 nav
	13 run parameter
	14 clock
	15 tide
	16 height
	17 heading
	18 attitude
	19 ssv
	20 angle
	21 event
	22 history
	23 summary
	24 processing parameters
	25 sensor parameters
	26 navigation error
	27 uninterpretable line
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d:</i>	time of current ping in seconds
	since 1/1/70 00:00:00
<i>navlon:</i>	longitude
<i>navlat:</i>	latitude
<i>speed:</i>	ship speed in km/s
<i>heading:</i>	ship heading in degrees
<i>distance:</i>	distance along shiptrack since last ping in km
<i>altitude:</i>	altitude of sonar above seafloor in m
<i>sonardepth:</i>	depth of sonar in m
<i>nbath:</i>	number of bathymetry values

<i>namp:</i>	number of amplitude values
<i>nss:</i>	number of sidescan values
<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters
<i>amp:</i>	array of amplitude values in unknown units
<i>bathacrosstrack:</i>	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack:</i>	array of alongtrack distances in meters corresponding to bathymetry
<i>ss:</i>	array of sidescan values in unknown units
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan
<i>comment:</i>	comment string
<i>error:</i>	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about read failures.

```

-----
int mb_put_all(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int usevalues,
    int kind,
    int time_i[7],
    double time_d,
    double navlon,
    double navlat,
    double speed,
    double heading,
    int nbath,
    int namp,
    int nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathacrosstrack,
    double *bathalongtrack,
    double *ss,
    double *ssacrosstrack,
    double *ssalongtrack,
    char *comment,
    int *error);

```

The function **mb_put_all** writes sonar data to the file listed in the **MBIO** descriptor pointed to by *MBIO_ptr*. The *verbose* value controls the standard error output verbosity of the function. A pointer to a data structure containing all of the data read is passed as *store_ptr*; the form of the data structure is determined by the sonar system associated with the format of the data being written. Additional data is passed if the data record is survey data (navigation, bathymetry, amplitude, and sidescan), navigation data (navigation only), or comment data (comment only). If the *usevalues* flag is set to 1, then the passed values will be inserted in the data structure pointed to by *store_ptr* before the data is written. If the *usevalues* flag is set to

0, the data structure pointed to by *store_ptr* will be written without modification. The values to be output are:

<i>store_ptr</i> :	pointer to complete data structure	
<i>usevalues</i> :	flag controlling use of data passed by value	
	0	do not insert into data structure before writing the data
	1	insert into data structure before writing the data
<i>kind</i> :	kind of data record to be written	
	1	survey data
	2	comment
	3	header
	4	calibrate
	5	mean sound speed
	6	SVP
	7	standby
	8	nav source
	9	parameter
	10	start
	11	stop
	12	nav
	13	run parameter
	14	clock
	15	tide
	16	height
	17	heading
	18	attitude
	19	ssv
	20	angle
	21	event
	22	history
	23	summary
	24	processing parameters
	25	sensor parameters
	26	navigation error
	27	uninterpretable line
<i>time_i</i> :	time of current ping (used if <i>time_i</i> [0] != 0)	
	<i>time_i</i> [0]: year	
	<i>time_i</i> [1]: month	
	<i>time_i</i> [2]: day	
	<i>time_i</i> [3]: hour	
	<i>time_i</i> [4]: minute	
	<i>time_i</i> [5]: second	
	<i>time_i</i> [6]: microsecond	
<i>time_d</i> :	time of current ping in seconds since	
	1/1/70 00:00:00 (used if <i>time_i</i> [0] = 0)	
<i>navlon</i> :	longitude	
<i>navlat</i> :	latitude	
<i>speed</i> :	ship speed in km/s	
<i>heading</i> :	ship heading in degrees	
<i>nbath</i> :	number of bathymetry values	
<i>namp</i> :	number of amplitude values	
<i>nss</i> :	number of sidescan values	

<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters
<i>amp:</i>	array of amplitude values in unknown units
<i>bathacrosstrack:</i>	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack:</i>	array of alongtrack distances in meters corresponding to bathymetry
<i>ss:</i>	array of sidescan values in unknown units
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan
<i>comment:</i>	comment string

The return values are:

<i>error:</i>	error value
---------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about write failures.

```
-----
int mb_put_comment(
    int verbose,
    char *mbio_ptr,
    char *comment,
    int *error);
```

The function **mb_put_comment** writes a comment to the file listed in the **MBIO** descriptor pointed to by *MBIO_ptr*. The *verbose* value controls the standard error output verbosity of the function. The data is in the form of a null terminated string. The maximum length of comments varies with different data formats. In general individual comments should be less than 80 characters long to insure compatibility with all formats. The values to be output are:

<i>comment:</i>	comment string
-----------------	----------------

The return values are:

<i>error:</i>	error value
---------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about write failures.

```
-----
int mb_extract(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    int *nbath,
    int *namp,
```

```

    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathacrosstrack,
    double *bathalongtrack,
    double *ss,
    double *ssacrosstrack,
    double *ssalongtrack,
    char *comment,
    int *error);

```

The function **mb_extract** extracts sonar data from the structure pointed to by **store_ptr* according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is stored in **store_ptr*. Additional data is returned if the data record is survey data (navigation, bathymetry, amplitude, and sidescan), navigation data (navigation only), or comment data (comment only).

The return values are:

<i>kind:</i>	kind of data record read
	1 survey data
	2 comment
	12 navigation
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d:</i>	time of current ping in seconds
	since 1/1/70 00:00:00
<i>navlon:</i>	longitude
<i>navlat:</i>	latitude
<i>speed:</i>	ship speed in km/s
<i>heading:</i>	ship heading in degrees
<i>distance:</i>	distance along shiptrack since last ping in km
<i>altitude:</i>	altitude of sonar above seafloor in m
<i>sonardepth:</i>	depth of sonar in m
<i>nbath:</i>	number of bathymetry values
<i>namp:</i>	number of amplitude values
<i>nss:</i>	number of sidescan values
<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters
<i>amp:</i>	array of amplitude values in unknown units
<i>bathacrosstrack:</i>	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack:</i>	array of alongtrack distances in meters corresponding to bathymetry

<i>ss:</i>	array of sidescan values in unknown units
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan
<i>comment:</i>	comment string
<i>error:</i>	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about extract failures.

```

-----
int mb_extract_lonlat(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    int *nbath,
    int *namp,
    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathlon,
    double *bathlat,
    double *ss,
    double *sslon,
    double *sslat,
    char *comment,
    int *error);

```

The function **mb_extract_lonlat** extracts sonar data from the structure pointed to by *store_ptr* according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is stored in *store_ptr*. Additional data is returned if the data record is survey data (navigation, bathymetry, amplitude, and sidescan), navigation data (navigation only), or comment data (comment only).

The return values are:

<i>kind:</i>	kind of data record read
	1 survey data
	2 comment
	12 navigation
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day

	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d</i> :	time of current ping in seconds since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>distance</i> :	distance along shiptrack since last ping in km
<i>altitude</i> :	altitude of sonar above seafloor in m
<i>sonardepth</i> :	depth of sonar in m
<i>nbath</i> :	number of bathymetry values
<i>namp</i> :	number of amplitude values
<i>nss</i> :	number of sidescan values
<i>beamflag</i> :	array of bathymetry flags
<i>bath</i> :	array of bathymetry values in meters
<i>amp</i> :	array of amplitude values in unknown units
<i>bathlon</i> :	array of longitude positions in degrees corresponding to bathymetry
<i>bathlat</i> :	array of latitude positions in degrees corresponding to bathymetry
<i>ss</i> :	array of sidescan values in unknown units
<i>sslon</i> :	array of longitude positions in degrees corresponding to sidescan
<i>sslats</i> :	array of latitude positions in degrees corresponding to sidescan
<i>comment</i> :	comment string
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about extract failures.

```
-----
int mb_insert(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int kind,
    int time_i[7],
    double time_d,
    double navlon,
    double navlat,
    double speed,
    double heading,
    int nbath,
    int namp,
    int nss,
    char *beamflag,
    double *bath,
    double *amp,
```

```

double *bathacrosstrack,
double *bathalongtrack,
double *ss,
double *ssacrosstrack,
double *ssalongtrack,
char *comment,
int *error);

```

The function **mb_insert** inserts sonar data into the structure pointed to by **store_ptr* according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is to be stored in **store_ptr*. Data will be inserted only if the data record is survey data (navigation, bathymetry, amplitude, and sidescan), navigation data (navigation only), or comment data (comment only). The values to be inserted are:

<i>kind:</i>	kind of data record inserted
	1 survey data
	2 comment
	12 navigation
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d:</i>	time of current ping in seconds
	since 1/1/70 00:00:00
<i>navlon:</i>	longitude
<i>navlat:</i>	latitude
<i>speed:</i>	ship speed in km/s
<i>heading:</i>	ship heading in degrees
<i>distance:</i>	distance along shiptrack since last ping in km
<i>altitude:</i>	altitude of sonar above seafloor in m
<i>sonardepth:</i>	depth of sonar in m
<i>nbath:</i>	number of bathymetry values
<i>namp:</i>	number of amplitude values
<i>nss:</i>	number of sidescan values
<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters
<i>amp:</i>	array of amplitude values in unknown units
<i>bathacrosstrack:</i>	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack:</i>	array of alongtrack distances in meters corresponding to bathymetry
<i>ss:</i>	array of sidescan values in unknown units
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan
<i>comment:</i>	comment string

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about insert failures.

```
-----
int mb_extract_nav(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    double *draft,
    double *roll,
    double *pitch,
    double *heave,
    int *error);
```

The function **mb_extract_nav** extracts navigation data from the structure pointed to by **store_ptr* according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is stored in **store_ptr*. Navigation data is returned if the data record is survey data (navigation, bathymetry, amplitude, and sidescan) or navigation data (navigation only).

The return values are:

<i>kind</i> :	kind of data record read
	1 survey data
	12 navigation
<i>time_i</i> :	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d</i> :	time of current ping in seconds
	since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>draft</i> :	sonar depth in meters
<i>roll</i> :	sonar roll in degrees
<i>pitch</i> :	sonar pitch in degrees
<i>heave</i> :	sonar heave in meters

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about extract failures.

```
-----
int mb_insert_nav(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int time_i[7],
    double time_d,
    double navlon,
    double navlat,
    double speed,
    double heading,
    double draft,
    double roll,
    double pitch,
    double heave,
    int *error);
```

The function **mb_insert_nav** inserts navigation data into the structure pointed to by **store_ptr* according to the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is to be stored in **store_ptr*. Data will be inserted only if the data record is survey data (navigation, bathymetry, amplitude, and sidescan), or navigation data (navigation only). The values to be inserted are:

<i>time_i</i> :	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d</i> :	time of current ping in seconds
	since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>draft</i> :	sonar depth in meters
<i>roll</i> :	sonar roll in degrees
<i>pitch</i> :	sonar pitch in degrees
<i>heave</i> :	sonar heave in meters

The return values are:

<i>error</i> :	error value
----------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about insert failures.

```
-----
int mb_extract_altitude(
```

```

    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    double *transducer_depth,
    double *altitude,
    int *error);

```

The function **mb_extract_altitude** extracts the sonar transducer depth (**transducer_depth**) below the sea surface and the sonar transducer **altitude** above the seafloor according to the **MBIO** descriptor pointed to by *mbio_ptr*. This function is not defined for all data formats. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is stored in *store_ptr*. These data are returned only if the data record is survey data. These values are useful for sidescan processing applications. Both transducer depths and altitudes are reported in meters.

The return values are:

<i>kind</i> :	kind of data record read (error if not survey data):
	1 survey data
<i>transducer_depth</i> :	depth of sonar in meters
<i>altitude</i> :	altitude of sonar above seafloor in meters.
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data extraction failures.

```

int mb_insert_altitude(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    double transducer_depth,
    double altitude,
    int *error);

```

The function **mb_insert_altitude** inserts sonar depth and altitude data into the structure pointed to by *store_ptr* according to the **MBIO** descriptor pointed to by *mbio_ptr*. This function is not defined for all data formats. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**. Data will be inserted only if the data record is survey data (navigation, bathymetry, amplitude, and sidescan). The values to be inserted are:

<i>transducer_depth</i> :	depth of sonar in meters
<i>altitude</i> :	altitude of sonar in meters

The return values are:

<i>error</i> :	error value
----------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about insert failures.

```
int mb_extract_svp(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int *nsvp,
    double *depth,
    double *velocity,
    int *error);
```

The function **mb_extract_svp** extracts a water sound velocity profile according to the **MBIO** descriptor pointed to by *mbio_ptr*. This function is not defined for all data formats. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**; the *kind* value indicates which type of record is stored in *store_ptr*. These data are returned only if the data record is a sound velocity profile record. These values are useful for calculating bathymetry from travel times and beam angles.

The return values are:

<i>kind</i> :	kind of data record read (error if not SVP data):
	6 SVP data
<i>nsvp</i> :	number of depth and sound speed data in the profile
<i>depth</i> :	array of depths in meters
<i>velocity</i> :	array of sound speeds in m/sec
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data extraction failures.

```
-----
int mb_insert_svp(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int nsvp,
    double *depth,
    double *velocity,
    int *error);
```

The function **mb_insert_svp** inserts a water sound velocity profile according to the **MBIO** descriptor pointed to by *mbio_ptr*. This function is not defined for all data formats. The *verbose* value controls the standard error output verbosity of the function. The form of the data structure is determined by the sonar system associated with the format of the data being read. A number of different data record types are recognized by **MB-System**. These data are inserted only if the data record is a sound velocity profile record. These values are useful for calculating bathymetry from travel times and beam angles. The inserted values are:

<i>nsvp</i> :	number of depth and sound speed data in the profile
<i>depth</i> :	array of depths in meters
<i>velocity</i> :	array of sound speeds in m/sec

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data insertion failures.

```
-----
int mb_ttimes(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int *nbeams,
    double *ttimes,
    double *angles,
    double *angles_forward,
    double *angles_null,
    double *heave,
    double *alongtrack_offset,
    double *draft,
    double *ssv,
    int *error);
```

The function **mb_ttimes** extracts travel times and beam angles from a sonar-specific data structure pointed to by *store_ptr*. These values are used for calculating swath bathymetry. The *verbose* value controls the standard error output verbosity of the function. The coordinates of the beam angles can be a bit confusing. The angles are returned in "takeoff angle coordinates" appropriate for raytracing. The array *angles* contains the angle from vertical and the array *angles_forward* contains the angle from across-track. This coordinate system is distinct from the roll-pitch coordinates appropriate for correcting roll and pitch values. A description of these relevant coordinate systems is given below. The *angles_null* array contains the effective sonar array orientation for each beam. The *angles_null* array may be used to correct beam angles using Snell's law if the *ssv* is changed. The *angles_null* values reflect the sonar configuration. For example, some multibeam sonars have a flat transducer array, and so the *angles_null* array consists of *nbeams* zero values. Other multibeams have circular arrays so that the *angles_null* values equal the *angles* values. The *along-track_offset* array accommodates sonars which report multiple pings in a single survey record; each ping occurs at a different position along the shiptrack, producing alongtrack offsets relative to the navigation for some beam values. The sum of the *draft* value and the *heave* array values gives the depth of the sonar for each beam. For hull mounted installations the *draft* value is generally static but the *heave* values vary with time. For towed sonars the *draft* varies with time and the *heave* values are typically zero. The *ssv* value gives the water sound velocity at the sonar array.

The return values are:

<i>kind</i> :	kind of data record read (error if not survey data): 1 survey data
<i>nbeams</i> :	number of beams
<i>ttimes</i> :	array of two-way travel times in seconds
<i>angles</i> :	array of angles from vertical in degrees
<i>angles_forward</i> :	array of angles from across-track in degrees
<i>angles_null</i> :	array of sonar array orientation in degrees
<i>heave</i> :	array of heave values for each beam in meters
<i>alongtrack_offset</i> :	array of alongtrack distance offsets for each beam in meters
<i>draft</i> :	draft of sonar in meters
<i>ssv</i> :	water sound velocity at sonar in m/seconds

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data extraction failures.

```
-----
int mb_detects(
    int verbose,
    void *mbio_ptr,
    void *store_ptr,
    int *kind,
    int *nbeams,
    int *detects,
    int *error);
```

The function **mb_detects** extracts beam bottom detect types from a sonar-specific data structure pointed to by *store_ptr*. These values indicate whether the depth value associated with a particular beam *i* derived from an amplitude detect (e.g. *detects*[*i*] = 1), a phase detect (e.g. *detects*[*i*] = 2), or the algorithm is unknown (e.g. *detects*[*i*] = 0). The *verbose* value controls the standard error output verbosity of the function.

The return values are:

<i>kind:</i>	kind of data record read (error if not survey data):	1	survey data
<i>nbeams:</i>	number of beams		
<i>detects:</i>	array of <i>nbeams</i> bottom detect algorithm flags	0 = unknown	1 =
			amplitude detect
			2 = phase detect
<i>error:</i>	error value		

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data extraction failures. This functionality is available for only a subset of the supported sonars. If the corresponding low level routine is undefined, **error* will be set to MB_ERROR_BAD_SYSTEM (14).

```
-----
int mb_gains(
    int verbose,
    void *mbio_ptr,
    void *store_ptr,
    int *kind,
    double *transmit_gain,
    double *pulse_length,
    double *receive_gain,
    int *error);
```

The function **mb_gains** extracts the most basic gain settings from a sonar-specific data structure pointed to by *store_ptr*. In many cases, sonars have more complicated gain functions, particularly with respect to the receiver TVG function. In those cases, the receive gain returned here refers to the constant gain setting and does not include any TVG parameters. The *verbose* value controls the standard error output verbosity of the function.

The return values are:

kind: kind of data record read (error

```

                                if not survey data):
                                    1          survey data
    transmit_gain:    transmit gain (dB)
    pulse_length:    transmit pulse length (sec)
    receive_gain:    receive gain (dB)
    error:            error value

```

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data extraction failures. This functionality is available for only a subset of the supported sonars. If the corresponding low level routine is undefined, **error* will be set to MB_ERROR_BAD_SYSTEM (14).

```

-----
int mb_extract_rawss(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int *kind,
    int *nrawss,
    double *rawss,
    double *rawssacrosstrack,
    double *rawssalongtrack,
    int *error);

```

This function has not yet been implemented for any data format. The notion is that since some formats carry both "raw" and "processed" sidescan imagery, there should be functions to extract and insert the "raw" sidescan. Given that the meaning of "raw" sidescan varies greatly among sonars, the processing one might apply to the data will depend on the sonar source. The definition of **mb_extract_rawss** may well change when we actually implement it.

```

-----
int mb_insert_rawss(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    int nrawss,
    double *rawss,
    double *rawssacrosstrack,
    double *rawssalongtrack,
    int *error);

```

This function has not yet been implemented for any data format. The notion is that since some formats carry both "raw" and "processed" sidescan imagery, there should be functions to extract and insert the "raw" sidescan. Given that the meaning of "raw" sidescan varies greatly among sonars, the processing one might apply to the data will depend on the sonar source. The definition of **mb_insert_rawss** may well change when we actually implement it.

```

-----
int mb_copyrecord(
    int verbose,
    char *mbio_ptr,
    char *store_ptr,
    char *copy_ptr,
    int *error);

```

The function **mb_copyrecord** copies the sonar-specific data structure pointed to by *store_ptr* into the data structure pointed to by **copy_ptr*. The data structures must already have been allocated.

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about data copy failures.

LEVEL 3 FUNCTIONS

```
int mb_buffer_init(
    int verbose,
    char **buff_ptr,
    int *error);
```

The function **mb_buffer_init** initializes the data structures required for buffered i/o. A pointer to the buffer data structure is returned as **buff_ptr*. The *verbose* value controls the standard error output verbosity of the function.

The return values are:

**buff_ptr*: pointer to buffer structure
error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer initialization failures.

```
-----
int mb_buffer_close(
    int verbose,
    char **buff_ptr,
    char *mbio_ptr,
    int *error);
```

The function **mb_buffer_close** releases all memory allocated for buffered i/o, including the structure pointed to by **buff_ptr*. The *verbose* value controls the standard error output verbosity of the function.

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer deallocation failures.

```
-----
int mb_buffer_load(
    int verbose,
    char *buff_ptr, char *mbio_ptr,
    int nwant,
    int *nload,
    int *nbuff,
    int *error);
```

The function **mb_buffer_load** loads data into the buffer pointed to by *buff_ptr* from the input file initialized in the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

nwant: The number of data records desired
in the buffer.

The returned values are:

nload: The number of data records loaded into the buffer.
nbuff: The total number of data records in the buffer after loading.
error: error value

The buffer may already contain data records when the **mb_buffer_load** call is made; if the number of previously loaded records is less than *nwant*, the function will attempt to read and load records until a total of *nwant* records are loaded. The *nload* value is the number of data records loaded during the current function call, and the *nbuff* value is the number of data records in the buffer at the completion of the **mb_buffer_load** call. A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer deallocation failures.

```
-----
int mb_buffer_dump(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    char *ombio_ptr,
    int nhold,
    int *ndump,
    int *nbuff,
    int *error);
```

The function **mb_buffer_dump** dumps data from the buffer pointed to by **buff_ptr* into the output file initialized in the **MBIO** descriptor pointed to by *ombio_ptr*. The data in the buffer were read from the input file initialized in the **MBIO** descriptor pointed to by *mbio_ptr*. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

nhold: The number of data records desired to be held
in the buffer.

The returned values are:

nload: The number of data records dumped from the buffer.
nbuff: The total number of data records in the buffer after dumping.
error: error value

If the number of loaded records is more than *nhold*, the function will attempt to write out records from the beginning of the buffer until *nhold* records are left in the buffer. The *ndump* value is the number of data records dumped during the current function call, and the *nbuff* value is the number of data records in the buffer at the completion of the **mb_buffer_dump** call. A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer deallocation failures.

```
-----
int mb_buffer_clear(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int nhold,
    int *ndump,
```

```
int *nbuff,
int *error);
```

The function **mb_buffer_clear** removes data from the buffer pointed to by **buff_ptr* without writing those data records to an output file. An **MBIO** descriptor pointed to by *mbio_ptr* is still required, and generally represents the **MBIO** descriptor used to read and load the data originally. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

nwant: The number of data records desired to be held
in the buffer.

The returned values are:

nload: The number of data records cleared from the buffer.
nbuff: The total number of data records in the buffer after dumping.
error: error value

If the number of loaded records is more than *nhold*, the function will attempt to clear out records from the beginning of the buffer until *nhold* records are left in the buffer. The *ndump* value is the number of data records cleared during the current function call, and the *nbuff* value is the number of data records in the buffer at the completion of the **mb_buffer_dump** call. A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer deallocation failures.

```
-----
int mb_buffer_info(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int id,
    int *system,
    int *kind,
    int *error);
```

The function **mb_buffer_clear** removes data from the buffer pointed to by **buff_ptr* without writing those data records to an output file. An **MBIO** descriptor pointed to by *mbio_ptr* is still required, and generally represents the **MBIO** descriptor used to read and load the data originally. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

nwant: The number of data records desired to be held
in the buffer.

The returned values are:

nload: The number of data records cleared from the buffer.
nbuff: The total number of data records in the buffer after dumping.
error: error value

If the number of loaded records is more than *nhold*, the function will attempt to clear out records from the beginning of the buffer until *nhold* records are left in the buffer. The *ndump* value is the number of data records cleared during the current function call, and the *nbuff* value is the number of data records in the buffer at the completion of the **mb_buffer_dump** call. A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer deallocation failures.

```

int mb_buffer_get_next_data(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int start,
    int *id,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    int *nbath,
    int *namp,
    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathacrosstrack,
    double *bathalongtrack,
    double *ss,
    double *ssacrosstrack,
    double *ssalongtrack,
    int *error);

```

The function **mb_buffer_get_next_data** searches for the next survey data record in the buffer, beginning at buffer index *start*. Since buffer indexes begin at 0, the first call to **mb_buffer_get_next_data** should have *start* = 0. If a survey data record is found at or beyond *start*, **mb_buffer_get_next_data** returns the buffer index of that record in *id*. Data is also returned in the forms of bathymetry, amplitude, and sidescan survey data. No comments or other non-survey data records are returned. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

start: The buffer index at which to start searching for a survey data record.

The returned values are:

id: The buffer index of the first survey data record at or after *start*.

time_i: time of current ping
time_i[0]: year
time_i[1]: month
time_i[2]: day
time_i[3]: hour
time_i[4]: minute
time_i[5]: second
time_i[6]: microsecond

time_d: time of current ping in seconds since 1/1/70 00:00:00

navlon: longitude

navlat: latitude

speed: ship speed in km/s

heading: ship heading in degrees

nbath: number of bathymetry values

<i>namp:</i>	number of amplitude values		
<i>nss:</i>	number of sidescan values	<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters		
<i>amp:</i>	array of amplitude values in unknown units		
<i>bathacrosstrack:</i>	array of acrosstrack distances in meters corresponding to bathymetry		
<i>bathalongtrack:</i>	array of alongtrack distances in meters corresponding to bathymetry		
<i>ss:</i>	array of sidescan values in unknown units		
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan		
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan		
<i>error:</i>	error value		

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures. The most common error occurs when no more survey data records remain to be found in the buffer; in this case, *error* = -14.

```
-----
int mb_buffer_extract(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int id,
    int *kind,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    int *nbath,
    int *namp,
    int *nss,
    char *beamflag,
    double *bath,
    double *amp,
    double *bathacrosstrack,
    double *bathalongtrack,
    double *ss,
    double *ssacrosstrack,
    double *ssalongtrack,
    char *comment,
    int *error);
```

The function **mb_buffer_extract** extracts and returns a subset of the data in a buffer record. The *verbose* value controls the standard error output verbosity of the function. The buffer record is specified with the buffer index *id*. The data is either in the form of bathymetry, amplitude, and sidescan survey data or a comment string.

The input control parameters have the following significance:

id: The buffer index of the data

record to extract.

The returned values are:

<i>kind:</i>	kind of data record extracted
	1 survey data
	2 comment
	>=3 other data that cannot be passed by mb_buffer_extract
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d:</i>	time of current ping in seconds since 1/1/70 00:00:00
<i>navlon:</i>	longitude
<i>navlat:</i>	latitude
<i>speed:</i>	ship speed in km/s
<i>heading:</i>	ship heading in degrees
<i>nbath:</i>	number of bathymetry values
<i>namp:</i>	number of amplitude values
<i>nss:</i>	number of sidescan values
<i>beamflag:</i>	array of bathymetry flags
<i>bath:</i>	array of bathymetry values in meters
<i>amp:</i>	array of amplitude values in unknown units
<i>bathacrosstrack:</i>	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack:</i>	array of alongtrack distances in meters corresponding to bathymetry
<i>ss:</i>	array of sidescan values in unknown units
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan
<i>comment:</i>	comment string
<i>error:</i>	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about extract failures.

```
-----
int mb_buffer_insert(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int id,
    int time_i[7],
    double time_d,
    double navlon,
    double navlat,
    double speed,
```

```

double heading,
int nbath,
int namp,
int nss,
char *beamflag,
double *bath,
double *amp,
double *bathacrosstrack,
double *bathalongtrack,
double *ss,
double *ssacrosstrack,
double *ssalongtrack,
char *comment,
int *error);

```

The function **mb_buffer_insert** inserts data into a buffer record, replacing a subset of the original values. The *verbose* value controls the standard error output verbosity of the function. The buffer record is specified with the buffer index *id*. The data is either in the form of bathymetry, amplitude, and sidescan survey data or a comment string.

The input control parameters have the following significance:

id: The buffer index of the data record to insert.

The returned values are:

<i>kind</i> :	kind of data record inserted
	1 survey data
	2 comment
	>=3 other data that cannot be passed by mb_buffer_insert
<i>time_i</i> :	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d</i> :	time of current ping in seconds since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>nbath</i> :	number of bathymetry values
<i>namp</i> :	number of amplitude values
<i>nss</i> :	number of sidescan values
<i>beamflag</i> :	array of bathymetry flags
<i>bath</i> :	array of bathymetry values in meters
<i>amp</i> :	array of amplitude values in unknown units
<i>bathacrosstrack</i> :	array of acrosstrack distances in meters corresponding to bathymetry
<i>bathalongtrack</i> :	array of alongtrack distances in meters corresponding to bathymetry

<i>ss:</i>	array of sidescan values in unknown units
<i>ssacrosstrack:</i>	array of acrosstrack distances in meters corresponding to sidescan
<i>ssalongtrack:</i>	array of alongtrack distances in meters corresponding to sidescan
<i>comment:</i>	comment string

The returned values are:

<i>error:</i>	error value
---------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about insert failures.

```

-----
int mb_buffer_get_next_nav(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int start,
    int *id,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    double *draft,
    double *roll,
    double *pitch,
    double *heave,
    int *error);

```

The function **mb_buffer_get_next_nav** searches for the next survey data record in the buffer, beginning at buffer index *start*. Since buffer indexes begin at 0, the first call to **mb_buffer_get_next_nav** should have *start* = 0. If a survey data record is found at or beyond *start*, **mb_buffer_get_next_nav** returns the buffer index of that record in *id*. Navigation and vertical reference sensor data is also returned. No comments or other non-survey data records are returned. The *verbose* value controls the standard error output verbosity of the function.

The input control parameters have the following significance:

<i>start:</i>	The buffer index at which to start searching for a survey data record.
---------------	---

The returned values are:

<i>id:</i>	The buffer index of the first survey data record at or after <i>start</i> .
<i>time_i:</i>	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d:</i>	time of current ping in seconds

	since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>roll</i> :	ship roll in degrees
<i>pitch</i> :	ship pitch in degrees
<i>heave</i> :	ship heave in meters

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures. The most common error occurs when no more survey data records remain to be found in the buffer; in this case, *error* = -14.

```
-----
int mb_buffer_extract_nav(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int id,
    int *kind,
    int time_i[7],
    double *time_d,
    double *navlon,
    double *navlat,
    double *speed,
    double *heading,
    double *draft,
    double *roll,
    double *pitch,
    double *heave,
    int *error);
```

The function **mb_buffer_extract_nav** extracts and returns a subset of the data in a buffer record. The *verbose* value controls the standard error output verbosity of the function. The buffer record is specified with the buffer index *id*. The data returned consists of navigation and vertical reference sensor data.

The input control parameters have the following significance:

<i>id</i> :	The buffer index of the data record to extract.
-------------	---

The returned values are:

<i>kind</i> :	kind of data record extracted
	1 survey data
	2 comment
	>=3 other data that cannot
	be passed by mb_buffer_extract_nav
<i>time_i</i> :	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond

<i>time_d</i> :	time of current ping in seconds since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>roll</i> :	ship roll in degrees
<i>pitch</i> :	ship pitch in degrees
<i>heave</i> :	ship heave in meters

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about extract failures.

```

-----
int mb_buffer_insert_nav(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int id,
    int time_i[7],
    double time_d,
    double navlon,
    double navlat,
    double speed,
    double heading,
    double draft,
    double roll,
    double pitch,
    double heave,
    int *error);

```

The function **mb_buffer_insert_nav** inserts navigation and vertical reference sensor data into a buffer record, replacing a subset of the original values. The *verbose* value controls the standard error output verbosity of the function. The buffer record is specified with the buffer index *id*.

The input control parameters have the following significance:

<i>id</i> :	The buffer index of the data record to insert.
-------------	---

The returned values are:

<i>kind</i> :	kind of data record inserted
	1 survey data
	2 comment
	>=3 other data that cannot be passed by mb_buffer_insert_nav
<i>time_i</i> :	time of current ping
	<i>time_i</i> [0]: year
	<i>time_i</i> [1]: month
	<i>time_i</i> [2]: day
	<i>time_i</i> [3]: hour
	<i>time_i</i> [4]: minute
	<i>time_i</i> [5]: second
	<i>time_i</i> [6]: microsecond
<i>time_d</i> :	time of current ping in seconds

	since 1/1/70 00:00:00
<i>navlon</i> :	longitude
<i>navlat</i> :	latitude
<i>speed</i> :	ship speed in km/s
<i>heading</i> :	ship heading in degrees
<i>roll</i> :	ship roll in degrees
<i>pitch</i> :	ship pitch in degrees
<i>heave</i> :	ship heave in meters

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about insert failures.

```
-----
int mb_buffer_get_ptr(
    int verbose,
    char *buff_ptr,
    char *mbio_ptr,
    int id,
    char **store_ptr,
    int *error);
```

The function **mb_buffer_get_ptr** returns a pointer to the data structure in a buffer record. The *verbose* value controls the standard error output verbosity of the function. The buffer record is specified with the buffer index *id*. The data returned consists of a pointer to the data structure stored in the specified buffer record.

The input control parameters have the following significance:

<i>id</i> :	The buffer index of the data record to locate.
-------------	---

The return values are:

<i>*store_ptr</i> :	pointer to data in specified buffer record
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about buffer failures.

MISCELLANEOUS FUNCTIONS

```
int mb_defaults(
    int verbose,
    int *format,
    int *pings,
    int *lonflip,
    double bounds[4],
    int *btime_i,
    int *etime_i,
    double *speedmin,
    double *timegap);
```

The function **mb_defaults** provides default values of control parameters used by some of the **MBIO** functions. The *verbose* value controls the standard error output verbosity of the function. The other parameters are set by the function; the meaning of these parameters is discussed in the listings of the functions **mb_read_init** and **mb_write_init**. If an *.mbio_defaults* file exists in the user's home directory, the *lonflip*

and timegap defaults are read from this file. Otherwise, the values are set as:

**longflip* = 0

**timegap* = 1

The other values are simply set as:

**format* = 0

**pings* = 1

bounds[0] = -360.

bounds[1] = 360.

bounds[2] = -90.

bounds[3] = 90.

btime_i[0] = 1962;

btime_i[1] = 2;

btime_i[2] = 21;

btime_i[3] = 10;

btime_i[4] = 30;

btime_i[5] = 0;

btime_i[6] = 0;

etime_i[0] = 2062;

etime_i[1] = 2;

etime_i[2] = 21;

etime_i[3] = 10;

etime_i[4] = 30;

etime_i[5] = 0;

etime_i[6] = 0;

**speedmin* = 0.0

A status value is returned to indicate success or failure.

```
-----
int mb_env(
    int verbose,
    char *psdisplay,
    char *imgdisplay,
    char *mbproject);
```

The function **mb_env** provides default values of Postscript and image display programs invoked by some **MB-System** programs and macros, and a default value for a working project name that will be used by future applications. The *verbose* value controls the standard error output verbosity of the function. If an *.mbio_defaults* file exists in the user's home directory, the **psdisplay*, **imgdisplay*, **mbproject* defaults are read from this file. Otherwise, the values are set as:

```
psdisplay = "xpsview" (IRIX OS)
             "pageview" (Solaris OS)
             "gv" (other OS)
             "ghostview" (other OS)
imgdisplay = "gimp" (Linux OS)
             "xv" (other than Linux OS)
mbproject = "none"
```

```
-----
int mb_format(
    int verbose,
    int *format,
    int *error);
```

Given the format identifier *format*, **mb_format** checks if the format is valid. If the format id corresponds to a value used in previous (<4.00) versions of **MB-System**, then the format value will be aliased to the current corresponding value.

The return values are:

<i>format</i> :	MBIO format id
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_format_register(
    int verbose,
    int *format,
    char *mbio_ptr,
    int *error);
```

The function **mb_format_register** is called by **mb_read_init** and **mb_write_init** and serves to load format specific parameters and function parameters into the **MBIO** control structure pointed to by **error*. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values.

The input values are:

<i>*format</i> :	MBIO format id
<i>*mbio_ptr</i> :	pointer to data in specified buffer record

The return values are:

<i>error</i> :	error value
----------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_format_info(
    int verbose,
    int *format,
    int *system,
    int *beams_bath_max,
    int *beams_amp_max,
    int *pixels_ss_max,
    char *format_name,
    char *system_name,
    char *format_description,
    int *numfile,
    int *filetype,
    int *variable_beams,
    int *traveltime,
    int *beam_flagging,
    int *nav_source,
    int *heading_source,
    int *vru_source,
    double *beamwidth_xtrack,
```

```
double *beamwidth_ltrack,
int *error);
```

The function **mb_format_info** returns a variety of data format specific parameters. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values.

The input values are:

format*: **MBIO format id

The return values are:

format*: **MBIO format id
system*: **MBIO sonar system id
**beams_bath_max*: maximum number of bathymetry beams
**beams_amp_max*: maximum number of amplitude beams
**pixels_ss_max*: maximum number of sidescan pixels
format_name*: **MBIO format name
system_name*: **MBIO sonar system name
format_description*: **MBIO format description
**numfile*: number of parallel data files used in format
**filetype*: type of data files
**variable_beams*: number of beams can vary [boolean]
**traveltime*: travel time data available [boolean]
**beam_flagging*: beam flagging supported [boolean]
**nav_source*: kind of data records containing navigation
**heading_source*: kind of data records containing heading
**vru_source*: kind of data records containing attitude
**beamwidth_xtrack*: typical athwartships beam width [degrees]
**beamwidth_ltrack*: typical alongtrack beam width [degrees]
error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_format_system(
    int verbose,
    int *format,
    int *system,
    int *error);
```

The function **mb_format_system** returns the **MBIO** sonar system id. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values. The input values are:

format*: **MBIO format id

The return values are:

format*: **MBIO format id
system*: **MBIO sonar system id
error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed

information about failures.

```
-----
int mb_format_description(
    int verbose,
    int *format,
    char *description,
    int *error);
```

The function **mb_format_description** returns a short description of the format in the string **description*. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values. The input values are:

format*: **MBIO format id

The return values are:

format*: **MBIO format id

format_description*: **MBIO format description

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_format_dimensions(
    int verbose,
    int *format,
    int *beams_bath_max,
    int *beams_amp_max,
    int *pixels_ss_max,
    int *error);
```

The function **mb_format_dimensions** returns the maximum numbers of beams and pixels associated with a particular data format. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values. The input values are:

format*: **MBIO format id

The return values are:

format*: **MBIO format id

**beams_bath_max*: maximum number of bathymetry beams

**beams_amp_max*: maximum number of amplitude beams

**pixels_ss_max*: maximum number of sidescan pixels

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_format_flags(
    int verbose,
    int *format,
    int *variable_beams,
    int *traveltime,
    int *beam_flagging,
    int *error);
```

The function **mb_format_flags** returns flags indicating certain characteristics of the specified data format. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values. The input values are:

<i>*format:</i>	MBIO format id
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16
17	17
18	18
19	19
20	20
21	21
22	22
23	23
24	24
25	25
26	26
27	27
28	28
29	29
30	30
31	31
32	32
33	33
34	34
35	35
36	36
37	37
38	38
39	39
40	40
41	41
42	42
43	43
44	44
45	45
46	46
47	47
48	48
49	49
50	50
51	51
52	52
53	53
54	54
55	55
56	56
57	57
58	58
59	59
60	60
61	61
62	62
63	63
64	64
65	65
66	66
67	67
68	68
69	69
70	70
71	71
72	72
73	73
74	74
75	75
76	76
77	77
78	78
79	79
80	80
81	81
82	82
83	83
84	84
85	85
86	86
87	87
88	88
89	89
90	90
91	91
92	92
93	93
94	94
95	95
96	96
97	97
98	98
99	99

The return values are:

<i>*format:</i>	MBIO format id
-----------------	-----------------------

**variable_beams*: number of beams can vary [boolean]

**traveltime:* travel time data available [boolean]

**beam_flagging*: beam flagging supported [boolean]

<i>error:</i>	error value
---------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
int mb_format_source(  
    int verbose,  
    int *format,  
    int *nav_source,  
    int *heading_source,  
    int *vru_source,  
    int *error);
```

The function **mb_format_source** returns flags indicating what kinds of data records contain navigation, heading, and attitude values in the specified data format. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values. The input values are:

<i>*format:</i>	MBIO format id
-----------------	----------------

The return values are:

<i>*format:</i>	MBIO format id
-----------------	-----------------------

**nav_source*: kind of data records containing navigation

**heading_source*: kind of data records containing heading

**vru_source*: kind of data records containing attitude

<i>error:</i>	error value
---------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
int mb_format_beamwidth(  
    int verbose,  
    int *format,  
    double *beamwidth_xtrack,  
    double *beamwidth_ltrack,  
    int *error);
```

The function **mb_format_beamwidth** returns typical, upper bound values for athwartships and alongtrack beam widths. The format id **format* is first checked for validity. In some cases, formerly valid but now obsolete format id values are mapped to current values. The input values are:

<i>*format:</i>	MBIO format id
-----------------	-----------------------

The return values are:

<i>*format:</i>	MBIO format id
-----------------	-----------------------

* <i>beamwidth_xtrack</i> :	typical athwartships beam width [degrees]
-----------------------------	--

<i>*beamwidth_ltrack:</i>	typical alongtrack beam width [degrees]
---------------------------	--

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
int mb_datalist_open(
    int verbose,
    char **datalist,
    char *path,
    int look_processed,
    int *error);
```

The function **mb_datalist_open** initializes reading from a datalist tree. The string *path* is the path to the top level datalist file to be opened. The value *look_processed* indicates whether the datalist parsing should look for or ignore processed data files (see the **mbprocess** and **mbdatalist** manual pages). The input values are:

path: datalist file to be opened
look_processed: processed file behavior
 0 : unset
 1 : ignore processed files
 2 : return processed files

The return values are:

***datalist:* pointer to datalist
 structure
error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
int mb_datalist_read(
    int verbose,
    char *datalist,
    char *path,
    int *format,
    double *weight,
    int *error);
```

The function **mb_datalist_read** reads from a datalist tree, attempting to return the path to the next valid swath data file, the corresponding data format id, and a gridding weight (see the **mbprocess** and **mbdatalist** manual pages). Information about the datalist tree is embedded in a data structure pointed to by **datalist*. The input values are:

**datalist:* pointer to datalist
 structure

The return values are:

**path:* swath data file
format:* **MBIO format id
weight:* **mbgrid gridding weight
error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
int mb_datalist_close(
    int verbose,
```



```
char **datalist,
int *error);
```

The function **mb_datalist_close** closes an open datalist tree, and deallocates the data structure pointed to by **datalist*. The input values are:

**datalist*: pointer to datalist
structure

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_alloc(
    int verbose,
    char *mbio_ptr,
    char **store_ptr,
    int *error);
```

The function **mb_alloc** allocates a data structure for internal storage of swath sonar data and returns a pointer to this structure in **store_ptr*. The data structure is specific to the data format identified in the **MBIO** data structure pointed to by **mbio_ptr*. The input values are:

mbio_ptr*: pointer to **MBIO structure

The return values are:

***store_ptr*: pointer to storage data
structure

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_deall(
    int verbose,
    char *mbio_ptr,
    char **store_ptr,
    int *error);
```

The function **mb_deall** deallocates a format specific swath sonar data structure pointed to by **store_ptr*. The input values are:

mbio_ptr*: pointer to **MBIO structure

**store_ptr*: pointer to storage data
structure

The return values are:

error: error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_error(
    int error,
    int error,
    char **message);
```

Given the error value *error*, **mb_format_inf** returns a short error message in the string ***message*. The

verbose value controls the standard error output verbosity of the function. The return status value signals success if *format* is valid and failure otherwise.

```
-----
int mb_navint_add(
    int verbose,
    char *mbio_ptr,
    double time_d,
    double lon,
    double lat,
    int *error);
```

The function **mb_navint_add** adds a navigation fix to a circular buffer of navigation values maintained in the **MBIO** data structure pointed to by **mbio_ptr*. This buffer is used to interpolate navigation for data formats where the navigation is asynchronous (where navigation and survey pings come in different data records). The input values are:

<i>*mbio_ptr</i> :	pointer to MBIO structure
<i>time_d</i> :	time of navigation fix in seconds since 1/1/70 00:00:00
<i>lon</i> :	longitude (degrees)
<i>lat</i> :	latitude (degrees)

The return values are:

<i>error</i> :	error value
----------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_navint_interp(
    int verbose,
    char *mbio_ptr,
    double time_d,
    double heading,
    double rawspeed,
    double *lon,
    double *lat,
    double *speed,
    int *error);
```

The function **mb_navint_interp** interpolates navigation to the time *time_d* using a circular buffer of navigation values maintained in the **MBIO** data structure pointed to by **mbio_ptr*. This buffer is used to interpolate navigation for data formats where the navigation is asynchronous (where navigation and survey pings come in different data records). The input values are:

<i>*mbio_ptr</i> :	pointer to MBIO structure
<i>time_d</i> :	time of current ping in seconds since 1/1/70 00:00:00
<i>heading</i> :	heading in degrees
<i>rawspeed</i> :	speed in km/hr (zero if not known)

The return values are:

<i>*lon</i> :	longitude (degrees)
<i>*lat</i> :	latitude (degrees)
<i>*speed</i> :	speed made good in km/hr
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed

information about failures.

```
-----
int mb_attint_add(
    int verbose,
    char *mbio_ptr,
    double time_d,
    double heave,
    double roll,
    double pitch,
    int *error);
```

The function **mb_attint_add** adds an attitude (heave, roll, pitch) data point to a circular buffer of attitude values maintained in the **MBIO** data structure pointed to by ***mbio_ptr**. This buffer is used to interpolate attitude for data formats where the attitude is asynchronous (where attitude and survey pings come in different data records). The input values are:

<i>*mbio_ptr</i> :	pointer to MBIO structure
<i>time_d</i> :	time of attitude in seconds since 1/1/70 00:00:00
<i>heave</i> :	heave (meters, up +)
<i>roll</i> :	roll (degrees, starboard up +)
<i>pitch</i> :	pitch (degrees, forward up +)

The return values are:

<i>error</i> :	error value
----------------	-------------

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_attint_interp(
    int verbose,
    char *mbio_ptr,
    double time_d,
    double *heave,
    double *roll,
    double *pitch,
    int *error);
```

The function **mb_attint_interp** interpolates attitude (heave, roll, pitch) data to the time *time_d* using a circular buffer of attitude values maintained in the **MBIO** data structure pointed to by ***mbio_ptr**. This buffer is used to interpolate attitude for data formats where the attitude is asynchronous (where attitude and survey pings come in different data records). The input values are:

<i>*mbio_ptr</i> :	pointer to MBIO structure
<i>time_d</i> :	time of current ping in seconds since 1/1/70 00:00:00

The return values are:

<i>*heave</i> :	heave (meters, up +)
<i>*roll</i> :	roll (degrees, starboard up +)
<i>*pitch</i> :	pitch (degrees, forward up +)
<i>error</i> :	error value

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```
-----
int mb_hedint_add(
```

```

    int verbose,
    char *mbio_ptr,
    double time_d,
    double heading,
    int *error);

```

The function **mb_hedint_add** adds a heading point to a circular buffer of heading values maintained in the **MBIO** data structure pointed to by ***mbio_ptr**. This buffer is used to interpolate heading for data formats where the heading is asynchronous (where heading and survey pings come in different data records). The input values are:

```

    *mbio_ptr:    pointer to MBIO structure
    time_d:       time of heading value in seconds
                  since 1/1/70 00:00:00
    heading:      heading (degrees)

```

The return values are:

```

    error:        error value

```

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```

-----
int mb_hedint_interp(
    int verbose,
    char *mbio_ptr,
    double time_d,
    double *heading,
    int *error);

```

The function **mb_hedint_interp** interpolates heading to the time *time_d* using a circular buffer of heading values maintained in the **MBIO** data structure pointed to by ***mbio_ptr**. This buffer is used to interpolate heading for data formats where the heading is asynchronous (where heading and survey pings come in different data records). The input values are:

```

    *mbio_ptr:    pointer to MBIO structure
    time_d:       time of current ping in seconds
                  since 1/1/70 00:00:00

```

The return values are:

```

    *heading:     heading in degrees
    error:        error value

```

A status value indicating success or failure is returned; the error value argument *error* passes more detailed information about failures.

```

-----
int mb_get_double(
    double *value,
    char *str,
    int nchar);

```

The function **mb_get_double** parses the first *nchar* characters of the string **str* for a floating point value, storing this value as a double in **value*.

```

-----
int mb_get_int(
    int *value,
    char *str,
    int nchar);

```

The function **mb_get_int** parses the first *nchar* characters of the string **str* for an integer value, storing this value as a int in **value*.

```
-----  
int mb_get_binary_short(  
    int swapped,  
    void *buffer,  
    short *value);
```

The function **mb_get_binary_short** extracts a short int value from the first two bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of **value* is swapped.

```
-----  
int mb_get_binary_int(  
    int swapped,  
    void *buffer,  
    int *value);
```

The function **mb_get_binary_int** extracts an int value from the first four bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of **value* is swapped.

```
-----  
int mb_get_binary_float(  
    int swapped,  
    void *buffer,  
    float *value);
```

The function **mb_get_binary_float** extracts a float value from the first four bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of **value* is swapped.

```
-----  
int mb_get_binary_double(  
    int swapped,  
    void *buffer,  
    double *value);
```

The function **mb_get_binary_double** extracts a double value from the first eight bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of **value* is swapped.

```
-----  
int mb_put_binary_short(  
    int swapped,  
    short value,  
    void *buffer);
```

The function **mb_put_binary_short** inserts a short int value into the first two bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of *value* is swapped.

```
-----  
int mb_put_binary_int(  
    int swapped,  
    int value,  
    void *buffer);
```

The function **mb_put_binary_int** inserts an *int* value into the first four bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of value is swapped.

```
-----
int mb_put_binary_float(
    int swapped,
    float value,
    void *buffer);
```

The function **mb_put_binary_float** inserts a *float* value into the first four bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of value is swapped.

```
-----
int mb_put_binary_double(
    int swapped,
    double value,
    void *buffer);
```

The function **mb_put_binary_double** inserts a *double* value into the first eight bytes pointed to by **buffer*. If the boolean *swapped* is true, the byte order of value is swapped.

```
-----
int mb_get_bounds(
    char *text,
    double *bounds);
```

The function **mb_get_bounds** parses the string **text* and extracts geographic bounds of a rectangular region in the form:

```
bounds[0]: minimum longitude
bounds[1]: maximum longitude
bounds[2]: minimum latitude
bounds[3]: maximum latitude
```

where **text* is in the standard **GMT** bounds form. The longitude and latitude values in **text* should separated by a '/' character, and individual values may be represented in decimal degrees or in "dd:mm:ss" form (dd=degrees, mm=minutes, ss=seconds).

```
double mb_ddmmss_to_degree(
    char *text);
```

The function **mb_ddmmss_to_degree** parses the string **text* and extracts a decimal longitude or latitude value from a "dd:mm:ss" (dd=degrees, mm=minutes, ss=seconds) value.

```
-----
int mb_takeoff_to_rollpitch(
    int verbose,
    double theta,
    double phi,
    double *alpha,
    double *beta,
    int *error);
```

The function **mb_takeoff_to_rollpitch** translates angles from the "takeoff" coordinate reference frame to the "rollpitch" coordinate system. See the discussion of coordinate systems below.

```
-----
int mb_rollpitch_to_takeoff(
    int verbose,
    double alpha,
    double beta,
    double *theta,
    double *phi,
    int *error);
```

The function **mb_rollpitch_to_takeoff** translates angles from the "rollpitch" coordinate reference frame to the "takeoff" coordinate system. See the discussion of coordinate systems below.

```
-----
int mb_double_compare(double *a,
    double *b);
```

The function **mb_double_compare** is used with the **qsort** function. This function returns 1 if $a > b$ and -1 if $a \leq b$.

```
-----
int mb_int_compare(
    int *a,
    int *b);
```

The function **mb_int_compare** is used with the **qsort** function. This function returns 1 if $a > b$ and -1 if $a \leq b$.

COORDINATE SYSTEMS USED IN MB-SYSTEM

I. Introduction

The coordinate systems described below are used within **MB-System** for calculations involving the location in space of depth, amplitude, or sidescan data. In all cases the origin of the coordinate system is at the center of the sonar transducers.

II. Cartesian Coordinates

The cartesian coordinate system used in **MB-System** is a bit odd because it is left-handed, as opposed to the right-handed x-y-z space conventionally used in most circumstances. With respect to the sonar (or the ship on which the sonar is mounted), the x-axis is athwartships with positive to starboard (to the right if facing forward), the y-axis is fore-aft with positive forward, and the z-axis is positive down.

III. Spherical Coordinates

There are two non-traditional spherical coordinate systems used in **MB-System**. The first, referred to here as takeoff angle coordinates, is useful for raytracing. The second, referred to here as roll-pitch coordinates, is useful for taking account of corrections to roll and pitch angles.

III.1. Takeoff Angle Coordinates

The three parameters are r, theta, and phi, where r is the distance from the origin, theta is the angle from vertical down (that is, from the positive z-axis), and phi is the angle from across-track (the positive x-axis) in the x-y plane. Note that theta is always positive; the direction in the x-y plane is given by phi. Raytracing is simple in these coordinates because the ray takeoff angle is just theta. However, applying roll or pitch corrections is complicated because roll and pitch have components in both theta and phi.

$$0 \leq \theta \leq \pi/2$$

$$-\pi/2 \leq \phi \leq 3\pi/2$$

```

x = rSIN(theta)COS(phi)
y = rSIN(theta)SIN(phi)
z = rCOS(theta)

```

```

theta = 0    ---> vertical, along positive z-axis
theta = PI/2 ---> horizontal, in x-y plane
phi = -PI/2  ---> aft, in y-z plane with y negative
phi = 0      ---> port, in x-z plane with x positive
phi = PI/2   ---> forward, in y-z plane with y positive
phi = PI     ---> starboard, in x-z plane with x negative
phi = 3*PI/2 ---> aft, in y-z plane with y negative

```

III.2. Roll-Pitch Coordinates

The three parameters are r, alpha, and beta, where r is the distance from the origin, alpha is the angle forward (effectively pitch angle), and beta is the angle from horizontal in the x-z plane (effectively roll angle). Applying a roll or pitch correction is simple in these coordinates because pitch is just alpha and roll is just beta. However, raytracing is complicated because deflection from vertical has components in both alpha and beta.

```

-PI/2 <= alpha <= PI/2
0 <= beta <= PI

```

```

x = rCOS(alpha)COS(beta)
y = rSIN(alpha)
z = rCOS(alpha)SIN(beta)

```

```

alpha = -PI/2 ---> horizontal, in x-y plane with y negative
alpha = 0      ---> ship level, zero pitch, in x-z plane
alpha = PI/2   ---> horizontal, in x-y plane with y positive
beta = 0       ---> starboard, along positive x-axis
beta = PI/2    ---> in y-z plane rotated by alpha
beta = PI      ---> port, along negative x-axis

```

IV. SeaBeam Coordinates

The per-beam parameters in the SB2100 data format include angle-from-vertical and angle-forward. Angle-from-vertical is the same as theta except that it is signed based on the across-track direction (positive to starboard, negative to port). The angle-forward values are also defined slightly differently from phi, in that angle-forward is signed differently on the port and starboard sides. The SeaBeam 2100 External Interface Specifications document includes both discussion and figures illustrating the angle-forward value. To summarize:

Port:

```

theta = absolute value of angle-from-vertical

```

```

-PI/2 <= phi <= PI/2
is equivalent to
-PI/2 <= angle-forward <= PI/2

```

```

phi = -PI/2 ---> angle-forward = -PI/2 (aft)
phi = 0      ---> angle-forward = 0   (starboard)
phi = PI/2   ---> angle-forward = PI/2 (forward)

```


Starboard:

theta = angle-from-vertical

$\text{PI}/2 \leq \text{phi} \leq 3 \cdot \text{PI}/2$

is equivalent to

$-\text{PI}/2 \leq \text{angle-forward} \leq \text{PI}/2$

$\text{phi} = \text{PI}/2 \rightarrow \text{angle-forward} = -\text{PI}/2$ (forward)

$\text{phi} = \text{PI} \rightarrow \text{angle-forward} = 0$ (port)

$\text{phi} = 3 \cdot \text{PI}/2 \rightarrow \text{angle-forward} = \text{PI}/2$ (aft)

V. Usage of Coordinate Systems in **MB-System**

Some sonar data formats provide angle values along with travel times. The angles are converted to takeoff-angle coordinates regardless of the storage form of the particular data format. Currently, most data formats do not contain an alongtrack component to the position values; in these cases the conversion is trivial since $\text{phi} = \text{beta} = 0$ and $\text{theta} = \text{alpha}$. The angle and travel time values can be accessed using the **MBIO** function **mb_ttimes**. All angle values passed by **MB-System** functions are in degrees rather than radians.

The programs **mbbath** and **mbvelocitytool** use angles in take-off angle coordinates to do the raytracing. If roll and/or pitch corrections are to be made, the angles are converted to roll-pitch coordinates, corrected, and then converted back prior to raytracing.

BEAM FLAGS USED IN MB-SYSTEM

MB-System uses arrays of 1-byte "beamflag" values to indicate beam data quality. Each beamflag value is actually an eight bit mask allowing fairly complicated information to be stored regarding each bathymetry value. In particular, beams may be flagged as bad, they may be selected as being of special interest, and one or more reasons for flagging or selection may be indicated. This scheme is very similar to the convention used in the HMPS hydrographic data processing package and the SAIC Hydrobat package. The beam selection mechanism is not currently used by any **MB-System** programs.

The flag and select bits:

xxxxxx00 => This beam is neither flagged nor selected.

xxxxxx01 => This beam is flagged as bad and should be ignored.

xxxxxx10 => This beam has been selected.

Flagging modes:

00000001 => Flagged because no detection was made by the sonar.

xxxxx101 => Flagged by manual editing.

xxxx1x01 => Flagged by automatic filter.

xxx1xx01 => Flagged because uncertainty exceeds 1 X IHO standard.

xx1xxx01 => Flagged because uncertainty exceeds 2 X IHO standard.

x1xxxx01 => Flagged because footprint is too large

1xxxxx01 => Flagged by sonar as unreliable.

Selection modes:

00000010 => Selected, no reason specified.

xxxxx110 => Selected as least depth.

xxxx1x10 => Selected as average depth.

xxx1xx10 => Selected as maximum depth.

xx1xxx10 => Selected as location of sidescan contact.

x1xxxx10 => Selected, spare.

1xxxxx10 => Selected, spare.

SEE ALSO

mbsystem(1), **mbformat**(1)

BUGS

What could go wrong in a mere 374,852 lines of C code?
Let us know...