libsegy Programmer's Reference Manual

Nate Gauntt

Last Modified: August 11, 2008

Contents

1 Introduction

libsegy was originally created to be a simple, C language API for reading and writing SEG-Y (Rev. 0) format files. This file format is in common use (at present) in geophysics industry and research communities, and is defined in [1] and [2]. Currently libsegy reads and writes Revision 0 type files, with data stored on disk as the native IBM floating point format.

2 Why Use libsegy?

There are several freely available projects in the seismic community that deal with reading, writing, formatting and analyzing seismic trace data, many of which have SEG-Y support. There are many more commercial products available to do this task, often with sophisticated analysis and visualization toolkits built in. The motivation for creating libsegy stems from the following requirements.

The project I was working on needed a SEGY library that:

- was freely available
- had a well documented programming interface
- was low level (no visualization or analysis needed)
- can be made to interoperate with MATLAB in a short time frame

That said, here is a short list of some popular SEG-Y projects and why I didn't choose to use them for my initial project.

GSEGYView	− primarily a visualization toolkit − high level interfaces − doesn't support SEG-Y Revision 0
Seismic UNIX (SU)	+ well known and well documented − separate programs, shell script integration − difficult to integrate with MATLAB
Seismic-Toolkit	− primarily a visualization toolkit − only reads SAC format input files
Kogeo	+ full support for SEG-Y and SU file formats − primarily a visualization toolkit − doesn't build or run natively on Linux
SegyPy	+ full support for SEG-Y Revision 0 and 1 − written in python, difficult to integrate with MATLAB
SegyMAT	+ full support for SEG-Y Revision 0 and 1 + well integrated with MATLAB (front-end for SegyPy) − reading SEG-Y too slow for interactive graphics − not well documented, difficult to modify

3 Building and Installation

3.1 Building C-Library Interface

From the base directory, usually 'libsegy', type 'make' at the command line. This should build two files in the 'libsegy/obj' directory called 'libsegy.a', the static library, and 'libsegy.so', the dynamic shared object library. These two libraries are exactly the same, other than one is shared and one is static.

3.2 Building MATLAB MEX Interface

From the base directory, type 'cd mex' and then type 'make'. You may have to customize the file 'libsegy/mex/Makefile' to point to the root of your MATLAB installation. Then add the directory 'libsegy/obj' to your MATLAB path and you should have access to the built MEX functions. The shared libraries implementing the mex functions have the extension '.mexglx'. See [3] and [4] for more details on building and using the MEX MATLAB interface.

4 C Language Functions

This section describes the (native) C Language Application Programming Interface (API). In general, the API was designed to closely match the Unix system call API for files. The sequence for accessing SEG-Y files is: open, read/write/seek, close. SEG-Y files are represented, as in UNIX, as an opaque integer file handle, and this is used internally in the library as identifiers for file resources.

4.1 File Management: segy_open_file, segy_close_file

```
segy_open_file(filename, mode)
```

`segy_open_file` is for opening existing and creating new SEGY files. `filename` is a C string of the file you want to create or open, and `mode` is one of the library constants SEGY_RDONLY or SEGY_RDWR, for read-only and read+write modes, respectively. A successful call will return the integer file handle for an open UNIX file, or -1 on error.

```
segy_close_file(fd)
```

`segy_close_file` closes an open segy file. A successful call will return 0.

Example code:

```
#include <libsegy.h>

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

  // open foo.segy for read-only access
  int file_handle = segy_open_file("foo.segy", SEGY_RDONLY);

  // check for errors
  if (file_handle < 0) {
    // error code
  }

  // or use macro name
  if (file_handle == SEGY_OPEN_ERR) {
    // error code
  }

  // ... use file_handle ...
```

```
19
20    // close file
21    int close_error = segy_close_file(file_handle);
22
23    return 0;
24  }
```

4.2 File Positioning: segy_seek_trace

```
segy_seek_trace(fd, tracenum)
```

`segy_seek_trace` is for moving around in a SEGY file to the start of a given trace block. `fd` is an integer file descriptor of an open UNIX file, and `tracenum` is the trace (starting from 1) number of the trace to seek to. A successful call will position the library such that the next or write will act on the given trace. On error, an error status is returned and the library attempts to seek to the previous valid position if possible.

Example code:

```
1   // ... assume we have an open segy file ...
2   int format = 1;                 // files stored as IBM fp
3   segy_trace_header header_buf;   // stores trace headers
4   float data_buf[1024] = {0};     // example 1024 samples/trace
5   int data_len = 1024;            // size of data_buf
6
7   // seek to first trace
8   int errcode = segy_seek_trace(file_handle, 1);
9
10  // read first trace with headers
11  errcode = segy_read_trace(file_handle, format, header_buf,
12                            data_buf, data_len);
13
14  // seek to third trace
15  errcode = segy_seek_trace(file_handle, 2);
16
17  // read third trace with headers
18  errcode = segy_read_trace(file_handle, format, header_buf,
19                            data_buf, data_len);
```

4.3 Reading a File: segy_read*

```
segy_read_file_header(fd, header)
```

`segy_read_file_header` is for reading the file header information only, information such as the samples per trace or data format code number. `fd` is the integer file handle of an open UNIX file, and `header` is a pointer to a `segy_file_header` structure. After a successful call, `header` is filled in with SEGY file header data and the file position is at the start of the first trace header.

```
segy_read_trace_header(fd, format, header)
```

`segy_read_trace_header` is for reading trace header information while skipping over trace data. `fd` is the integer file handle of an open UNIX file, `format` is the format number of the binary format for trace data points, and `header` is a pointer to a `segy_trace_header` structure. Note that at this time, only `format` = 1 is supported, the IBM binary floating point format code. After a successful call, `header` is filled in with trace header data and the file position is at the start of the next trace header.

```
segy_read_data(fd, format, samples, buf, buf_len)
```

`segy_read_data` is for reading in trace data while skipping past the header data. `fd` is the integer file handle of an open UNIX file, `format` is the binary format number, `samples` is the number of trace data samples in this trace, `buf` is a pointer to a C float array, and `buf_len` is the number of floats in `buf`. Note that you have to know `samples`, as this information is contained in the (skipped) trace header. After a successful call, the lesser of `samples` and `buf_len` trace datums are copied into `buf`, and the file position is at the start of the next trace header.

```
segy_read_trace(fd, format, header, buf, buf_len)
```

`segy_read_trace` is for reading both the trace headers and trace data. `fd` is the integer file handle of an open UNIX file, `format` is the binary format number, `header` is a pointer to a `segy_trace_header` structure, `buf` is a pointer to a C float array, and `buf_len` is the number of floats in `buf`. After a successful call to `segy_read_trace`, `header` is filled in with trace header data, `buf` is filled in with at most `buf_len` trace data samples, and the file position is at the start of the next trace header.

Example code:

```
// ... assume we have an open segy file ...
int format = 1;                 // files stored as IBM fp
segy_trace_header thead_buf;    // stores trace headers
segy_file_header  fhead_buf;    // stores file header

// read file header, create buffer for trace data
int errcode = segy_read_file_header(fd, fhead_buf);
int samples = fhead_buf.ReelSamplesPerTrace;
float *tdata = (float*)malloc(samples * sizeof(float));

// read first trace, headers only
errcode = segy_read_trace_header(fd, format, thead_buf);

// read second trace, data only
errcode = segy_read_data(fd, format, samples, tdata, samples);

// read third trace, headers and data
errcode = segy_read_trace(fd, format, thead_buf, tdata,
                          samples);

// ... other code ...
```

4.4 Writing a File: segy_write*

`segy_write_file_header(fd, header)`

`segy_write_file_header` overwrites the file header on disk with a new file header. `fd` is the integer file handle of an open UNIX file, `header` is a pointer to a `segy_file_header` struct. After a successful call, the on-disk SEGY binary file header should contain the same information as in `header`, and the file position is at the start of the first trace header.

`segy_write_trace(fd, format, header, buf, buf_len)`

`segy_write_trace` writes the given trace header and trace data to disk at the current file position. `fd` is the integer file handle of an open UNIX file, `format` is the binary format code, currently only $format = 1$ and $format = 5$ are supported, `header` is a pointer to a `segy_trace_header` structure, `buf` is a pointer to a C float array, and `buf_len` is the number of floats in `buf`. After a successful call, the given trace header and data are committed to disk and the file position is at the start of the next trace.

Example Code:

```
// ... assume we have an open segy file ...
int format = 1;                 // files stored as IBM fp
segy_trace_header thead;        // stores trace headers
segy_file_header  fhead;        // stores file header

// modify text header, 3200 chars, 40 'lines' of 80 chars each
char *txthd = fhead.TextHeader;
char *line1 = &txthd[0], *line2 = &txthd[80],
     *line40 = &txthd[3120];

// write lines to text header, for Rev. 0 the C## are important
strncpy(line1, "C01 First Line", 80);
strncpy(line2, "C02 Second Line", 80);
strncpy(line40, "C40 Last Line, End SEGY REV 0", 80);

// modify file header, these fields are "strongly recommended"
fhead.LineNumber = 0;
fhead.ReelNumber = 0;
fhead.TracesPerRecord = 2000;
fhead.AuxTracesPerRecord = 2000;
fhead.ReelSampleInterval = 3000;
fhead.ReelSamplesPerTrace = 1024;
fhead.SampleFormatCode = format;
fhead.CDPFold = 0;

// write file header to disk
int errcode = segy_write_file_header(fd, &fhead);

// modify trace header, these fields are "strongly recommended"
thead.LineSeqNumber = 0;
thead.FieldRecordNumber = 0;
thead.TraceNumber = 1;
thead.TraceCode = 1;
thead.SamplesPerTrace = 1024;     // also important to libsegy
thead.SampleInterval  = 3000;

// generate some trace data
float tdata[1024];
int i;
for (i = 0; i < 1024; i++) tdata[i] = (float)i / 1024.0;

// write first trace to disk
errcode = segy_write_trace(fd, format, &thead, &tdata, 1024);
```

```
44
45  // write second trace to disk
46  thead.TraceNumber = 2;
47  errcode = segy_write_trace(fd, format, &thead, &tdata, 1024);
48
49  // ... other code ...
```

4.5 Detecting Errors: segy_error_message

Most functions prefixed by `segy_*` return an integer error status when called. With the exception of `segy_open_file`, a non-zero result means that an error has occurred. Some errors are status errors, such as SEGY_EOF, SEGY_BAD_INPUT, and SEGY_BADF, for example. However, most errors are serious errors and it is recommended to halt or exit the program if they occur. An error return of 0 or SEGY_NO_ERROR indicates success.

```
char * segy_error_message(errnum)
```

`segy_error_message` returns a statically allocated error message string for the error number entered. `errnum` is the error number returned by a `segy_*` function (with the exception of `segy_open_file`).

Below is a list of `segy_*` library functions and the errors they return:

segy_open_file	SEGY_OPEN_ERR
segy_close_file	SEGY_CLOSE_ERR
segy_seek_trace	SEGY_BADF SEGY_SEEK_ERR
segy_read_file_header	SEGY_BADF SEGY_READ_ERR
segy_read_trace_header	SEGY_BADF SEGY_READ_ERR SEGY_EOF
segy_read_trace	SEGY_BADF SEGY_READ_ERR SEGY_EOF SEGY_BAD_INPUT
segy_read_data	SEGY_BADF SEGY_READ_ERR SEGY_EOF SEGY_BAD_INPUT
segy_write_file_header	SEGY_BADF SEGY_WRITE_ERR
segy_write_trace	SEGY_BADF SEGY_WRITE_ERR SEGY_BAD_INPUT

4.6 Utility Functions

`segy_ascii2ebcdic(buf, buf_len)`

`segy_ascii2ebcdic` converts ascii character buffer into EBCDIC (codepage 00037) formatted character data. `buf` is a pointer to a C character array, and `buf_len` is the length in characters of `buf`. Note that `segy_write_trace` does this automatically for ascii data in the `segy_file_header.TextHeader` field. After a successful call, `buf` is converted to EBCDIC formatted character data.

`segy_ebcdic2ascii(buf, buf_len)`

`segy_ebcdic2ascii` converts EBCDIC character buffer into ascii formatted character data. `buf` is a pointer to a C character array, and `buf_len` is the size in characters of `buf`. After a successful call, `buf` is converted to ascii formatted character data.

`convert2ieee(format, data, data_len, buf, buf_len)`

`convert2ieee` converts a byte array of SEGY trace data into an array of IEEE 32-bit floating point data. `format` is the data format code, currently only $format = 1$ is supported, `data` is a pointer to a C character array where trace byte data is stored, `data_len` is the length of `data` in characters (bytes), `buf` is a pointer to a C float array for output data, `buf_len` is the size in floats of `buf`. Note that `segy_read*` functions automatically call this for data read off hard disk. After a successful call, `buf` contains the lesser of {`buf_len`, `data_len`/4} converted floats.

`convert2segy(format, data, data_len, buf, buf_len)`

`convert2segy` converts a *byte* array of IEEE 32-bit floating point data to SEGY REV 0 format data. `format` is the data format code, currently only $format = 1$ and $format = 5$ are supported, `data` is a pointer to a C character array for input IEEE data, `data_len` is the length in characters (bytes) of `data`, `buf` is a pointer to a C float array of output SEGY data, and `buf_len` is the length in floats of `buf`. Note that `segy_write*` functions automatically call this for data written to hard disk. After a successful call, `buf` contains the lesser of {`buf_len`, `data_len`/4} converted floats.

5 MATLAB (R2007b) Language Bindings

The following section describes the MATLAB binding for the C libsegy library. The MATLAB interface is briefly described here, as it follows closely the C interface.

```
[fd] = segy_open_file(filename, openmode)
```

`fd` is an integer file handle of the open UNIX file, `filename` is the string name of the file, `openmode` is 0 for read-only access and 1 for read+write access.

```
[err] = segy_close_file(fd)
```

`err` is the error number returned, `fd` is an integer file handle of the open UNIX file.

```
[err] = segy_seek_trace(fd, tracenum)
```

`err` is the error number returned, `fd` is an integer file handle of the open UNIX file, `tracenum` is the number (starting at 1) of the trace to seek to.

```
[err, fhead] = segy_read_file_header(fd)
```

`err` is the error number returned, `fhead` is the MATLAB struct containing the file header fields, `fd` is an integer file handle of the open UNIX file.

```
[err, thead] = segy_read_trace_header(fd)
```

`err` is the error number returned, `thead` is the MATLAB struct containing the trace header fields, `fd` is an integer file handle of the open UNIX file.

```
[err, thead, tdata] = segy_read_trace(fd, tlen)
```

`err` is the error number returned, `thead` is the MATLAB struct containing the trace header fields, `tdata` is the MATLAB array containing the trace data, `fd` is an integer file handle of the open UNIX file, and `tlen` is the maximum number of data samples to read into `tdata`.

```
[err, tdata] = segy_read_data(fd, samples)
```

`err` is the error number returned, `tdata` is the MATLAB array containing the trace data, `fd` is an integer file handle of the open UNIX file, and `samples` is the number of trace data samples per trace.

```
[err] = segy_write_file_header(fd, fhead)
```

`err` is the error number returned, `fd` is an integer file handle of the open UNIX file, and `fhead` is the MATLAB struct containing file header fields to write, see `segy_new_file_header()`.

```
[err] = segy_write_trace(fd, format, thead, tdata)
```

`err` is the error number returned, `format` is the binary format number, `fd` is an integer file handle of the open UNIX file, `thead` is the MATLAB struct containing trace header fields to write, see `segy_new_trace_header()`, and tdata is the MATLAB *single* type float array containing trace data to write.

```
[fhead] = segy_new_file_header()
```

`fhead` is a newly initialized MATLAB struct representing the file header fields of a SEGY file.

```
[thead] = segy_new_trace_header()
```

`thead` is a newly initialized MATLAB struct representing the trace header fields of a SEGY file.

```
[msg] = segy_error_message(err)
```

`msg` is the string error message, and `err` is the error number returned by a segy function.

```
[ebcdic] = segy_ascii2ebcdic(ascii)
```

`ebcdic` is a MATLAB Char array containing EBCDIC (CP 00037) formatted text, `ascii` is a MATLAB Char array of ascii text.

```
[ascii] = segy_ebcdic2ascii(ebcdic)
```

`ascii` is a MATLAB Char array of ascii formatted text, and `ebcdic` is a MATLAB Char array of ebcdic formatted text.

6 Appendix A: C Library function headers

The following descriptions begin with information extracted from the library header file 'src/libsegy.h', followed by explanation of the function call and code examples of the function's use. The header comments follow the following template:

```
// return_type function_name(arg1, ..., argN)
//   arg1: arg1_type  = arg1 description
//   ...
//   argN: argN_type  = argN description
// general description of the function and any side effects or return values
EXPORT return_type function_name(arg1_type arg1, ..., argN_type argN);
```

return_type is the C type (if any) that the function returns to caller

function_name is the callable name of the function

arg1 is the name (if any) of the first function argument

arg1_type is the C type of the first function argument

arg1 description is a brief description of the first function argument

argN is the name (if any) of the N^{th} function argument

argN_type is the C type of the N^{th} function argument

argN description is a brief description of the N^{th} function argument

the line following the last C comment line is the valid C prototype declaration for the function. On most systems EXPORT is defined to be empty. If BUILD_DLL is preprocessor defined, EXPORT maps to '__declspec(dllexport)', and if LINK_DLL is defined, EXPORT maps to '__declspec(dllimport)'. This facillitates building shared libraries on WIN32 platforms.

```
// segy_open_file(filename, mode)
//   filename: char * = name of .segy file to open
//   mode    : int    = file access mode, either SEGY_RDONLY or SEGY_RDWR
// returns open file descriptor of .segy file, or -1 on error
EXPORT int segy_open_file(const char *filename, int mode);


// segy_close_file(fd)
//   fd: int = open file descriptor
// returns 0 on successfully closing fd, nonzero on error
EXPORT int segy_close_file(int fd);
```

```
// segy_seek_trace(fd, tracenum)
//   fd      : int = open file descriptor of .segy file
//   tracenum: int = trace number
// seeks to beginning of given trace
EXPORT int segy_seek_trace(int fd, int tracenum);


// segy_read_file_header(fd, header)
//   fd    : int = open file descriptor of .segy file
//   header: segy_file_header * = pointer to file header
// reads file header information from .segy file
EXPORT int segy_read_file_header(int fd, segy_file_header *header);


// segy_read_trace_header(fd, format, header)
//   fd    : int = open file descriptor for .segy file
//   format: int = sample format code
//   header: segy_trace_header * = pointer to trace header
// reads next trace header into buffers
EXPORT int segy_read_trace_header(int fd, int format,
                                  segy_trace_header *header);


// segy_read_data(fd, format, samples, buf, buf_len)
//   fd      : int = open file descriptor of .segy file
//   format  : int = sample format code
//   samples : int = number of samples per each trace in .segy file
//   buf     : float * = pointer to trace data buffer
//   buf_len : short   = size of trace data buffer (number of floats)
// reads next trace data into buffers
EXPORT int segy_read_data(int fd, int format, int samples,
                          float *buf, short buf_len);


// segy_read_trace(fd, format, header, buf, buf_len)
//   fd      : int = open file descriptor of .segy file
//   format  : int = sample format code
//   header  : segy_trace_header * = pointer to trace header
//   buf     : float * = pointer to trace data buffer
//   buf_len : short   = size of trace data buffer (number of floats)
// reads next trace header and data into buffers
EXPORT int segy_read_trace(int fd, int format, segy_trace_header *header,
                           float *buf, short buf_len);
```

```
// segy_write_file_header(fd, header)
//   fd   : int = open file descriptor of .segy file
//   header: segy_file_header * = pointer to file header
// writes file header information to .segy file
EXPORT int segy_write_file_header(int fd, const segy_file_header *header);


// segy_write_trace(fd, format, header, buf, buf_len)
//   fd      : int   = open file descriptor of .segy file
//   format  : int   = data format, see SampleFormatCode in file header
//   header  : segy_trace_header * = pointer to trace header
//   buf     : float * = pointer to trace data buffer
//   buf_len : short   = size of trace data buffer (number of floats)
// writes trace header and data from buffers to file
EXPORT int segy_write_trace(int fd, int format,
    const segy_trace_header *header,
    const float *buf, short buf_len);


// segy_ebcdic2ascii(buf, buf_len)
//   buf    : char * = pointer to ebcdic character data
//   buf_len: int    = length of buffer
// converts buf from ebcdic (codepage 00037) to ascii code
EXPORT void segy_ebcdic2ascii(char *buf, int buf_len);


// segy_ascii2ebcdic(buf, buf_len)
//   buf    : char * = pointer to ebcdic character data
//   buf_len: int    = length of buffer
// converts buf from ascii code to ebcdic (codepage 00037)
EXPORT void segy_ascii2ebcdic(char *buf, int buf_len);


// convert2ieee(format, data, data_len, buf, buf_len)
//   format  : short   = data format code, see SampleFormatCode in file header
//   data    : char *  = trace data bytes
//   data_len: int     = size of data in bytes
//   buf     : float * = buffer for output ieee data
//   buf_len : short   = size of buf (number of floats)
// converts segy rev0 data formats to ieee float format
EXPORT int convert2ieee(short format, const char *data, int data_len,
float *buf, short buf_len);


// convert2segy(format, data, data_len, buf, buf_len)
//   format  : short   = data format code, see SampleFormatCode in file header
```

```
//   data    : char *  = IEEE input float array, as bytes
//   data_len: int     = size of data in bytes
//   buf     : float * = buffer for output segy data
//   buf_len : short   = size of buf (number of floats)
// converts ieee float format to segy rev0 format
EXPORT int convert2segy(short format, const char *data, int data_len,
float *buf, short buf_len);
```

References

[1] K.M. Barry, D.A. Cavers, C.W. Kneale: *Recommended Standards for Digital Tape Formats*, Geophysics, v. 40, no. 02, 244-252 (1975)

[2] *./doc/seg_y_rev0.pdf*, Digital version of [1]

[3] Matlab Website, http://www.mathworks.com

[4] Matlab External Interface Guide