Rationalizing varying seismic data formats

Henry C. Bland

ABSTRACT

Inconsistent seismic data formats are a significant source of frustration to geophysical processors. Since field data are acquired in one format, stored onto field tapes in another format, and then processed in one or more formats, the probability of making mistakes is great. The SEG has attempted to standardize the data formats by publishing data format recommendations, but these recommendations are not always followed. The cause of data format problems is often due to poor understanding of data format issues and a lack of consistent nomenclature. Many developers of seismic processing software choose to ignore the SEG standards because adherence increases development time, and can affect the software's speed.

This paper proposes some nomenclature to aid in the discussion of seismic data formats. Some of the common problems in dealing are illustrated using the SEG-Y standard as an example. A conversion function library is proposed to solve the problem of binary format translation.

NOMENCLATURE

To begin the discussion on nomenclature we will define two terms:

Binary format: The way a vector of data samples are encoded into binary form for storage in a digital computer.

Data format: A generic term referring to the format of an entire data tape or data file.

The nomenclature that is presented will apply to binary formats only. To understand the different kinds of binary formats, it is helpful to consider the parameters which define the way data samples are stored. The term "data type" is commonly considered as one of the built-in storage types available in common computer languages such as C or Fortran:

Table 1. Data types available in common programming languages

C Data types	Fortran Data Types
short int	integer*2
int	integer *4
float	real
double	double precision

The C and FORTRAN data types are a useful start in defining how data samples are stored. It is necessary use terminology that is less generic since the number of bits allocated is implementation dependent. For example, the following C data types produce different sized results depending on the hardware and operating system:

Table 2. Bit sizes of C data types on various platforms

Data Type	Intel 80386 MS-DOS	Sun Sparc SUNOS	DEC Alpha OSF/1
short int	16	16	16
int	16	32	32
long	32	32	64

It is therefore useful to specify not only the type of data (real or integer) but also the size of the storage area in bits.

Generic data type: The number type (integer or floating point) as well as the total number of bits required to store a value.

Not all floating point data types with the same total number of bits are the same. Consider the IBM 370 and the IEEE 32-bit floating point encoding schemes. IBM 370 floating point numbers are implemented by breaking the number into two parts: a fractional part *F*, and an exponent part *E*. Any floating point number, *x*, is computed by:

$$x = F \times 16^{E-64} \tag{1}$$

The IBM 370 encoding scheme stores *F* using 24 bits and *E* using 7 bits. A sign bit *S* is stored in the top bit of the total 32 bit word:

Bit Number																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

The IEEE encoding scheme also splits the number into a fractional part *F* and an exponent part *E* but this scheme uses 2 as the exponent base.

$$x = F \times 2^{E-127} \tag{2}$$

IEEE encoding stores *F* using 23 bits and *E* using 4 bits:

Bit Number																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

The difference appears subtle, but the encoding scheme affects the range and accuracy of possible data values:

Table 3. Range and accuracy of two floating point encoding schemes

Floating point scheme	Range	Accuracy
IBM 370	$10^{-78} - 10^{76}$	1 part in 0.79 x 10^{29}
IEEE	$10^{-39} - 10^{38}$	1 part in 1.6 x 10^{29}

The point of this discussion is not to explain the intricacies of the different floating point formats, but rather, to show that different ranges and resolution can be obtained from the same number of bits depending on the binary encoding scheme used. In theory, there could be thousands of possible integer, fixed point, and floating point schemes, though in practice, only a small number of schemes are used.

Encoding scheme: Defines how a single number is converted into binary. Examples: IBM 370 floating point, IEEE floating point.

One factor which affects data exchange between different systems is byte order. Consider the following 32 bit binary number in IEEE Floating point format:

Bit Number																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Almost all data processing systems shuffle data in groupings of 8, 16, 32 or 64 bits. Systems which use 8 bit groupings might store the above 32 bit word like this:

Memory Address	Bit Number 76543210
01	SEEEEEEE
02	EFFFFFFF
03	FFFFFFFF
04	FFFFFFFF

This system stores the most significant bits of the number first, and the least significant bits last. The term used to describe a system such as this is big-endian, because the *big end* is stored first. Other systems might store the same data like this:

Memory Address	Bit Number 76543210
01	FFFFFFFF
02	EFFFFFFF
03	FFFFFFFF
04	SEEEEEEE

This system stores the least significant bits first, and is called little-endian.

Byte Order: When a large number of bits are segmented into smaller byte-sized pieces, the byte order refers to the order in which the pieces are stored in byte-addressable storage (memory or tape).

As we have seen, different storage schemes can be used when samples need to be partitioned into groups of a smaller size. Consider the case in which several samples of 20 bits are stored in 8 bit groups. One scheme might pad the 20 bit number with four zeros t o make a 24 bit word. Each byte of the 24 bit word is then stored sequentially:

Byte Number	Bit number 76543210	Usage	
1	EEEEUUUU	Sample 1	4 bits exponent, 4 bits unused
2	SDDDDDDD	Sample 1	Sign and top 7 bits of fraction
3	DDDDDDDD	Sample 1	Sign and bottom 8 bits of fraction
4	EEEEUUUU	Sample 2	4 bits exponent, 4 bits unused
5	SDDDDDDD	Sample 2	Sign and top 7 bits of fraction
6	DDDDDDDD	Sample 2	Sign and bottom 8 bits of fraction
7	EEEEUUUU	Sample 3	4 bits exponent, 4 bits unused
8	SDDDDDDD	Sample 3	Sign and top 7 bits of fraction
9	DDDDDDDD	Sample 3	Sign and bottom 8 bits of fraction
10	EEEEUUUU	Sample 4	4 bits exponent, 4 bits unused
11	SDDDDDDD	Sample 4	Sign and top 7 bits of fraction
12	DDDDDDDD	Sample 4	Sign and bottom 8 bits of fraction

FIG. 1. A possible storage scheme for 20 bit floating point data samples

This storage method is wasteful, since 4 bits are wasted for every data value. One popular method for packing data samples is to store four samples at a time: first storing four four-bit exponents followed by four 16 bit functions.

Byte Number	Bit Number 76543210	Usage	
1	EEEEEEEE	Sample 2 Sample 1	Exponent (4 top bits) Exponent (4 bottom bits)
2	EEEEEEEE	Sample 4 Sample 3	Exponent (4 top bits) Exponent (4 bottom bits)
3	SDDDDDDD	Sample 1	Sign and 7 most significant bits
4	DDDDDDDD	Sample 1	8 least significant bits
5	SDDDDDDD	Sample 2	Sign and 7 most significant bits
6	DDDDDDDD	Sample 2	8 least significant bits
7	SDDDDDDD	Sample 3	Sign and 7 most significant bits
8	DDDDDDDD	Sample 3	8 least significant bits
9	SDDDDDDD	Sample 4	Sign and 7 most significant bits
10	DDDDDDDD	Sample 4	8 least significant bits

FIG. 2. A storage scheme for 20 bit floating point using exponent packing

Storage scheme: Defines how several data values are stored into memory. Most often, data are stored sequentially, but packing several exponents together and several fractional parts together can sometimes be more efficient.

From the discussion presented, it is clear that there are hundreds of permutations of data storage schemes. Many of the possible permutations are not used, because they are not useful, and because the cost of supporting several data types in data processing systems is expensive. At this time, there are probably less than 100 schemes in-use in the whole field of computing. Of these, only a quarter are commonly used for storing or processing seismic data. If we ignore the byte-order attribute we can reduce the number of commonly used binary formats to small number.

Table 4. Data representations in common use in seismic acquisition and processing

Data type		Encoding scheme				Storage Scheme	Common Name	Standard
Generic Type	**Total Bits**	**Exp. Base**	**Exp. Bits**	**Frac. Bits**	**Exp. Excess**			
Real	8	4	2	5	0	sequential	8 bit quaternary exponent	SEG-D
Real	8	16	2	5	0	sequential	8 bit hexadecimal exponent	SEG-D
Real	16	4	3	12	0	sequential	16 bit quaternary exponent	SEG-D
Real	16	16	2	13	0	sequential	16 bit hexadecimal exponent	SEG-D
Real	20	16	4	15	0	10 byte packed	20 bit floating point	SEG-D SEG-Y
Real	32	16	7	24	64	sequential	IBM 370 floating point	SEG-Y SEG-D
Real	32	2	8	23	127	sequential	IEEE 32 bit floating point	SEG-2
Real	64	2	11	52	1023	sequential	IEEE 64 bit floating point	SEG-2
Integer	8	-	-	-	-	sequential	8 bit fixed point	SEG-D
Integer	16	-	-	-	-	sequential	16 bit fixed point	SEG-D SEG-Y SEG-2
Integer	32	-	-	-	-	sequential	32 bit fixed point	SEG-Y SEG-D SEG-2
Integer	64	-	-	-	-	sequential	32 bit fixed point	SEG-2
Integer	24	2	8	15	0	sequential, 32 bit aligned	32 bit fixed point with gain values	SEG-Y

Describing a binary format using four different parameters is somewhat cumbersome. By grouping sets of parameters together we can describe most of the common binary formats using only 24 combinations of these four parameters. The parameter which tends to vary most from system to system is "byte order". To reduce the number of combinations we can exclude "byte order" as one of the parameters that makes up a combination. It is proposed that these combinations be called "data representations".

Data Representation: A combination of binary format parameters including: generic data type, encoding scheme, and storage scheme. Data representations are independent of byte order.

Using this definition of data representation, any binary format can be specified by a data representation and byte order.

CASE STUDY: SEG-Y FORMAT

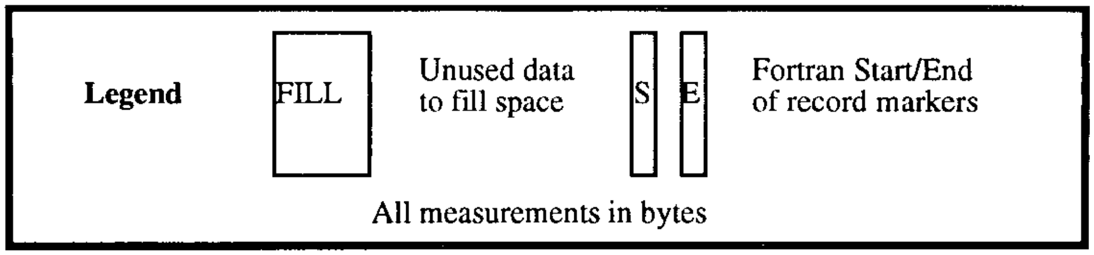
In 1975 the SEG Technical Standards Committee published the SEG-Y standard for demultiplexed field tapes. It is useful to study the way the SEG-Y standard has been implemented by software vendors, since it is a illustrates how problems can arise when a data format standard is not adhered-to. A large percentage of files that are called SEG-Y format are in fact, non-compliant. This frustrates users who may not understand the details of data formats, and are faced with the problem of working with these files on a daily basis.

The SEG-Y standard specifies that data samples should be stored using IBM 370 encoding schemes and byte order. Unfortunately, most current computers, including new IBM PC's and IBM workstations, use IEEE floating point encoding. Even though programmers who write software applications that generate SEG-Y format data should strive to conform to the SEG-Y standard, many do not. Due in part to processing-speed considerations, and more commonly, programming ease, many programmers choose ignore the strict binary encoding specifications in the standard. As a result, a large amount of software stores sample values in SEG-Y files using the default binary format of the host computer. Since most processing is no longer performed on IBM mainframes, many software packages produce non-conformant SEG-Y files.

More programs would conform to the SEG-Y standard if there was a method for simple and efficient binary format translation. The binary conversion library presented in this paper addresses this need.

There are other common problems with the SEG-Y format that are unrelated to binary format. Since the SEG-Y standard was designed for magnetic tape, there is no provision in the standard for storing SEG-Y format information onto non record-oriented storage media such as disk files (Figure 3). The most common implementation of SEG-Y on disk completely ignores record structures (Figure 4). Many implementations of Fortran simulate a record structure on disk files by interspersing record markers throughout output data files. Unfortunately, this unwanted feature is often impossible to defeat. As a result, many Fortran programs generate data files which contain Fortran record information (Figure 5). Naturally, compatibility problems arise when these tainted SEG-Y files are used as input to programs which do not expect Fortran record information. Constant-length records constitute a significant deviation from the SEG-Y standard.

Figure 6 shows another kind of SEG-Y variant. Fortran programs which use fixed-record length files for SEG-Y output produce this kind of file. These files not only contain unwanted Fortran record markers, but also padding to extend all records to a constant length.


Legend
FILL
Unused data to fill space
S
E
Fortran Start/End of record markers
All measurements in bytes

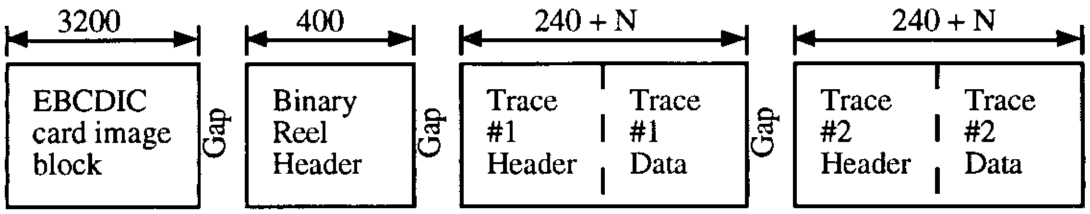


FIG. 3. SEG-Y on tape with inter-record gaps

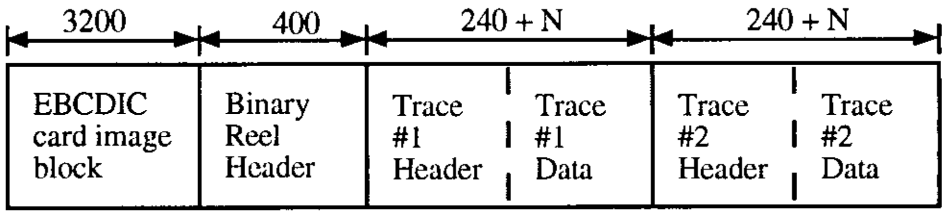


FIG. 4. SEG-Y on disk with no inter-record markers

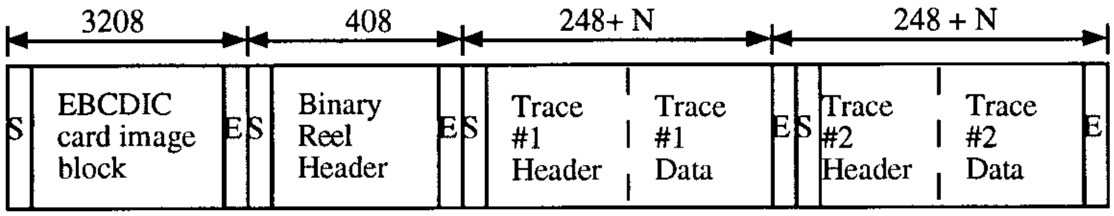


FIG. 5. SEG-Y on disk with Fortran start (S) and end (E) record markers

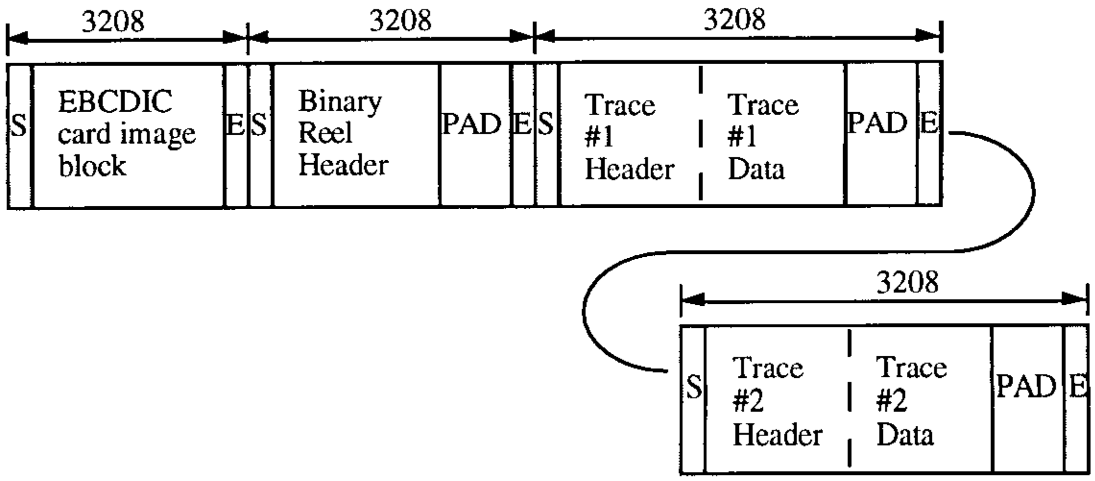


FIG. 6. SEG-Y on disk with Fortran record markers and record padding

BINARY FORMAT CONVERSION LIBRARY

A binary format conversion library was written to solve the problem of processing SEG-Y data on systems that use different floating point encoding schemes and byte orders than the SEG-Y files they process.

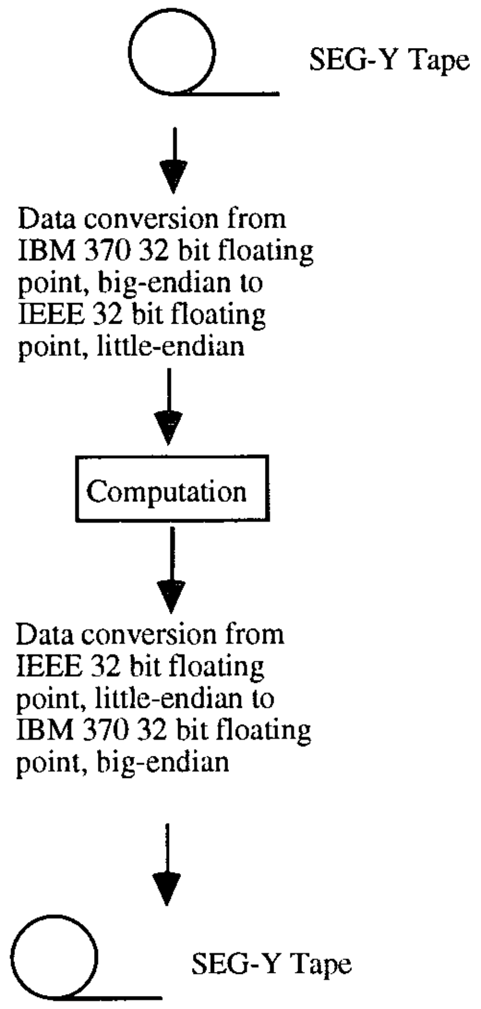

SEG-Y Tape
Data conversion from IBM 370 32 bit floating point, big-endian to IEEE 32 bit floating point, little-endian
Computation
Data conversion from IEEE 32 bit floating point, little-endian to IBM 370 32 bit floating point, big-endian
SEG-Y Tape

FIG. 7. Binary conversions necessary for processing SEG-Y data on a PC

A general purpose binary translation function library was deemed useful, not just for SEG-Y manipulating programs, but also for programs that deal with SEG-D and SEG-2 formats. A general purpose function was designed so that translation between any two binary representations and byte orders could be performed with a single function call. The binary translation function takes input from a memory buffer, containing one or more data samples in any binary format, and sends output to another memory buffer in a different binary format. There is no restriction on either the input or output buffer formats: neither needs to be in a format that is directly supported by the host hardware.

When designing the library, some overall design goals were chosen. Portability of code across different systems was considered extremely important. Not only did the function have to operate on systems with different byte orders and native encoding schemes, but it also had to compile on a 16 bit DOS compiler as well as 32 bit and 64 bit UNIX systems. Since there are, in theory, hundreds of possible formats, the library was constrained to handle only SEG-Y, SEG-2 data formats. Some of the more obscure formats found in SEG-D were not initially implemented, but allowances were made so that they could easily be added at a later date. The final two design goals were ease of use and speed.

To make the function easy to use, an abstraction called a *binary format conversion operator* was conceived. A conversion operator is an abstract data type that the end-user of the function library need not manipulate directly: a function is provided for setting-up the operator. This operator defines what binary conversion will be performed between the input and output data buffers. The conversion operator is defined by four parameters:

- input data representation
- output data representation
- input byte order
- output byte order

Before performing a binary conversion, the conversion operator must be obtained using the getBinCvtOper() function. The user supplies this function with the input buffer's binary representation and byte order as well as the desired binary representation and byte order for the output buffer. The function returns the appropriate data conversion operator to the user as its return value.

The actual binary conversion function is named binCvt(). The user supplies this function with a binary conversion operator and an input and output data buffer. The user also supplies the number of elements to be converted from one buffer to the other. The function has been designed so that the input and output buffers may be identical. In this case conversion is done in-place.

It may seem peculiar to have to first create a binary conversion operator, and then do the actual conversion. The two-step approach works well since binCvt() only needs four parameters. If all conversion related parameters were passed to a single function, the function would require seven parameters. Using a four parameter function improves code readability and, on some systems, increases speed. Another advantage of this calling convention is that the user's conversion request only needs validation once, and some pre-processing can be performed ahead of time so that multiple calls to binCvt() can run faster.

A final design feature of the binary conversion library is its ability to perform in-place translations. If the input data buffer and output data buffers are the same, the output overwrites the input.

The binary conversion library that is presented is currently under development at the CREWES Project. It is being incorporated into several CREWES programs that deal with SEG-Y and SEG-2 file conversion. This library will also be the basis for another high-level function library that will be used to generate SEG-Y and SEG-2 format files, as well as a number of vendor-specific format files. We hope to release this high-level multi-purpose I/O library in a future software release. The current CREWES software release will contain a prototypical binary conversion function library as part of the SG2TOSGY program. The files bincvt.c and bincvt.h contain the implementation of this library. If this software release is of interest, please refer to the paper entitled "CREWES software release" (Foltinek, 1993) in chapter 32.

CONCLUSION

Due to the large number of parameters involved in data format specification it is easy to understand why seismic data transfer is plagued with problems. A first step to reducing these problems is to improve software documentation to fully describe the data formats used for input and output files. The nomenclature presented in this paper could be used to aid in the explanation. Too often documentation of data formats is too ambiguous. "IEEE Floating point format" for example, does not deal with the issues of byte order or data storage scheme. Hopefully some of issues related to data file generation from Fortran will be corrected by new versions of Fortran which allow more flexible ways of writing non-record oriented files. In the mean time, programs which create data files which do not conform exactly to the SEG standards should make their non-conformance very clear to the user in the documentation as well as at run-time. Versatile and portable binary conversion function libraries, such as the one presented here, can also aid in generating standard conformant data files.

REFERENCES

Foltinek, CREWES Software Release: CREWES Research Report, v. 4

"Recommended standards for digital tape formats", Geophysics, v. 32, p. 1073-1084, v.37, p. 36-44

Vranesic, Z.G ., and Zaky, S. G., Microcomputer structures, Saunders College Publishing, New York, 1989, p. 699-703

Pullan, S.E., 1990, Recommended standard for seismic (/radar) data files in the personal computer environment, Geophysics, v. 10, p 1260-1271