GSE2.0/2.1: Some read/write routines in C that might come in handy

Authors: Stefan Stange, Andreas Greve, Dieter Stoll, and others.
Any remarks, questions, improvements to: Stefan Stange, stange@lgrb.uni-freiburg.de
No warranty can be granted for anything.

Format basics in: GSETT3 Documentation, Formats and protocols for data exchange, Annex 3. Group of Scientific Experts, Conference Paper 243. February 1995. Provisional GSE2.1, Message Formats & Protocols, Operations Annex 3. May 1997.

Contents

Read/write routines for waveform data with 6-byte compression (only that!) according GSE2.0/2.1. The routines are (see *gse_types.h*):

long *check_sum* (long *, int, long);
void *diff_2nd* (long *, int, int);
int *compress_6b* (long *, int);
void *write_header* (FILE *, struct header *);
void *read_header* (FILE *, struct header *);
void *rem_2nd_diff* (long *, int);
int *decomp_6b* (FILE *, int, long *);

They are collected in ***gse_functions.c***.
Furthermore, there are the auxiliary programs for character buffering as stated in *buf.h*, from which one needs:

int *buf_init* ();
int *buf_dump* (FILE *);
int *buf_free* ();

Together with some further programs for internal use they are collected in ***buf.c***.
Necessary include files are:

gse_header.h	with the description of the header data structure;
gse_types.h	with the routine description;
buf.h	with the buffer routines;
buf_intern.h	with additional declarations.

gse_driver.c	example driver program (main).

Compilation

The provided routines can be compiled with any decent C compiler (GNU for instance) and are tested under Solaris 2.6/2.7 and Suse LINUX 7.x. The program *gse_driver.c* gives an example of how to use the routines and which files to include. A simple make file for *gse_driver.c* could look as follows:

```
gse_driver: gse_driver.c buf.o gse_functions.o
        gcc gse_driver.c -o gse_driver buf.o gse_functions.o
buf.o: buf.c
        gcc -c buf.c
gse_functions.o: gse_functions.c
        gcc -c gse_functions.c
```

Usage

The routines are for 6-byte compressed data only! 2nd differences should be used because this is common practice.

Reading GSE2 data:
6-byte compressed GSE2 data are ASCII readable, hence, just open the data file first and then continue with searching and reading the first header line (which starts with WID2) by invoking *read_header*. This stores the header information from the file into a structure (which has to be declared by *struct header <name>*;). The elements match the GSE2.0 convention and are named as given in *gse_header.h* and *gse_driver.c*.
From the header we know the number of samples to be expected and can allocate space accordingly.
It follows: *decomp_6b* which looks for the line beginning with DAT2, extracts the compressed data, decompresses, and stores the data (long integers) into the allocated space (*data2 in *gse_driver.c*).
Now, remove the 2nd differences via *rem_2nd_diff*. Attention! *rem_2nd_diff* alters the data (the result is returned in the same vector, there is no need to provide additional memory).
The next line to read from the data file should be the checksum line. But, there might be an additional blank line. This is taken care for in *gse_driver.c* . The checksum read can be compared to the checksum computed from the data (after removing the 2nd differences!) by *check_sum* . The input value for *check_sum* has to be zero!
For the restoration of the original data (float for instance) CALIB and CALPER have to be considered. This is a common source of misunderstanding and neglect!
GSE2.0 prescribes a data unit of nm (nanometer, that is proportional to displacement). We now think of broadband or other modern registrations where the transfer function shows a noticeable flat part. Well within this passband the seismogram can be regarded as ground motion (displacement, velocity, or acceleration) and we may annotate SI units (m, m/s, m/s^2). If we had, for instance, displacement proportional data the formula for the restoration of the original seismogram z from the GSE2 counts is rather simple:

$$z[m] = GSEcounts \cdot calib \cdot 10^{-9}$$

For velocity data a symbolic differentiation has to be performed at the fixed (eigen-) period CALPER. Hence, the seismogram restoration formula reads:

$$z[m/s] = GSEcounts \cdot calib \cdot \frac{2\pi}{calper} \cdot 10^{-9}$$

Last but not least, an acceleration seismogram is restored from GSE2.0 accordingly:

$$z[m/s^2] = GSEcounts \cdot calib \cdot \frac{4\pi^2}{calper^2} \cdot 10^{-9}$$

Please be aware, that this is NOT a true ground motion restitution but only the restoration of the original seismogram. Restitution has to be performed by means of pole-zero information which is not the topic of this treatise.
How do we know whether the data are displacement, velocity, or acceleration, or even pressure, temperature, or something else exotic? Without pole-zero information there is no way to know from the GSE2 header line. In *gse_driver.c* we simply used AUXID as a flag (DIS, VEL, or ACC). This is only a work around, but nevertheless, simplifies the reading procedure (note: GSE1 provided such a flag).
A GSE2 file may be a volume and contain several data segments. To read these we just start over with *read_header* without closing the input file meanwhile.
Note: For GSE2 reading only, we do NOT need the routines from *buf.c* and the include files *buf.h* and *buf_intern.h*

Writing GSE2 data:
Data to be stored in GSE2 have to be integer (long). While no problem for digitizer output (counts), in other circumstances, the data must be converted into nm (nm/s, nm/s^2). This can easily be done by inverting the appropriate restoration formula (see above). But, to conserve the resolution of the data, we have to consider the integer quantization and the clipping of the GSE2 format. The 6-byte compression clips at 2^{27}-1. Keeping in mind that this is valid for the 2nd differences of the data the original clipping for decently sampled data may reach the integer resolution (32 bit). Taking no risk, we propose to quantize the data with 24 bits. That is, the maximum data value z_{max} should equal 2^{24}. (Of course, we could use 20 or 26 bits, too). For velocity data (as an example) this consideration results in:

$$\frac{z_{max}[m/s] \cdot calper}{2\pi \cdot calib} \cdot 10^9 \equiv 2^{24}$$

from where we can calculate the CALIB factor:

$$\text{calib} = \frac{z_{\max}[\text{m/s}] \cdot \text{calper}}{2\pi \cdot 2^{24}} \cdot 10^9 \approx \frac{z_{\max}[\text{m/s}] \cdot \text{calper}}{2\pi} \cdot 59.6$$

With CALIB the GSEcounts can be computed via:

$$\text{GSEcounts} = \frac{z[\text{m/s}] \cdot \text{calper}}{2\pi \cdot \text{calib}} \cdot 10^9$$

The data maximum $z_{\max}$ can be determined in two alternative ways: Either, it is the maximal value of each individual seismogram; this would "blow up" seismograms with tiny amplitudes to the entire dynamic range. The advantage is that we could convert any seismogram to GSE without further consideration. Or, $z_{\max}$ is the theoretical full scale value of a recording system which is thereby mapped into GSE2. Each seismogram would have the same CALIB factor.

Finally, with the data handy, the remaining procedure is rather simple:
First, compute the checksum (*check_sum*). At the beginning, the input value must be zero. The functionality of the original checksum algorithm was slightly modified to allow chunks of data to be merged in one GSE2 file. This is a concession to block data formats like the Lennartz-Mars88 format. For consecutive data blocks the input to *check_sum* is then the previously computed checksum value. The final result equals the checksum of the complete data set.
Next, we compute the 2nd differences (*diff_2nd*). Again, this can be done for data blocks, where the input parameter controls whether we have a new data set (CONT_FLAG=0) or the continuation of a data stream (CONT_FLAG$\neq$0). The result is returned in the same data vector.
Now, at the latest (!), we have to initialize the character buffer by invoking *buf_init();* .
The 3rd step then is the character encoding via *compress_6b*. Nothing has to be said here, besides, of course, several consecutive calls to *compress_6b* are possible.
If the data processing is completed we can fill the header structure and then need an open output file where we

write the header	(*write_header*),
write the data	(*buf_dump*),
write the checksum	(*fprintf (fp,"CHK2 %8ld\n\n",chksum);*)

together with the closing blank line.
At the end it is recommended to clean up via *buf_free();*.

If we want to append another data series to the same GSE2 file we just start over with *buf_init();* and the new data and header information, and follow the procedure as before. If finished we just close the file and are done.