

Norwegian National Seismic Network

Technical Report No. 18



**A toolbox for reading SEED and MiniSEED and writing
MiniSEED**

Prepared by

Rodrigo Luciano Pereira Canabrava

Dept. of Earth Science, University of Bergen

Allégt. 41, N-5007 Bergen, Norway

Tel: +47-55-583600 Fax: +47-55-583660 E-mail: seismo@geo.uib.no

December 2004

INTRODUCTION.....	3
BRIEF DESCRIPTION OF SEED.....	4
<u>SEED (V,A,S,T) HEADERS.....</u>	<u>4</u>
<u>THE DATA RECORDS.....</u>	<u>6</u>
A DESCRIPTION OF WHAT IS A MINI-SEED.....	8
<u>THE BLOCKETTE 1000.....</u>	<u>8</u>
GENERAL OUTLINE OF THE READING ROUTINES.....	12
<u>EXTERNAL FUNCTIONS.....</u>	<u>12</u>
<u>DIAGRAMS FOR FUNCTIONS OF THE FILE SUMMARIZATION.....</u>	<u>13</u>
<u>DIAGRAMS OF FUNCTIONS FOR READING CHANNEL DATA.....</u>	<u>16</u>
<u>DECLARED CONSTANTS.....</u>	<u>17</u>
<u>KNOWN PROBLEMS.....</u>	<u>17</u>
COMPILING ISSUES IN FORTRAN.....	18
ISSUES ON WINDOWS COMPILERS.....	19
LOG OF THE DIFFERENT FILES TESTED.....	20
<u>MINI-SEED FILES GENERATED BY RDSEED.....</u>	<u>20</u>
<u>QUANTERRA FILES.....</u>	<u>21</u>
<u>ORFEUS.....</u>	<u>21</u>
<u>IRIS.....</u>	<u>21</u>
<u>NANOMETRICS.....</u>	<u>22</u>
<u>GRC (UNKNOWN SOURCE).....</u>	<u>22</u>
<u>OTHER FILES.....</u>	<u>23</u>
WRITING MINI-SEED.....	24
<u>EXTERNAL FUNCTIONS IN THE FORTRAN LIBRARY.....</u>	<u>24</u>
<u>EXTERNAL FUNCTIONS IN THE C LIBRARY.....</u>	<u>24</u>
<u>WRITING STEIM 1.....</u>	<u>25</u>
<u>DIAGRAM OF THE WRITING FUNCTIONS.....</u>	<u>26</u>
CONCLUSION.....	27
REFERENCES.....	28

Introduction

The project of the internship consisted of creating reading/writing routines for files in the format known as Mini-SEED. This format is a simplified set of the SEED standard (Standard for the Exchange of Earthquake Data), defined by the IRIS Consortium (Incorporated Research Institutions for Seismology).

The routines were meant to be used by the institute's softwares, SEISAN and SEISLOG. The routines should have interface both for C and Fortran programs. They should also be able to run in Sun, Linux and Windows platforms.

The first step was to develop a library of functions in Fortran 77, that was supposed to be used by SEISAN, but that should be stand alone, so it could be used by other institutions in order to integrate the library in their own softwares.

After accomplishing the task of reading Mini-SEED files, we invested more time in trying to understand the SEED format as a whole. As a result we managed to make the reading routines also read some kinds of SEED files. We tested the reading routines with files generated by the most important institutions generating SEED files these days: IRIS, and ORFEUS.

The next step was to make functions to write Mini-SEED files. First, one set of routines was included in the Fortran library. Then, another set was developed in C, for use in SEISLOG. The library in C works in a slightly different way than the one developed in Fortran, so it could fit better for the purposes of SEISLOG. These differences are described in the appropriate section.

In the next two sections, there is a brief description of the SEED and Mini-SEED formats. This description is supposed to help to understand the general outline of both types of files, but it is not meant to substitute the Reference Manual. Indeed, the reader will be often asked to refer to the Manual for the outline of different blockettes, and for the description of the compression routines.

After that, we describe the reading library, explaining how to use the functions, and how they interact internally in order to accomplish their task. We also detail compiling issues that we experienced in writing a code that could work on all platforms.

Brief description of SEED

A SEED file, or volume, is divided in blocks of equal size. Most typically this size is 4096 bytes. Each block starts with the following three fields:

Field name	Type	Size (bytes)
Sequence number	ASCII	6
Header type	ASCII	1
Continuation code	ASCII	1

Table 1 - Common fields of each file block

The sequence number is the number of the block in the file. The first block has the sequence number “000001”. The header type is a letter that identifies the type of information in the block:

V	Volume Index Control Headers
A	Abbreviation Control Headers
S	Station Control Headers
T	Time Span Control Headers
D, R or Q	Data Records

Table 2 - Header Types

The continuation code tells if the current block is a continuation of the previous one. Its use will be better explained later, after the header types are better described.

The first 4 block types are SEED headers; the 5th one holds the data (the waveform). Each kind of block holds a different kind of information. So, a Volume Control Header holds information that describes the file itself; the Station Control Headers hold information of stations, their channels, and response data. The Time Span Headers have time indexes for the Data Records (the ones with the data). The Abbreviation Control Headers hold general description that gives the SEED format the power of being so extensible (and hard to deal with).

The blocks from each type come together in the file: they cannot be intercalated. This means that the file starts with a block of type V (usually there is just one); then a sequence of blocks of type A; then a sequence of type S; another sequence with type T; and the final sequence of type D, R or Q.

SEED (V,A,S,T) Headers

The SEED main headers are completely in ASCII. Their information is organized in structures called blockettes. A blockette is a header with a specific structure, and a number that identifies its type. The first field (3 bytes) of each blockette is the type. The second field (4 bytes) tells the blockette length, because most of the blockette types have variable length. The following fields depend on the blockette type.

The first block in the file must be of type V, and the first blockette in its block must be either the blockette 5, 8 or 10. The most important piece of information that can be obtained from those blockettes is the “Logical Record Length”, or block size

of the file. For information on the order of fields in these blockettes, see [1]. The first block of a file would be something like:

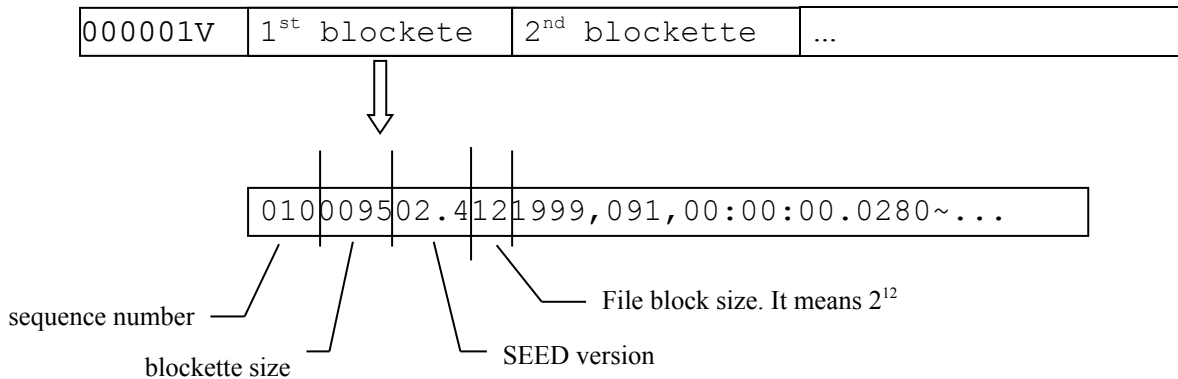


Figure 1 - A typical first block of a SEED file.

If the blockettes don't fill the whole block, the rest of it is filled with space characters (ASCII 20). On the other hand, if the sequence of blockettes is too big to fit in one block, another block of the same type is created, and the continuation code is set to the symbol * (ASCII 42), as shown in the figure below:

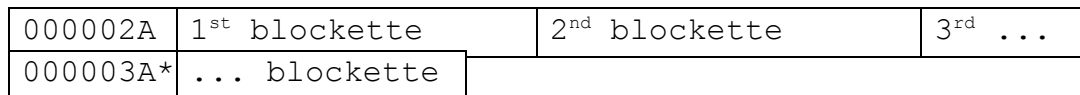


Figure 2 - The use of a continuation blockette

In the example, the 3rd blockette would not fit in the block 000002. The characters that fit this block are put in there, making use of the whole block. The rest is put on the following block, which is identified to be a continuation of the previous one.

In summary, a typical SEED file would look like this:

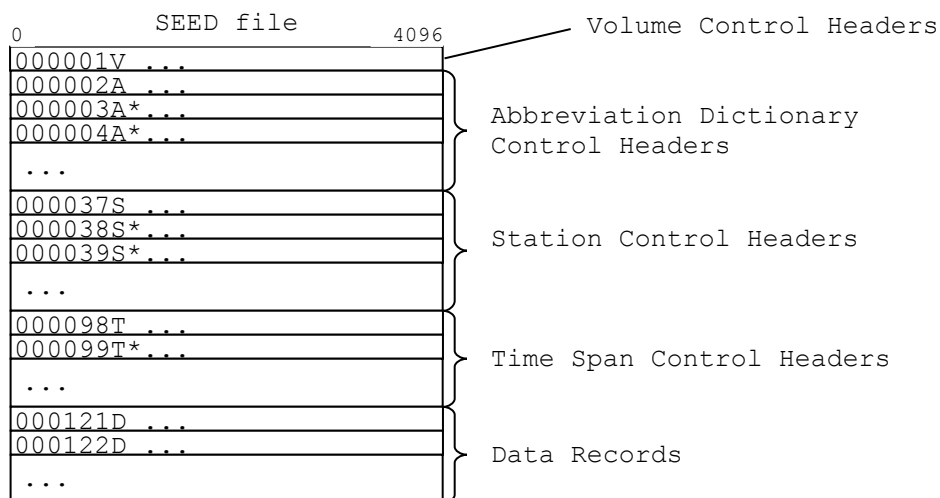


Figure 3 - The SEED file

The Data Records

The Data Record is where the waveform data is stored. Each Data Record has a small waveform record that corresponds to a small period of time for one channel of one station. Each channel usually uses several Data Records in the file.

The Data Record also has some headers. In those headers we can identify the channel, station, start time, sample rate and other information referring to the waveform it holds. The Data Records are mainly binary, except for a few ASCII fields in the Fixed Header. The fixed header corresponds to the first 48 bytes of the Data Record. In the table below there is a list of the fields of the fixed header. For a description of what each means, refer to [1]:

	Field name	Type	Length (bytes)
1	Sequence number	ASCII	6
2	“D” “R” “Q”	ASCII	1
3	Reserved byte	ASCII	1
4	Station code	ASCII	5
5	Location identifier	ASCII	2
6	Channel identifier	ASCII	3
7	Network code	ASCII	2
8	Record start time	BTIME	10
9	Number of samples	Unsigned word	2
10	Sample rate factor	Word	2
11	Sample rate multiplier	Word	2
12	Activity flags	Unsigned byte	1
13	I/O flags	Unsigned byte	1
14	Data quality flags	Unsigned byte	1
15	Number of blockettes that follow	Unsigned byte	1
16	Time correction	Long	4
17	Offset to beginning of data	Unsigned word	2
18	Offset to beginning of first blockette	Unsigned word	2

Table 3 - Fixed section of data header (48 bytes)

The first 3 fields (8 bytes) match the first 3 fields in each block in a SEED file, as specified in Table 1. Note, though, that the third field in this case is a reserved field, and should always contain a space character. The first 7 fields (20 bytes) are in ASCII. The rest are binary. The 8th field is defined as a structure referred to as BTIME, in the Manual [1]. It contains the start date and time of sequence.

Blockettes

Besides the fixed header, the Data Records can contain some blockettes. They are additional headers to the Data Record, and each one has an id number and specific fields. They are different from the blockettes in the main SEED headers in the following aspects:

- the blockettes in the Data Record are binary, while the ones in the main SEED headers are in ASCII;

- the blockettes in the Data Record cannot be split in two blocks, because each blockette only refers to one Data Record, and the Data Records cannot be split in two blocks;
- the blockettes in this case are not necessary contiguous in the file. Each blockette has a field that points to the next blockette.

A Data Record would look like the figure below:

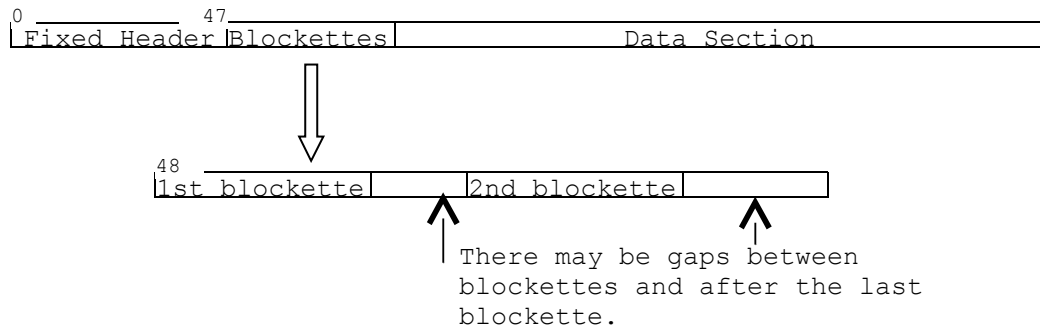


Figure 4 - Structure of the Data Record

The Data Section

The Data Section is where the waveform data really is. It occupies the rest of the block, after including the fixed header and the blockettes. The data can be encoded in different ways, but by far the most used ones are: 32-bit integers, Steim 1 and Steim 2 compression. The beginning of the Data Section is given by the field 17 of the fixed header. This field is the number of the byte where the data starts, counting as zero the first byte of the block.

Steim 1 and Steim 2 compression are based on differences from one sample to another. Basically, they hold the first sample X_0 , and then the differences between samples ($X_i - X_{i-1}$). In both cases, the information is organised in frames of 64 bytes. For this reason, the beginning of the Data Section in these cases must be in a byte multiple of 64. For a good understanding of these compression formats, see Appendix B in [1].

A description of what is a Mini-SEED

A Mini-SEED file corresponds to a SEED file containing only Data Records. The fixed header of the Data Record has the information that characterises the waveform data, like station name, channel, sample rate, etc. So, the SEED headers are not necessary, unless you need the response data, a quick overview of file contents and more channel information. There is only a little extra information needed to read the waveform data that is not on the Data Record fixed header: encoding format, byte order and block size. In order to be able to use a file without SEED main headers, a blockette was created in order to keep these three pieces of information inside each Data Record. This is the blockette 1000.

The blockette 1000

Without the main SEED headers, there is the need to include a blockette in the Data Records that gives the following mandatory information:

- record length;
- encoding format (Steim 1, Steim 2, 32-bit integers, etc.)
- byte order of binary fields (if SUN, or PC).

The blockette 1000 was designed to hold this information in the Data Record. It is mandatory for Data Records in Mini-SEED files, but it is also commonly used in SEED files. This blockette has 8 bytes, written in binary. This blockette has the following structure:

	Field name	Type	Length (bytes)
1	Blockette type	Unsigned word	2
2	Offset to beginning of next blockette	Unsigned word	2
3	Encoding format	Byte	1
4	Word order	Byte	1
5	Data record length	Unsigned byte	1
6	Reserved	Unsigned byte	1

Table 4 - Blockette 1000 (8 bytes)

The first 2 fields are common to every blockette.

The 1st field allows identifying which type of blockette it is. In the current case, it will contain the value 1000.

The 2nd field allows that each blockette points to the next one in the same Data Record. This offset is the number of bytes from the beginning of the data record. The last blockette contains zero. This allows that the blockettes can be read as if they were in a list. Each blockette points to the next, and the last one points to zero (meaning the end).

The information of these three fields is the information that is missing when there are no SEED headers.

A Mini-SEED file can be organised according to the structure:

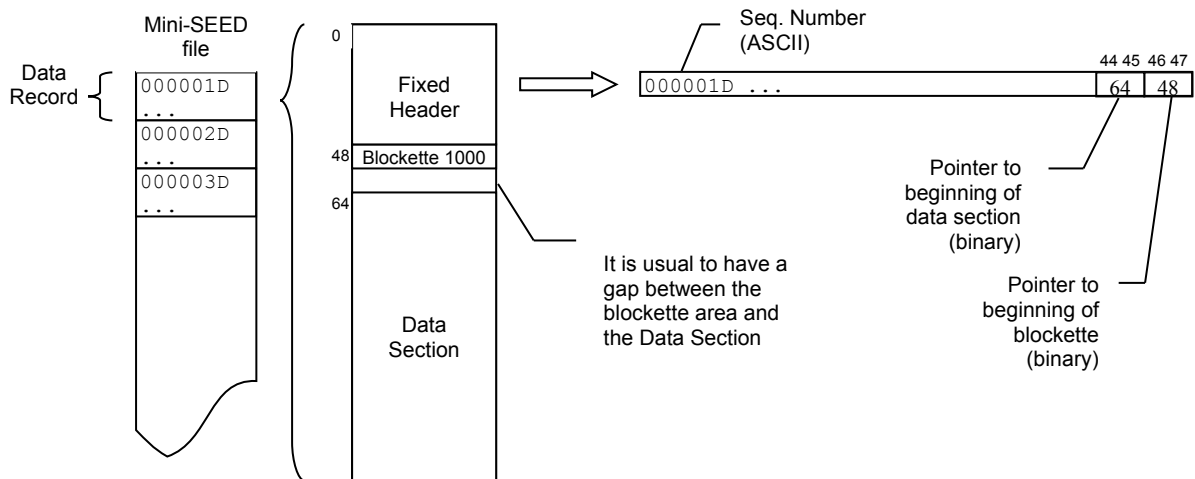


Figure 5 - The simplest Mini-SEED structure

Besides the blockette 1000, there can be other blockettes. The most widely used on both SEED and Mini-SEED files, besides the 1000, are the 100 and the 1001. Refer to [1] for a description.

Record length (block size)

In a SEED file, the record length is on the first block of the file either in the blockette 5, 8 or 10. In a Mini-SEED file, it is in the blockette 1000, in each Data Record. In both, the value of the field is a power of 2. So, if it holds 12 (by far the most common option), the block size is $2^{12} = 4096$ bytes.

Encoding format

The encoding format in blockette 1000 is a one-byte field, where each number specifies a format. For example, Steim 1 is code 10, Steim 2 is 11 and 32-bit integers, 3.

In the SEED file, this information is found in the blockette 30. It has the following fields:

	Fields	Type	Size
1	Blockette type	ASCII	3
2	Length of blockette	ASCII	4
3	Short descriptive name	ASCII variable length	1 to 50
4	Data format identifier code	ASCII	4
5	Data family type	ASCII	3
6	Number of decoder keys	ASCII	2
7	Decoder keys	ASCII variable length	any

Table 5 - Blockette 30

Here, it is a lot more complicated to define the encoding format. It is actually described in the “Decoder Keys” field, using symbols described in Appendix D [1]. In order to retrieve this information in the most robust way, it would be necessary to

parse the description and build the appropriate decoder, which is unviable if only a few formats are actually used.

Better than having such trouble, it is possible to discover the correct format by checking the “Short Descriptive Name” field.

It is important to note that different channels might be using different encoding formats, in the same file. In a Mini-SEED file, this is not a problem, since every Data Record has the information on the blockette 1000. But in SEED files, it means there will be more than one blockette 30 in the main headers. In order to figure out which channel is using each format it is necessary to cross-reference blockette 52 (Channel Identifier Blockette) with the 4th field of this blockette.

Byte order

Most of the Data Record is written in binary mode. In fields that are 2 or 4 bytes long, different computers interpret them in a different way. It means that a file written on Solaris must have their fields converted when they are being read on a PC, and vice-versa.

This information can be found in the SEED headers, in blockette 52; or in blockette 1000, in a Mini-SEED file.

In blockette 1000, this field is binary, but it has only 1 byte so, in theory, there is no problem in reading it directly in order to determine if the 2 and 4-byte fields must be swapped or not.

Then, in order to determine the byte order of the file, you must find blockette 1000, and read that field. The problem is that you need to know the byte order in order to find the blockette 1000. Why? Because the fields “Offset to Byte Order” and Blockette Type may be in a different byte order! Note that the Standard does not guarantee that the blockette 1000 will be the first one to follow the fixed header, so the program needs to go from one blockette to another looking for the blockette 1000.

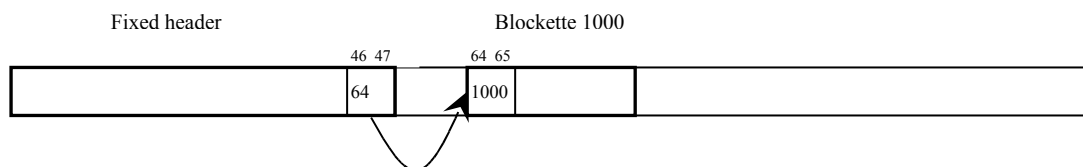


Figure 6 - Data Record in the right byte order

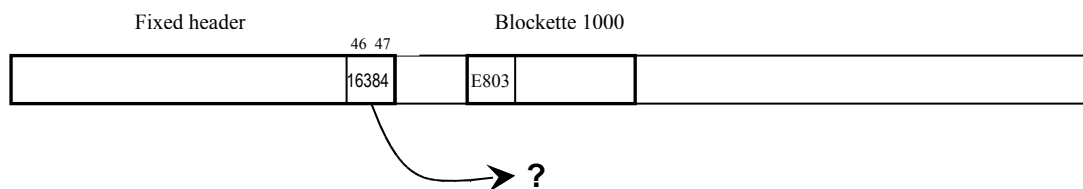


Figure 7 - Data Record in the wrong byte order

The examples in figures 2 and 3 show a Data Record in which the Blockette 1000 is not immediately following the fixed header. It starts at byte 64. When the byte order is taken correctly – that is, the byte order of the file is the same as the computer – the last field of the fixed header holds the number 64. That can be seen on figure 2.

Figure 3 shows a case when the byte order is inverted: the offset to the next byte is interpreted the wrong way, and then points to a byte that is outside the Data Record.

This shows that the byte order of the file must be determined before reading the blockette 1000. This field of the blockette is useless. As a matter of fact, IRIS SEED Reader (RDSEED) generates Mini-SEED files on Linux with Linux byte order, but sets this field to Solaris byte order, so even if it could be read, it cannot be trusted.

The RDSEED program tries an alternative way to determine the byte order. It checks if the field that holds the Year information (in the Start Time field) is between 1950 and 2010. (Probably, when we arrive in the year 2010, they will need to change the interval to 2020!). They could keep postponing the problem until 2056. The “byte order inverse” of 2056 is 2056, so we would be unable to determine if the word order is wrong. The file would be mistakenly considered to be in the right byte order, and the file reading would fail.

Defining the byte order

In order to determine the byte order, we should check the value of a binary field with 2 or 4 bytes. So, the fields Hour, Minute or Second cannot be used.

The Year will have a problem when we arrive in the year 2056, as described above. The day will also have problems in some values of the scope, because:

$\text{swap}(256) = 1$ and

$\text{swap}(257) = 257$,

so both the original value and the swapped value are within the limits: $\text{day} \leq 366$.

The Fracsec field, which has a scope $0 \leq \text{fracsec} \leq 9999$, has the same kind of problems.

So far the best alternative, as used here, is checking the year, but the format is condemned to be redesigned in order to survive the “bug of 2056”.

General outline of the reading routines

The reading routines we developed have the following capabilities:

- read SEED and MiniSEED files;
- read multiplexed files;
- use direct access to read;
- can read sections of large files;
- the data can be in 32-bit integers, Steim 1 or Steim 2;
- can read files with different compression formats in different blocks (only if the blocks have blockette 1000);
- can read block sizes of up to 32 Kbytes.

The functions in the library were designed in order to be used in two steps. First, the file is summarized (by reading all headers in file), that is, we identify the number of channels, and the following information for each of them: station name, channel (component) name, start time, sample rate, number of samples, flag of timing quality, start and end position in the file. This summary is built by checking the information of the fixed header in the Data Records. It means that the SEED headers are mostly ignored. Only the block size and the encoding format are used, when there are SEED headers.

After summarizing the file, it is possible to read the waveforms for each channel. Since some files may cover periods of time that are too large for the buffer in the calling program, the library offers a way to read just an interval of the channel.

The use of the corresponding functions for each task is explained next.

External functions

There are only three functions that need to be called by programs in order to achieve the complete task. They are:

- *seed_contents(filename)*: this function summarizes the file, whether it is SEED or Mini-SEED. The summary is held on the common block */file_summary/*.
- *chop_chn(filename, channel, time, start_point, block)*: this function identifies the block where *time* occurs in the channel specified. For a matter of optimization, it is possible to give a *start_point* for the search. The function will start the search from the block given by the parameter *start_point*. If the user gives a start point that is smaller than the beginning of the channel (including, zero, one or a negative number), the function will start from the beginning of the channel. Note, this is normally not the beginning of the file.

The parameter *block* is an output value. It always returns a valid block number for that channel. Besides this variable, the function returns an integer value indicating the status of the search, according to the description below:

situation	return value	Block
time is found	0	block containing time
time previous than the beginning of the channel	1	beginning of the channel
after the end of the channel	2	end of the channel
start point is after the end of the channel	3	start point
time is before the start point	4	start point
time is in a gap	5	first block after gap

Table 6 - Return values of function *chop_chn*

A block is considered to contain the specified time if :

- *time* is bigger than the beginning of the channel, and
 - *time* is smaller than the end of the channel plus the time of one sample
- *seed_read_chn(filename, channel, buffer, start_point, end_point)*: this function reads a section of the specified *channel*, putting the samples in the given *buffer*. The section read is delimited by the *start point* and *end point*, which refer to block numbers. In case the *start point* happens before the beginning of the channel, the function start in the beginning of the channel. If the *end point* is after the end of the channel, the function reads until the end of the channel. Optionally, if both start and end point are zero, the function reads the whole channel. If the buffer is not big enough to hold all the samples, the function will read as many blocks as possible. In this case, an error message is set on the common block */error_handle/*.

Diagrams for functions of the file summarization

seed_contents

The function *seed_contents* is the function that should be called by the external programs, in order to summarize the file. This function finds the block size of the file, reads the seed headers (if any), determines the block size of the data records, and reopens the file if necessary, and summarizes the contents of the file. As a matter of fact, it accomplishes all that just by calling other functions. The diagram in the following figure shows which functions are called:

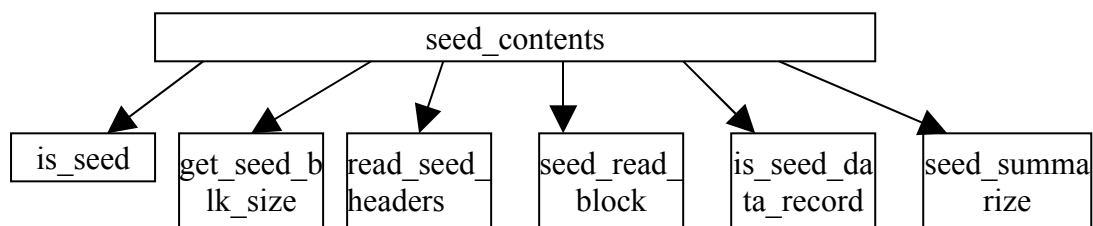


Figure 8 - Diagram of function *seed_contents*

is_seed

This function checks if the file is SEED (and not Mini-SEED). For that it calls one more function, which is shown in the next figure:

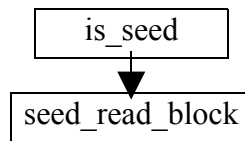


Figure 9 - Diagram of function *is_seed*

get_seed_blk_size

This function returns the block size in a SEED volume. It obtains this information by reading the first block of the file, which must be a Volume Header:

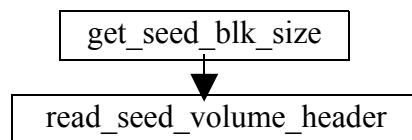


Figure 10 - Diagram of function *get_seed_blk_size*

seed_read_block

This function is the only one that actually accesses the file. Any other function that needs access to the file calls this one.

read_seed_headers

This function reads the SEED headers. Actually, at the moment it mostly skip the whole header, and just counts the number of blocks it occupies.

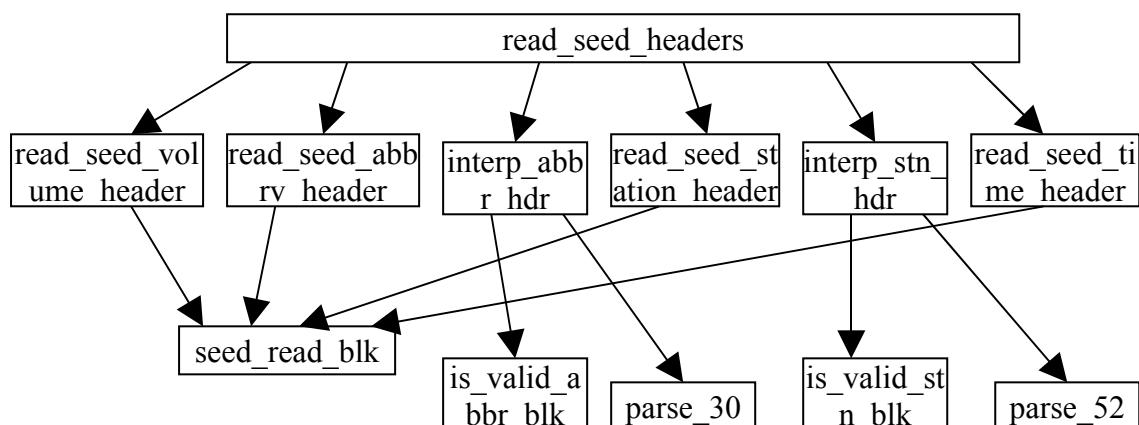


Figure 11 - Diagram of function *seed_read_headers*

The functions *read_seed*_header* read a block, and check if the block belongs to the corresponding type.

The functions *interp*_hdr* control the flow among blocks of the same type, going from one blockette to another within the corresponding section of headers. In case there is the need of reading different kinds of blockettes, this is the only function that needs to be changed.

The functions *parse_** parse the blockette of the corresponding number. In case there is the need of reading more blockettes from the main headers, more functions *parse_** should be created.

The functions *is_valid*_blk* validate the blockette types allowed in each block. This function is mandatory for defining the end of the blockettes in a block, since we don't know the number of blockettes in advance.

seed_summarize

This is a control function, that decides the execution flow between summarizing the file as having sequential channels or as having intercalated (multiplexed) channels.

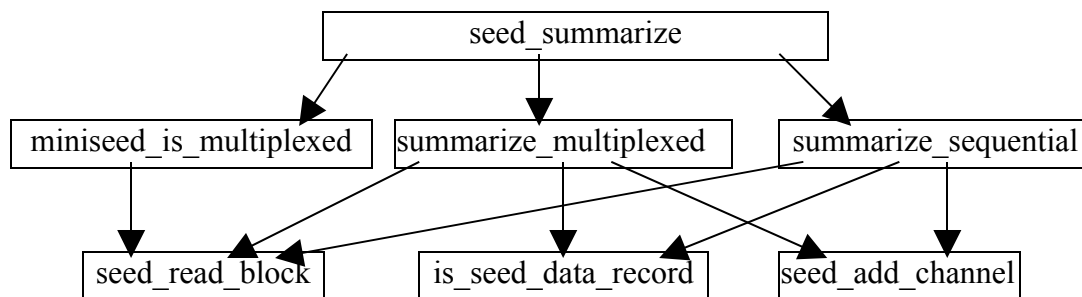


Figure 12 - Diagram of function *seed_summarize*

The function *miniseed_is_multiplexed* checks whether the blocks that belong to one channel come all grouped in the file, or if they come all intercalated.

The functions *summarize_multiplexed* and *summarize_sequential* run through all the Data Records, generating the file summary in the appropriate way.

is_seed_data_record

This function checks if the block read is a Data Record. Besides that, it checks the byte order, swaps the bytes, and read the Data Record's blockettes.

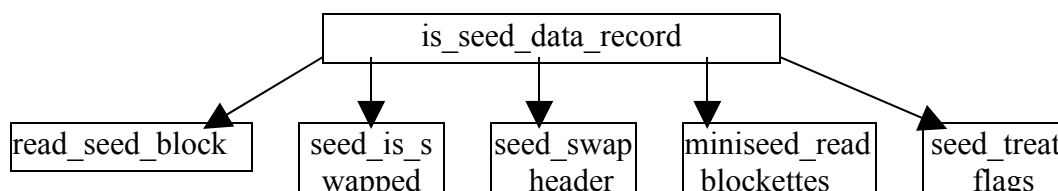


Figure 13 - Diagram of function *is_seed_data_record*

The function *seed_is_swapped* checks if the file byte order is different from the computer byte order (by checking the year, as it was described previously, in the

section “Checking the Byte Order”). The function *seed_swap_header* swaps the bytes of fields in the main header. The function *miniseed_read_blockettes* goes from one blockette to another, parsing it and extracting available information. The only blockettes that are checked so far in this part of the file are the numbers 100, 1000 and 1001. The function *seed_treat_flags* checks for some flags that were set on the Data Record main header (in fields 12, 13 and 14).

Diagrams of functions for reading channel data

chop_chn

This function finds a block number where a specific time is present. This function is used from calling programs using the library.

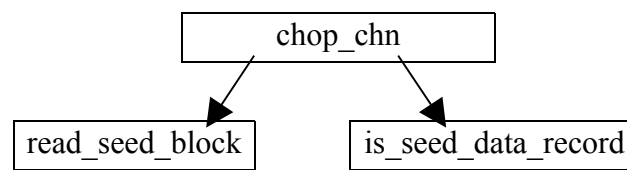


Figure 14 - Diagram of function chop_chn

seed_read_chn

This function is used to read the whole channel, or a part of it. This is also called from the outside programs, as described before. The function runs through all the blocks within the given interval, checking if the block belongs to the channel, and extracting the data. This behavior allows that this function works correctly for both multiplexed and sequential files. Obviously it will be faster for sequential files.

This function then calls the appropriate function to decompress the data. The encoding format may have been set previously, when reading the SEED main headers, or can be obtained by checking the blockette 1000. In case the data record has the blockette 1000, the information in it will overwrite the previous information. All this checking of the existence of the blockette 1000 is not done in this function itself, but it is done automatically when this function calls *is_seed_data_record*, in order to test if this block is a valid data record.

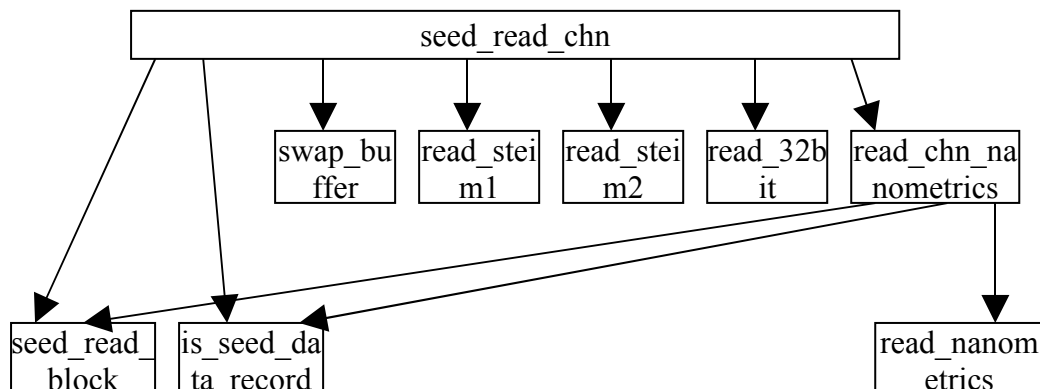


Figure 15 - Diagram of function seed_read_chn

If this function starts to decompress the channel as a regular Steim 1 and if, at any block, it determines a bad integrity constant, it assumes this file was generated by Nannometrics, and it tries to decompress it in the Nannometrics way (read the next section for a further description of the problem with Nannometrics files.). In this case, the control is transferred to the function *read_chn_nannometrics*. This function also runs through each block in the interval, but it always calls the same decompressing function: *read_nannometrics*.

The functions *read_nannometrics*, *read_steim1*, *read_steim2* and *read_32bit* decompress data from one block, putting it in the output buffer.

The function *swap_buffer* swaps the bytes of the whole data, before decompressing. The function *seed_read_chn* checks first if there is the need of swapping the data.

Declared constants

The library has a set of declared constants in the include files:

MAX_N_CHN: defines the maximum number of channels that will be supported.

MAX_SAMP: defines the maximum size of the data buffer.

MAX_BLK_SIZE: defines the maximum size of one block. In order to make the program accept bigger blocks, it is not only necessary to change this variable, but also to change the declared size of the variable *seed_record* (in the common */seed_data_record/*).

Known problems

We experienced some files that have time gaps in the channel data. This is a problem that was not predicted in the beginning, so the function *seed_read_chn* does not check if there is a time gap. However, the user can check this since header times of all blocks read are returned together with sample number of header times (variables *seed_blk_times* and *seed_blk_index*).

Our routines can only read files with different encoding formats in the same file if the Data Records have the blockette 1000. In case of SEED files, this blockette is not mandatory, so it is possible that someday people will start generating SEED files that will make our routines fail. In order to solve this problem we would need to read and interpret more of the SEED main headers.

Compiling issues in Fortran

In order to make a platform independent code, we had to skip many simplifications that are accepted on new Fortran compilers. Among other things:

- Fortran 77 doesn't have structures. Instead of it, we have several variables. In order to keep things simple (let subroutines have reasonable number of parameters), the record read from the file is on a common block (which works the same way as global memory).
- Sun compiler does not accept `integer*1`, so 1-byte fields should be read as `character*1` fields.
- F77 doesn't have cast between variables, and arithmetic operations are not allowed on characters. So, a tricky way to convert 1-byte character into integer is based on overloading the same memory space with 2 variables. It works as follows. Suppose we have a variable `character*1 char`, that we read from the file with the hexadecimal value `0x4A`. If we want to interpret it as an integer, we have to put it in an `integer*2` variable. So, we do:

```
character*2 c_int
integer*2 int
equivalence (int, c_int)
...
c_int(1:1) = char
```

The code above works according to the picture:

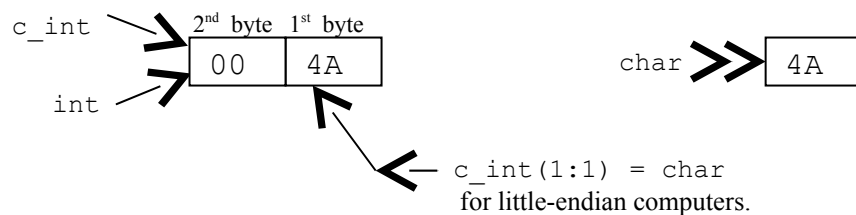


Figure 16 – Getting the integer value in a character variable

Now the variable `int` has the value we wanted. This is the way for little-endian computers. For big-endian, we do: `c_int(2:2) = char`.

Issues on Windows compilers

- When opening a file for direct access, the RECL attribute supposed to be informed in bytes. But with the Windows compiler used (Digital Visual Fortran 6.0), it interprets the argument as the size in words (32 bits). In order to make the Windows compiler interpret the number the right way, the following flag must be used to compile:

`/assume : byterec`

- In the logical attributes on Unix the value **.FALSE.** is zero. On Windows, it's the opposite. This difference introduced a bug in the code when trying to read the value "number of blockettes" from the fixed header of the Data Record. The expression: `if (num_blockettes)` would evaluate differently on Windows and the others. This problem was corrected by skipping this check: our routine just tries to read the blockettes, and if there is none, it will not find any.

Log of the different files tested

Different SEED and Mini-SEED file writers generate files with different particularities. That often forced us to change the way to read the files, so our program could work with all the different files tested. This probably covers most file types used. Here we will relate the problems each kind of file brought and the solutions adopted in our reading routines.

Mini-SEED files generated by RDSEED

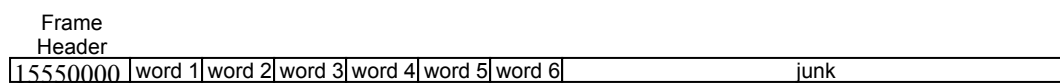
The files generated by RDSEED always have blocks of 4096 bytes, and are always compressed with Steim 1. The algorithm used with RDSEED in order to write the output file gives two problems for identifying a sequence of continuous-in-time blocks for each channel.

The first problem is that this program always appends the output to the file *mini.seed*. So, if the user runs RDSEED without care, the program may put two sequences of one station/channel in the same file, where they may not be continuous in time, or may have overlapping time.

On the other hand, if the user tries to extract a sequence from a SEED file that is too large for the buffer in RDSEED, it will break the sequence of that channel in the file. In other words: RDSEED gets as much data as it can hold, and writes to the file a sequence of blocks, with the sequence number starting at 000001. Then it gets another chunk of data from the same channel and writes another sequence, starting at 000001 again. Both sequences are actually only one continuous sequence from the same channel, without gaps or overlaps, and should be treated this way.

In order to solve both problems, identifying correctly continuous and non-continuous sequences, we decided to identify as continuous blocks, those that have the same station/channel and are adjacent in the file. It is not a flawless solution, but it is a lot easier than trying to check the time labels, since it is not uncommon to have imprecise time tags, that would demand statistical treatment to identify time sequences as continuous.

The two problems above are related to summarizing the file. But RDSEED presents another problem that we face when decompressing the data. This problem occurs in blocks that have less data than it's capacity, when sometimes RDSEED leaves wrong information in the frame header (see the Steim 1 compression).



Wrong header: the 4th hexadecimal digit is saying wrongly that there are samples on words 6 and 7 (see [1]), but actually the word 7 is garbage. If you consider this wrongly you'll have disagreements on the number of samples and on the reversal integration constant. This situation was actually found on files generated by RDSEED.

Figure 17 - A frame with wrong information

As a result, sometimes in the last block, our routines found a disagreement in the number of samples. In order to solve that, we changed the code in our program so it would stop immediately to decompress samples as soon as it got the number of samples expected.

Quanterra files

RDSEED files have all the blocks from one channel grouped together in the file, but Quanterra files have the blocks from different channels intercalated, or “multiplexed”. In other words, we may have one block from channel BHE, then one from BHZ, then one BHN, and so on.

Dealing with this peculiarity conflicts with the way of solving the problems of RDSEED files, because the program would wrongly interpret each intercalated block as another sequence, and would consider it as a different channel. In order to deal with both kinds of files, our routines treat them differently. Internally, the routines identify which kind of file it is (if intercalated or sequential) in the beginning, and treat each case differently.

Another issue about intercalated blocks is that they cannot be considered uniformly intercalated. Often you can find sequences like: BHE BHN BHE BHZ ... It means that, when want to actually read the waveform, we cannot take full advantage of direct access, unless we index all the blocks in memory. Since we only store the start and end point of each channel, by the time we decompress the waveform we have to go again through all blocks in between, check the headers and discard the ones that don't belong to the channel.

Orfeus

SEED files originated by Orfeus (*orfeus.knmi.nl*) have more than one blockette 30. This means that different channels can have different encoding formats. This is not a problem for our program since each Data Record also has the blockette 1000, so, by the time of the decompression, we figure out correctly the encoding format for each block. But our routines would fail if they didn't have the blockette 1000. This happens because our routines only store the encoding information once. So each blockette 30 would overwrite the information of the previous one.

IRIS

The files created there may also have different encoding formats for each channel and, as before, this is not a problem since every Data Record has the blockette 1000.

Besides that, these files present another peculiarity. The record length of the Data Records may differ from the length in the headers. The only such weird combination seen so far was 4096-byte blocks for headers and 512-byte blocks for Data Records. To deal with this, our routines do the following:

- open the file with block size of 256, and find out block size of headers;
- close the file, and reopen it with the block size read from file;
- run through the headers;
- read the first data record and find out its size;
- close the file and reopen it again, on the new size, hoping that all the data records will have the same length.

So this will fail if different channels have different block sizes.

It's worth noting that IRIS files use the 8th byte of data records as a continuation code, although their manual of the SEED format [1] says it should always have a space character (ASCII 20).

Nannometrics

The SEED files generated by Nannometrics conversion software use Steim 1 format. They don't have blockette 1000, but it is not a problem as long as they only use one compression format.

The problem about these files is that the compression is written in a way that diverges from the way RDSEED writes Steim 1 compression. At least that was the case with the sample files we tested.

This problem is due to the way that 2 or 4 sample differences are assembled in one word, as it is supposed to be done on Steim 1 compression. There are two ways to assemble the differences. On big-endian computers, both ways generate the same result, while on PC they generate different files. RDSEED uses one way, and Nannometrics uses the other, so running both on PC will generate different files. Note that the file we tested was generated on PC (whether Windows or Linux).

The first way to assemble the differences is using shifting. So, suppose you want to compress the 2-byte differences $0x0001$ and $0xFFAA$. They should be put together in a 4-byte variable called `word`. The algorithm is:

- put the first difference in `word`;
- shift `word` 16 bits to the left (towards the most significant);
- add the 16 least significant bits to `word`;

Following this we will have:

in big endian	in little-endian
<code>00 01 FF AA</code>	<code>AA FF 01 00</code>

The other way is to access bytes directly. The algorithm is:

- first 2 bytes of `word` receive first difference;
- last 2 bytes receive second difference.

In this case, `word` will look like:

in big endian	in little endian
<code>00 01 FF AA</code>	<code>01 00 AA FF</code>

As it can be seen in the figures above, both ways of encoding give the same result, when running on SUN machines, but they give different results when running on PC. RDSEED writes files in the first way, but it reads files in the second way. For this reason, RDSEED can read Nannometrics files. (It would probably not be able to read files generated by itself on Linux, however this was not tested). In order to read both kinds of files, our program tries to decompress in the first way. If the Integrity constant does not check, it tries the second way.

GRC (unknown source)

This file was a Mini-SEED without blockette 1000. There was no hidden information in blank areas to tell the block size or encoding format, but they happen to be 4096 and Steim 1, respectively. This file was the only one that did not start with sequence number 000001. Besides that, it had some weird characteristic on the encoding. The very first difference in the block held some value whose meaning could not be determined. According to the manual:

“The very first difference (d_0) of the data actually represents the difference between the last sample of the previous record and x_0 , the first sample of the record.”[1].

In other words, we can use d_0 to generate x_0 , but x_0 is always given in the block. This could be used in order to do some decompression checking: we could verify that the information in one block is correctly following the previous one. But since this file fails on this check, it cannot be used. So, in our routines we always discard the first difference.

Other files

We also tried files from Seedlink, Kinematics and Guralp SCREAM. They are all Mini-SEED well behaved files.

Writing Mini-SEED

Functions for writing Mini-SEED were developed in Fortran as well as in C. The Fortran routines are in the same library as the functions for reading SEED files. They have the following characteristics:

- the data can be written in 32-bit integers, or in Steim 1;
- the block size can be set to up to 4096 bytes.
- they write one channel at a time, using as many blocks as necessary;
- they write the blocks to a file.

The routines in C, have the following characteristics:

- write data in 32-bit integers or Steim 1;
- the block size can be set to any size that is valid according to the standard;
- they write one channel at a time, using as many blocks as necessary;
- they write the blocks to the memory – it is up to the calling program to write the blocks to the file, or to send them on a network connection.

External functions in the Fortran library:

- *seed_open_file (filenumber, filename, length)*: opens a file with the given *filename*, in the desired block *length*, and identifying it with *filenumber*. This function is supposed to be called by the external programs in order to open the file, checking the correct block length. Since this function sets the block length variable (which was made inaccessible for calling programs on purpose), the reading function will fail if this one is not called in advance.
- *seed_write_chn(file, buf, n_blocks)*: writes data from *buf* to the *file*. The parameter *n_blocks* is an input/output, which tells the number of the next block to be written in the file (same as direct access position). Before the first call to this function, it must be set to 1, so it will write starting from the first block of the file. The function then updates it, returning the number of the number of the next block to write, in case the function is called again.

Besides those parameters, the function uses the data in the common block */write_mseed_param/*. This common must be set with the channel information before the function is called.

External functions in the C library:

- *write_chn(info, records)*: writes miniseed blocks in the memory area pointed by *records*, using the information in the structure *info*. The structure *info* contains all the data that the function needs: block size, encoding format (Steim 1, etc.), channel and station names, start time, sample rate, and the waveform data. The memory area for *records* is allocated inside the function, and should be freed by the calling program.

Writing Steim 1

Compressing the Data in Steim 1 is a lot more complicated than reading it. The algorithm we adopted is the same that is used in the RDSEED code. In order to understand the algorithm used for writing, it is necessary to be familiar with the compression scheme, which is not explained here. Refer to the Appendix B in [1].

The algorithm is described as a DFA, where we have a set of different states, and depending on the new entry, we move from one state to another. Each entry, in this case, corresponds to a difference between two samples. What matters for the algorithm is whether the difference needs 1, 2 or 4 bytes to be represented.

We always have to verify a few differences – at most 4 – before deciding how to compress them. Thus, we hold them in an array of 4 positions, named *buffer*. In each state of the DFA, the *buffer* has a different configuration.

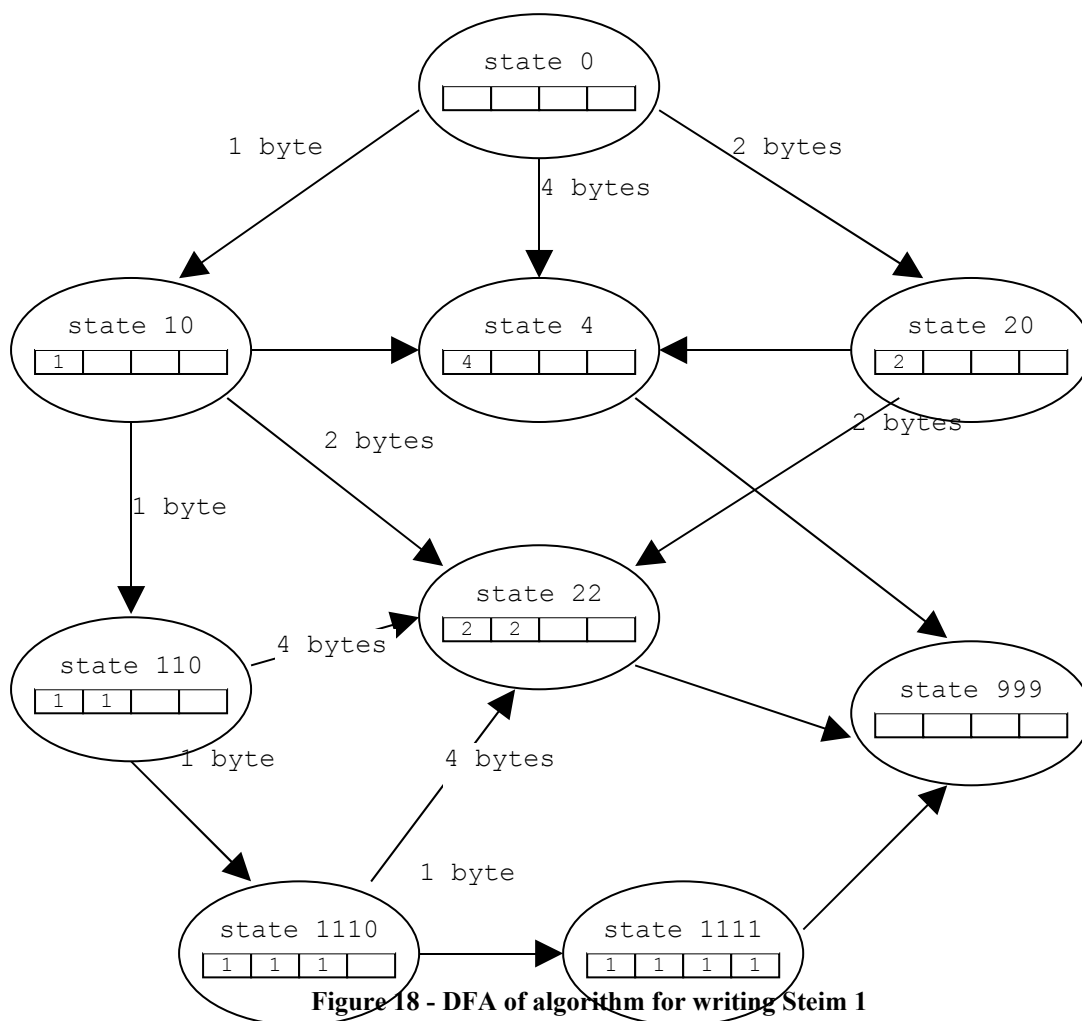


Figure 18 - DFA of algorithm for writing Steim 1

The start state is State 0. From there, depending on the size of the difference, we go to a different state. The numbers of the states try to be self-explanatory, informing what we have in the *buffer*. For instance, state 10, has one 1-byte difference in the first position, and the rest is empty. The state 110 has 2 1-byte differences, and so on.

States 4, 22, and 1111 are final. It means a decision has been made, and a new word will be written. In state 4, the decision was to write one difference of 4 bytes; in state 22, to write two differences of 2 bytes; At last, on state 1111, 4 differences of 1 byte. Note that we do not always decide about all the differences that we checked. For example, when we are in state 1110, and the 4th difference cannot be written in 1 byte, we actually discard 2 differences, and write only the first two as 2-byte integers. The two discarded differences will be reconsidered on the next loop in the DFA.

After writing a new word, whatever the final state, we need to check if we reached the end of a frame. If so, we have to finish the frame, and start a new one. Since this checking is common to all final states, it was put in another state, which was numbered 999.

The algorithm loops from state 0 to 999 until all the samples are used, or we fill the whole block. In both cases, the compression function ends. If there are still more samples, we start a new block and the compression function starts again.

Diagram of the writing functions

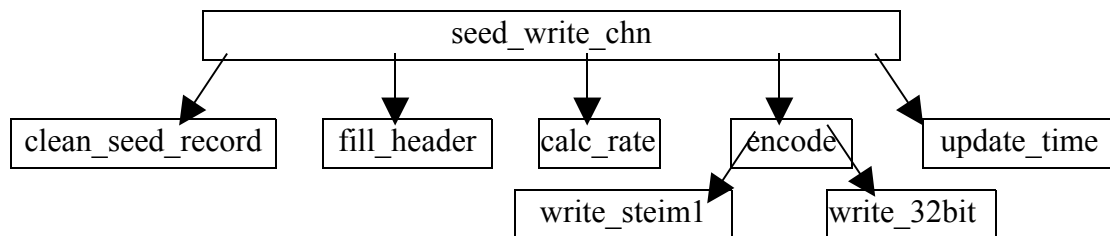


Figure 19 - Diagram of function *seed_write_chn*

All the other functions are only used internally, and should not be called from the main program. Here is a description of what they do:

clean_seed_record: fills the whole block with zeros, in order to allow the space to be reused for the next block, and guarantee that no junk will be left in the block before it is written to the file.

fill_header: fills the fields of the main header and of the blockette 1000 in the block.

calc_rate: calculates the integer arguments factor and multiplier, based on the sample rate (a real number) of the channel. If it defines that it cannot represent the real sample rate as integers with the correct precision, it writes a blockette 100 in the current Data Record (this blockette is used to store the sample rate as a real number).

encode: chooses the correct encoding function, according to the format specified by the user in the common block */write_mseed_param/*.

update_time: updates the fields that compose the start time of the block, adjusting it for the next block.

write_32bit, *write_steim1*: write the data to the block as 32-bit integers, or Steim 1 compressed.

Conclusion

During the internship, we managed to develop a library in standard Fortran 77 that offers functions to read SEED and MiniSEED files, and also to write Mini-SEED. The library is stand alone, and can be included in any software that wants to work with those formats, in Unix, Linux and Windows.

A new SEISAN beta version, that used the library for reading SEED and Mini-SEED, was released and sent to some institutions around the world. Most of them reported success on the use of the new version on reading SEED files. Only two error cases were reported, and fixed.

Several new features were included in the library after the last released SEISAN beta version, including reading sections of channels, and writing MiniSEED. These features have only been tested in the institute, and it is most likely that some bugs will be reported when the next SEISAN beta version is released.

Besides the Fortran routines, we created a C library to write MiniSEED. The software has been tested internally, and hopefully it has no bugs. The library can also be included in any software, in the three platforms. This library was designed in order to be integrated in Unix and Windows versions of SEISLOG, but it has not been integrated in the software so far.

Last but not least, this manual is probably the most important result in this internship, and we hope it will ease the way of those who try to understand SEED and MiniSEED and those who need to modify the developed routines, either to correct bugs or to add new features to it.

References

IRIS Consortium. Standard for the Exchange of Earthquake Data – Reference Manual, 2nd Edition, February 1993.

Acknowledgement

The author thanks the University of Bergen for the IAESTE scholarship, which made this work possible.