GFZ

Helmholtz-Zentrum
POTSDAM

HELMHOLTZ-ZENTRUM POTSDAM
DEUTSCHES
GEOFORSCHUNGSZENTRUM

Oliver Ritter, Reinhard Klose, Ute Weckmann

EMERALD

Data Format for Magnetotelluric Data

Scientific Technical Report STR15/08 - Data

www.gfz-potsdam.de

HELMHOLTZ
GEMEINSCHAFT

Recommended citation:

Ritter, O., Klose, R., Weckmann, U. (2015), EMERALD Data Format for Magnetotelluric Data, *Scientific Technical Report 15/08 - Data, GFZ German Research Centre for Geosciences.*
DOI: http://doi.org/10.2312/GFZ.b103-15082

Imprint

Helmholtz Centre Potsdam
GFZ German Research Centre
for Geosciences

Telegrafenberg
D-14473 Potsdam

Published in Potsdam, Germany
September 2015

ISSN 2190-7110

DOI: http://dx.doi.org/10.2312/GFZ.b103-15082
URN: urn:nbn:de:kobv:b103-15082

This work is published in the GFZ series
Scientific Technical Report (STR)
and electronically available at GFZ website
www.gfz-potsdam.de

Oliver Ritter, Reinhard Klose, Ute Weckmann

EMERALD

Data Format for Magnetotelluric Data

Scientific Technical Report STR15/08 - Data

EMERALD

DATA FORMAT FOR MAGNETOTELLURIC DATA

Oliver Ritter, Reinhard Klose, Ute Weckmann

GFZ German Research Centre for Geosciences,

Potsdam, Germany

Version 1.0

September, 2015

Contents

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

1 Preamble

The original version of this document was drafted by OR as part of his PhD project at the University of Edinburgh between 1991 and 1996. This updated text is based on version 0.2 from November 7, 1997.

2 EMERALD

The acronym EMERALD was supposed to stand for ElectroMagnetic Equipment, Raw data And Locations Database. What survived over the years was the *EMERALD* processing, a set of computer programs to analyse MT time series data, and the *EMERALD* file format for storing MT data. This document describes the *EMERALD* file format and how to use it with the C++, C and FORTRAN programming languages. Interface functions also exist for Matlab and Powershell but they are not described here.

EMERALD data files typically come in pairs of two files with the same name but differing file name extensions, sometimes called *RAW* and *XTR* files. *XTR* (extract) files are plain ASCII files, which can be read and modified with text editors. *RAW* files or more generally, EMERALD -type data files are in most cases binary and used to store all kind of magnetotelluric (MT) data such as time series, cross- and auto spectra and calibration data. The EMERALD -type data files store any number of channels of data in matrix form, but do not contain any description of the data. This information is stored in the according XTR file.

In 2015 the original XTR files were replaced by a modernized version based on the Extensible Markup Language (XML). The new files have the extension *.XTRX*.

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

3 EMERALD Extract Files (XTR/XTRX)

3.1 The XTRX File Structure

XTRX files are ASCII files which are internally structured using the Extensible Markup Language (XML).

The characters making up an XML document are divided into markup and content. Strings that constitute markup usually begin with the character '<' and end with a '>'. Strings of characters that are not markup are content. A Tag is a markup construct with start-tags `<section>`, end-tags `</section>` or empty-element tags `<line-break />`.

A logical document component begins with a start-tag and ends with a matching end-tag. The characters between the start- and end-tags, are the element's content, and may contain markup, including other elements, which are called child elements. An example of an element is `<Greeting>Hello, world.</Greeting>`.

Attributes are markup constructs consisting of name/value pairs that exists within a start-tag. In the example `<step number="3">Connect A to B.</step>`, the element step has the attribute "number" with the value "3". Everything between `<!-- this -->` is a comment.

For more information on XML please refer to: https://en.wikipedia.org/wiki/XML and references therein.

The principle building blocks of EMERALD XTRX files contain a number of predefined tags which are mostly useful to describe magnetotelluric data (acquisition). Please refer to the comments (in blue) for an explanation of the tags:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- XML declaration -->

<EmeraldData>
<!-- always the root Tag in XTRX files -->

  <XtrxVersion>1.0</XtrxVersion>
  <!-- XTRX version counter -->

  <ProjectName>PROJ</ProjectName>
  <!-- content: Project name (type string) -->

  <SampleRate Unit="Hz">500.0</SampleRate>
  <!-- content: Sampling rate of time series data (type: double), attribute: Unit:
  "Hz | s" (type string) -->

  <Lowpass Unit="Hz">200.0</Lowpass>
  <!-- content: low-pass filter cut-off frequency (type: double), attribute: Unit:
  "Hz | s" (type string) -->

  <Highpass Unit="s">0.0</Highpass>
  <!-- content: high-pass filter cut-off frequency (type: double). If value is
  set to 0, HP filter is switched off. Attribute: Unit: "Hz | s" (type string) -->

  <StartTime Unit="UTC">2014-01-01_12:10:00.000000000</StartTime>
  <!-- content: Date and time of the first data item in the associated data file
  (type: string), attribute: Unit: "UTC" (type string) -->

  <StopTime Unit="UTC">2014-01-01_12:20:00.936000000</StopTime>
  <!-- content: Date and time of the last data item in the associated data file
  (type: string), attribute: Unit: "UTC" (type string) -->
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```xml
<DataEncoding>4Byte,Float,Binary,LittleEndian</DataEncoding>
<!-- content: Auxiliary Information on the data file format (type: string)-->

<DataFileName>
   SSSS_LP000500Hz_HP000000s_D20150125_T121025.RAW
</DataFileName>
<!-- content: name of associated EMERALD data file (type string); filename
convention: SSSS: Site Number, LP: Lowpass, HP: Highpass, D: Date, T: Time -->
<!-- Alternative file name: SSSS-RRRR_LP…: Two Sites, SSSS: Site number of local
site, RRRR: Site number of Remote Reference site -->
<!-- Alternative file name: MCCC_LP…: multi-site- CCC indicates number of sites -->

<Site Index="1">
<!-- content: Site counter Tag, attribute: index (type string) -->

  <SiteNumber>0001</SiteNumber>
  <!-- content: 4 digit site number (type integer) -->

  <Operator>OR</Operator>
  <!-- content: initials of operator (type string) -->

  <Comment>test</Comment>
  <!-- content: a comment relevant for that site(type string) -->

  <Latitude Unit="deg">0.0</Latitude>
  <!-- content: latitude of site (type double); convention: northern hemisphere:
  positive, southern Hemisphere: negative. Attribute: Unit: "deg" (type string) -->

  <Longitude Unit="deg">0.0</Longitude>
  <!-- content: longitude of site (type double); convention: East of Greenwich:
  positive, West of Greenwich: negative. Attribute: Unit: "deg" (type string) -->

  <Altitude Unit="m">0.0</Altitude>
  <!-- content: Height above sea-level (type double), Attribute: Unit: "m"
  (type string)-->

  <Declination Unit="deg">0.0</Declination>
  <!-- content: declination (deviation of magnetic north from geographic north)
  (type double); convention: Positive when magnetic north is east of true north,
  and negative when it is to the west. Attribute: Unit: "deg" (type string) -->

  <Channel IndexInFile="1">
  <!-- content: Channel counter Tag. Attribute: IndexInFile (type string);
  Index counter starting at 1, has to be unique in one file -->

    <Type>Bx</Type>
    <!-- content: Type of channel (type string); can be "Bx | By | Bz | Ex | Ey |
    Ez | I1 | I2 | I3 | Na (=not available"). Note: a channel of the same Type
    may occur only once per site. -->

    <DataUnit>Volt</DataUnit>
    <!-- content: Unit of channel (type string); e.g. "Volt | nT | mV/km " -->

    <Comment>optional_information</Comment>
    <!-- content: a comment relevant for that channel (type string) -->

    <Gain>1.0</Gain>
    <!-- content: any static gain factors that may need to be applied to the data
    (type double) -->

    <DCOffset>0.0</DCOffset>
    <!-- content: i.e. self potential between electrodes (type double)-->

    <VerticalOrientation Unit="deg">
      0.0
    </VerticalOrientation>
    <!-- content: Rotation of sensor with respect to surface (tilt) (type double).
    Attribute Unit "deg" (type string) -->
```

```
<HorizontalOrientation Unit="deg">
  0.0
</HorizontalOrientation>
<!-- content: Rotation of sensor with respect to magnetic north(type double).
Attribute Unit "deg" (type string) -->

<ModuleResponse Type="INFO">
  SP4_50.0_052XXXX--TYPE-OFF_HF-ID-000018.RSP
</ModuleResponse>
<!-- content: Name of File containing data to correct the frequency response of
a hardware module (type string). The module response files typically contain data
of the form: Frequency, Amplitude, Phase. Attribute Type "Info | Resp" (type
string). Several of such module responses are allowed. -->

<ModuleResponse Type="INFO">
  SPAM_SensorBox----TYPE-HF0002-ID-000024.RSP
</ModuleResponse>
<ModuleResponse Type="RESP">
  Metronix_Coil-----TYPE-007_HF-ID-000121.RSP
</ModuleResponse>

<DipoleLength Unit="m">0.0</DipoleLength>
<!-- content: Length of telluric lines (requires a channel Type beginning with
'E' (type float). Attribute Unit "m" (type string).-->

<ContactResistancePos Unit="kOhm">
  0.35
</ContactResistancePos>
<!-- content: Contact resistance of telluric electrode. Convention: if channel
Type contains 'x': Pos=North, Neg=South. Attribute: Unit="Ohm | kOhm | MOhm". ->

<ContactResistanceNeg Unit="kOhm">
  0.35
</ContactResistanceNeg>
<!-- content: Contact resistance of telluric electrode. Convention: if channel
Type contains 'y': Pos=East, Neg=West. Attribute: Unit="Ohm | kOhm | MOhm".  -->

<ContactResistanceGnd Unit="kOhm">
  0.35
</ContactResistanceGnd>
<!-- content: Contact resistance of ground electrode. Attribute: Unit="Ohm |
kOhm | MOhm".  -->

</Channel>

<ProcessingStep>
   <Date>YYYY-MM-DD_HH:MM</Date>
   <Program>emnotch(Version:1.49)</Program>
   <CommandLine>Notch50Hz</CommandLine>
</ProcessingStep>
<!-- content: Optional time domain processing steps -->

</Site>

</EmeraldData>
```

3.1.1 An Exemplary XTRX file

An example XTRX file for a 5 channel MT setup is shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<EmeraldData>
    <ProjectName>NULL</ProjectName>
    <SampleRate Unit="Hz">250.000000</SampleRate>
    <Lowpass Unit="Hz">100.000000</Lowpass>
    <Highpass Unit="s">0.000000</Highpass>
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
<StartTime Unit="UTC">2015-06-20_00-00-00.007200</StartTime>
<StopTime Unit="UTC">2015-06-20_22-16-57.403200</StopTime>
<DataEncoding>4Byte,Float,Binary,LittleEndian</DataEncoding>
<DataFilename>0996_LP00100Hz_HP00000s_R015_W171.RAW</DataFilename>
<Site Index="1">
    <SiteNumber>996</SiteNumber>
    <Operator>OR</Operator>
    <Comment>none</Comment>
    <Latitude Unit="deg">53.239047</Latitude>
    <Longitude Unit="deg">12.547704</Longitude>
    <Altitude Unit="deg">123.000000</Altitude>
    <Declination Unit="deg">0.000000</Declination>
    <Channel IndexInFile="0">
        <Type>Bx</Type>
        <DataUnit>NULL</DataUnit>
        <Gain>1.000000</Gain>
        <DCOffset>0.000000</DCOffset>
        <HorizontalOrientation Unit="deg">
            0.000000
        </HorizontalOrientation>
        <VerticalOrientation Unit="deg">
            0.000000
        </VerticalOrientation>
        <ModuleResponse Type="INFO">
            SP4_0.50_2XXXXXX--TYPE-OFF_500-ID-000000.RSP
        </ModuleResponse>
        <ModuleResponse Type="INFO">
            CASTLE_SensorBox--TYPE-V2------ID-000000.RSP
        </ModuleResponse>
        <ModuleResponse Type="RESP">
            Metronix_Coil-----TYPE-006_LF--ID-000133.RSP
        </ModuleResponse>
    </Channel>
    <Channel IndexInFile="1">
        <Type>By</Type>
        <DataUnit>NULL</DataUnit>
        <Gain>1.000000</Gain>
        <DCOffset>0.000000</DCOffset>
        <HorizontalOrientation Unit="deg">
            90.000000
        </HorizontalOrientation>
        <VerticalOrientation Unit="deg">
            0.000000
        </VerticalOrientation>
        <ModuleResponse Type="INFO">
            SP4_0.50_2XXXXXX--TYPE-OFF_500-ID-000000.RSP
        </ModuleResponse>
        <ModuleResponse Type="INFO">
            CASTLE_SensorBox--TYPE-V2------ID-000000.RSP
        </ModuleResponse>
        <ModuleResponse Type="RESP">
            Metronix_Coil-----TYPE-006_LF--ID-000135.RSP
         </ModuleResponse>
    </Channel>
    <Channel IndexInFile="2">
        <Type>Bz</Type>
        <DataUnit>NULL</DataUnit>
        <Gain>1.000000</Gain>
        <DCOffset>0.000000</DCOffset>
        <HorizontalOrientation Unit="deg">
            0.000000
        </HorizontalOrientation>
        <VerticalOrientation Unit="deg">
            -90.000000
        </VerticalOrientation>
```

```
        <ModuleResponse Type="INFO">
            SP4_0.50_2XXXXXX--TYPE-OFF_500-ID-000000.RSP
        </ModuleResponse>
        <ModuleResponse Type="INFO">
            CASTLE_SensorBox--TYPE-V2------ID-000000.RSP
        </ModuleResponse>
        <ModuleResponse Type="INFO">
            Metronix_Coil-----TYPE-010_LF--ID-000005.RSP
        </ModuleResponse>
    </Channel>
    <Channel IndexInFile="3">
        <Type>Ex</Type>
        <DataUnit>NULL</DataUnit>
        <Gain>1.000000</Gain>
        <DCOffset>0.000000</DCOffset>
        <HorizontalOrientation Unit="deg">
            0.000000
        </HorizontalOrientation>
        <VerticalOrientation Unit="deg">
            0.000000
        </VerticalOrientation>
        <DipoleLength Unit="m">58.700000</DipoleLength>
        <ContactResistancePos Unit="ohm">
            0.000000
        </ContactResistancePos>
        <ContactResistanceNeg Unit="ohm">
            0.000000
        </ContactResistanceNeg>
        <ContactResistanceGnd Unit="ohm">
            0.000000
        </ContactResistanceGnd>
        <ModuleResponse Type="INFO">
            SPAM_MkIV_55XXXXX-TYPE-6.25HF-ID-000052.RSP
        </ModuleResponse>
        <ModuleResponse Type="INFO">
            SPAM_SensorBox----TYPE-001_V2-ID-131000.RSP
        </ModuleResponse>
        <ModuleResponse Type="RESP">
            TelluricElectrode-TYPE-AgAgCl-ID-000000.RSP
        </ModuleResponse>
    </Channel>
    <Channel IndexInFile="4">
        <Type>Ey</Type>
        <DataUnit>NULL</DataUnit>
        <Gain>1.000000</Gain>
        <DCOffset>0.000000</DCOffset>
        <HorizontalOrientation Unit="deg">
            90.000000
        </HorizontalOrientation>
        <VerticalOrientation Unit="deg">
            0.000000
        </VerticalOrientation>
        <DipoleLength Unit="m">56.400000</DipoleLength>
        <ContactResistancePos Unit="ohm">
            0.000000
        </ContactResistancePos>
        <ContactResistanceNeg Unit="ohm">
            0.000000
        </ContactResistanceNeg>
        <ContactResistanceGnd Unit="ohm">
            0.000000
        </ContactResistanceGnd>
        <ModuleResponse Type="INFO">
            SPAM_MkIV_55XXXXX-TYPE-6.25HF-ID-000052.RSP
        </ModuleResponse>
```

```
            <ModuleResponse Type="INFO">
                SPAM_SensorBox----TYPE-001_V2-ID-131000.RSP
            </ModuleResponse>
            <ModuleResponse Type="RESP">
                TelluricElectrode-TYPE-AgAgCl-ID-000000.RSP
            </ModuleResponse>
        </Channel>
    </Site>
</EmeraldData>
```

3.2 The Application Program Interface for XTRX Files

3.2.1 Definition of the XTRX Interface in C++

Access to XML – and therefore XTRX - files is easily handled with modern programming languages such as C++, Matlab, C#, Powershell, etc. Below we describe an interface implemented in C++.

Please refer to section 3.1 for the general structure of the XTRX files and the meaning of the Tags / variables.

```cpp
// --------------------------------------------------------------------
// ---------------------- Emerald Xtrx Interface-----------------------
// --------------------------------------------------------------------

struct EmModule
{
  string resp;
  string mode;
};

struct EmChannel
{
  ChanType type;
  string dataUnit;
  string comment;
  double dipoleLength;
  double verticalOrientation;
  double horizontalOrientation;
  double gain;
  double dcOffset;
  double resPos;
  double resNeg;
  double resGnd;
  unsigned int indexInFile;
  unsigned int indexInSample;
  list<EmModule> modules;
};

struct EmSite
{
  string siteName() const;
  double latitude;
  double longitude;
  double altitude;
  double declination;
  unsigned int siteNumber;
  string operatorName;
  string comment;
  map<ChanType, EmChannel> channels;
  vector<EmLogEntry> processingSteps;
};
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```cpp
struct EmXtrxFile
{
  EmXtrxFile();
  void load();
  void save();
  void clear();
  string str() const;
  string fileNameFromContents() const;
  string dataFileFullPath() const;
  unsigned int chanCount() const;
  EmSite* site(unsigned int siteNumber);
  EmSite* site(const string& siteName);
  EmSite* site(const EmChannel& c);
  EmChannel* channel(unsigned int indexInFile);
  string projectName;
  string dataEncoding;
  string dataFileName;
  double startTime;
  double stopTime;
  double xtrxVersion;
  EmBand band;
  vector<EmSite> sites;
  string file;
};
```

3.2.2 Examplary Program Accessing XTRX files in C++

The example below reads information from an XTRX file to convert an EMERALD (.RAW) data file (see chapter 4) to ASCII. The program requires the declarations given in the previous section.

```cpp
// Example Program
int main( int argc, char *argv[] )
{
  const double version = 1.0;
  EmXtrxFile xtrx;
  try {
    std::cout << "*******************************************************************\n";
    std::cout << "*                           emxtoasc                              *\n";
    std::cout << "*                    Version: ";
    std::cout << setprecision(1) << fixed << version;
    std::cout << " - " << __DATE__ << "                      *\n";
    std::cout << "*                          GFZ Potsdam                            *\n";
    std::cout << "*******************************************************************\n";
    std::cout << "* emtoasc converts Emerald .raw binary data files to              *\n";
    std::cout << "* ASCII data files. Usage: emtoasc [xtrx filename]                *\n";
    std::cout << "*******************************************************************\n";
    std::cout << "\n";
    std::cout.flush();

// needs 2 parameters from the commandline. The first is the program name (emtoasc).
// The 2nd paramter is the name of the .xtrx file associated with the EMERALD data
// file to be converted.
    if (argc == 1) {
      throw EmException(EmError::Usage, "No XTRX file specified");
    } else
// expect the second parameter to be filename
// load() throws exception, if unable to load a .xtrx file
    if (argc == 2) {
       xtrx.load(std::string(argv[1]));
    } else {
       throw EmException(EmError::Usage, "Too many command line arguments");
    }

// The purpose of this output is to show how to access the XTR data fields
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
// (just a few examples here)
    std::cout << "XTR filename: " << xtrx.file << "\n";
    std::cout << "Derived filename: " << xtrx.fileNameFromContents() << "\n";
    std::cout << "Data filename: " << xtrx.dataFileName << "\n";
    std::cout << "Project name: " << xtrx.projectName << "\n";
// rates in xtr files underly the conversion that negative numbers have unit Hz
// while positive numbers have unit second. Use the conversion routines to handle.
    std::cout << "SampleFrequency: " << xtrx.band.sampleFreq << " Hz\n";
    std::cout << "StartTime: " << emTimeToStr(xtrx.startTime, true) << "\n";
    std::cout << "StopTime: " << emTimeToStr(xtrx.stopTime, true) << "\n";
    for (unsigned int i = 0; i < xtrx.chanCount(); ++i) {
      const EmChannel* c = xtrx.channel(i);
      if (c != nullptr) {
        const EmSite* s = xtrx.site(*c);
        std::cout << (i+1) << ". Chan Site:" << s->siteName() << "\n";
        std::cout << (i+1) << ". Chan Name:" << nameOf(c->type) << "\n";
        std::cout << (i+1) << ". Chan Comment:" << c->comment << "\n";
      }
    }
    std::cout << "\n\n";
// To change a field in the .XTRX, just change the variable and save the file
    xtrx.projectName = "PROJ";
    EmChannel* c = xtrx.channel(2);
    if (c != nullptr) {
      c->type = ChanType::Ey;
    }
    xtrx.save();

// Now load the .raw file with the binary data
    EmRawFile raw;
    raw.open(emGetFileName(xtrx.file, true, false) + ".RAW");
    std::cout << "RAW filename: " << raw.file() << "\n";
    std::cout << "EventCount: " << raw.generalHeader().totalEvents << "\n";
    std::cout << "RecordCount: " << raw.generalHeader().totalRecs << "\n";
    std::cout << "RecordLength: " << raw.generalHeader().recLen << " Bytes\n";
    std::cout << "StartTime: " << emTimeToStr(raw.eventHeader().startTime(), true)<<"\n";
    std::cout << "StopTime: " << emTimeToStr(raw.eventHeader().stopTime(), true) << "\n";
    std::cout.flush();

// start reading and
// Open the file to store ASCII output
    const std::string asciiFileName = emGetFileName(xtrx.file, true, false) + ".ASC";
    ofstream asciiStream(asciiFileName);
    asciiStream.exceptions(ofstream::badbit);
    bool ok = asciiStream.is_open();

    vector<float> sample;
// Write only the first 100 Samples
    unsigned int nSamples = 100;
    while (ok && raw.readNextRecord(sample)) {
      for (const auto& f : sample) {
        asciiStream << f << " ";
      }
      asciiStream << "\n";
      ok = asciiStream.good();
      ok &= (--nSamples >= 0);
    }

    asciiStream.close();
    raw.close();
    std::cout << "\n Done.\n\n";
    std::cout.flush();

    std::cout << "\n\n emtoasc successfully finished\n";
    std::cout << "******************************************************************\n";
```

```
    std::cout.flush();
    return 0;
  } catch (EmException& e) {
    std::cout << "\n\n Error: emtoasc failed\n";
    std::cout << " " << e.what() << "\n";
    std::cout << "*******************************************************************\n";
    std::cout.flush();
    return e.errorCode();
  } catch (...) {
    std::cout << "\n\n Error: emtoasc failed\n";
    std::cout << " UnknownError\n";
    std::cout << "*******************************************************************\n";
    std::cout.flush();
    return static_cast<int>(EmError::Unknown);
  }
}
```

The output produced by the above program should look like this:

```
**********************************************************************
*                            emxtoasc                                *
*                    Version: 1.0 - Jul 24 2015                      *
*                              GFZ Potsdam                           *
**********************************************************************
* emtoasc is a program to convert Emerald .raw binary data files to  *
* ASCII data files. Usage: emtoasc [xtrx filename]                   *
**********************************************************************

XTR filename: C:\TEST\0001_LP02500Hz_HP00000s_D2015205_T124000.XTRX
Derived filename: 0001_LP02500Hz_HP00000s_D2015205_T124000.XTRX
Data filename: 0001_LP02500Hz_HP00000s_D2015205_T124000.RAW
Project name: PROJ
SampleFrequency: 6250.0 Hz
StartTime: 2015-07-24_12-40-00.000000
StopTime: 2015-07-24_12-41-59.999840
1. Chan Site:0001
1. Chan Name:Bx
1. Chan Comment:
2. Chan Site:0001
2. Chan Name:By
2. Chan Comment:
3. Chan Site:0001
3. Chan Name:Ex
3. Chan Comment:
4. Chan Site:0001
4. Chan Name:Ey
4. Chan Comment:


RAW filename: C:\TEST\0001_LP02500Hz_HP00000s_D2015205_T124000.RAW
EventCount: 1
RecordCount: 750012
RecordLength: 16 Bytes
StartTime: 2015-07-24_12-40-00.000000
StopTime: 2015-07-24_12-41-59.999840

Done.

 emtoasc successfully finished
**********************************************************************
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

3.3 The XTR File Structure

XTR files are plain ASCII files, which are internally divided into logical units, called *sections*. A section contains one or more *keywords*, followed by the data items for that keyword. Within a section, there may be multiple definitions of a keyword. If the keyword is multiple defined, the first data item is the *key-index*, to identify the keyword.

Note XTR files are now superseded by the XTRX interface (see above).

Sections

Section names are unique and per extract file any section can be defined only once. However, they can be defined anywhere in the extract file, the sequence is irrelevant. Section names are made of uppercase letters [A..Z], enclosed in square brackets, e.g. [EMERALD]. The length of the section names should not exceed 10 characters. If a section contains calibration data, the name consists of a 7 digit integer number. The number is calculated using the following formula:

```
section = 1000000 + (channel_number * 1000) + module_number            (1)
```

In 2011 the instrument response descriptions were updated, which is also reflected in the section identifiers:

```
section = 2000000 + (channel_number * 1000) + module_number            (2)
```

The `channel_number` and `module_number` variables are derived in the `[CHAN]` section (see chapter 3.3.1).

A number of section names are predefined and described in more detail further below. The `[TITLE]` section, for example, typically holds information about the program that created or modified the file. To ensure compatibility with older versions of FORTRAN, each section, including the first one, is preceded by a blank line. The first character in each line must be left blank. Therefore, the section names start on the second position of each line.

Keywords

Keywords consist of uppercase letters `[A..Z]`, followed by an equal sign and are enclosed in quotes, e.g. `'COMMENT='`. The length of the keywords should not exceed 10 characters.

Again, the first character in each line must be left blank. Keywords start on the second position of a line and only one keyword is permitted per line. Keywords are related to their sections and must not appear elsewhere in the files.

Key-indices

Key-indices are used for sections where keywords appear more than once (e.g. the `'CHAN'` keyword in the `[DATA]` section in chapter 3.2. If multiple keywords are used, the sections always contain the keyword `'ITEMS'`. `ITEMS` specifies how often an indexed keyword appears in the section. If the section contains more than one keyword, other keywords are not necessarily indexed.

3.3.1 Definition of Sections and Keywords in XTR Files

The following list contains the definitions of sections and keywords.

For this documentation, indexed keywords are marked by the → character and always the first data item after the keyword is the key-index (_index). For the presentation here, the '*' character is used as a separator between any two data items. In the extract files data items are separated by blanks and string-type data items are enclosed in quotes.

Values can be integer, floats, or strings. Indices are always integers. Comments are marked in blue, they are not part of the XTR files.

```
[CHANNAME]
'ITEMS=' _number_of_channel_names(int)
→ 'NAME=' _index(int) * chan_name(string) * chan_units(string)
→ 'DEFAULT=' _index(int) * chan_distance(float) * chan_rotation(float) *
             chan_tilt(float) * chan_scaling(float)
→ 'COMMENT=' _index(int) * chan_descrpt(string)
Note: The 'DEFAULT=' Keyword is not used and obsolete.

[CONFIG]
'NAME=' conf_channels(int) * conf_field[0](int) * conf_field[1](int)
'METHOD=' conf_method(string) * conf_fieldset(string)
'COMMENT=' conf_comment(string)
Note: The [CONFIG] section with all keyword is not used and obsolete.

[DATA]
'ITEMS=' _number_of_channels(int)
→ 'CHAN=' _index(int) * _[SITE]_index(int) * _[CHANNAME]_index(int) *
          cfgc_distance(float) * cfgc_rotation(float) * cfgc_tilt(float) *
          cfgc_scaling(float) * cfgc_modules(int)
Note: cfgc_distance: E-channel: dipole length or B-channel: sensor number
Note: cfgc_rotation: =0 for north, 90 for east
Note: cfgc_tilt: 0 for horizontal, -90 for vertical
Note: cfgc_scaling: static gains, factor to convert data into volts.


→ 'CHANAUX=' _index(int) * caux_value1(float) * caux_value2(float)*
             caux_value3(float) * caux_value4(float)
Note: value 2-4: used to store contact resistances.

[FILE]
'NAME=' file_name(string) * file_run(int) * recorded_events(int) *
        sampling_rate(float)
Note: recorded events is usually not used

'DATE=' start_timedate(int) * start_microseconds(int) * stop_timedate(int) *
        stop_microseconds(int)
Note: start/stop as UTC time (seconds since January 1970)

'FREQBAND=' frequency_band(string) * lowpass(float) * highpass(float) *
            description(string)
[PROJECT]
'NAME=' proj_name(string) * proj_owner(string)
'COUNTRY=' proj_country(string) * proj_state(string)
'SURVEY=' proj_survey(string) * proj_contractor(string)
'DATE=' proj_start(int) * proj_stop(int) * proj_tmzone(int)
'COORDS=' proj_lat(string) * proj_long(string) * proj_elev(float)
'COMMENT='proj_descrpt(string)
Note: in most cases only the project name is given (known) when the data files are created.

[SITE]
'ITEMS=' _number_of_sites(int)
→ 'NAME=' _index(int) * site_name(string) * site_number(int) *
          site_field[0](int) * site_field[1](int)
→ 'ALIAS=' _index(int) * site_alias(string) * site_comment(string)
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
→ 'COORDS=' _index(int) * site_lat(string) * site_long(string) *
            Site_elev(float)
Note: Alias keyword is not used and obsolete
Note: 'DECLINATION' keyword was introduced but is not supported by C and FORTRAN interface
Note: site_fields[0,1] are not used and obsolete.
```

```
[STATUS]
'STATUS=' XTR_file_status(string)
```

```
[TITLE]
'VERSION=' program_name(string) * program_version(string)
'DATE=' current_date(string)
'AUTHOR=' author(string)
'COMMENT=' comment(string)
```

```
[1/2nnnnnn] (0 ≤ n ≤ 9, see chapter 0)
'ITEMS=' _number_of_CALDATA_items(int)
'MODULE=' module_name(string) * module_number(int) *
          module_description(string)
'CATEGORY=' category_name(string) * category_number(int) *
            category_description(string)
'NAME=' clbr_name(string) * clbr_date(int) * clbr_user(string)
'COMMENT=' calibration_description(string)
→ 'CALDATA=' _index(int) * frequency(float) * amplitude(float) *
             phase(float)
Note: NAME keyword is not used and obsolete
Note: CALDATA keyword is not used and obsolete
```

3.4 The Application Program Interface for XTR Files

An API to access XTR files is provided in C++, FORTRAN, and C. The FORTRAN and C interfaces are very similar in their functionality (see Table 1) and are therefore described together.

function name	FORTRAN file name	C file name	short description
XTRfile	xtrfilef.for	xtrfilec.c	to open and close a new or existing file
XTRdata	xtrdataf.for	xtrdatac.c	to read, search, or write extract files.

Table 1: The EMERALD extract file API

Note, the XTR interface has been superseded by the XML based XTRX files (see section 3.1). It is highly recommended to use the library functions only to read existing XTR files. Any new projects should be based on the XTRX interface.

3.4.1 Definition of the XTR Interface in C++

```
// -----------------------------------------------------------------
// ------------------------- Emerald Xtr File ----------------------
// -----------------------------------------------------------------
class EmXtrFile
{
public:
  EmXtrFile();
  void load();
  void save();
  string str() const;
  EmXtrxFile toXtrx() const;
  void fromXtrx(const EmXtrxFile& xtrx);
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
  string fileName() const;
  string fileNameFromContents(int windowNumber = -1) const;

  class ChanSec {
  public:
  struct NameKey {
    string str() const;
    unsigned int index;
    string name;
    string unit;
  };
  struct CommentKey {
    string  str() const;
    unsigned int index;
    string text;
  };
  string str() const;
  unsigned int nItems() const;
  bool isNameKey() const;
  bool isCommentKey() const;
  NameKey* nameKey(unsigned int item);
  CommentKey* commentKey(unsigned int item);
  void resize(unsigned int nItems);
  void setNameKey(bool isOn);
  void setCommentKey(bool isOn);

private:
  bool   _isNameKey;
  bool   _isCommentKey;
  vector<NameKey> _nameKey;
  vector<CommentKey> _commentKey;
  unsigned int _nItems;
} chanSec;

class TitleSec {
public:
  struct VersKey {
    string str() const;
    string name;
    double vers;
  };
  struct DateKey {
    string str() const;
    string text;
  };
  struct AuthorKey {
    string str() const;
    string name;
  };
  struct CommentKey {
    string str() const;
    string text;
  };
  string   str() const;
  bool isCommentKey() const;
  bool isVersKey() const;
  bool isDateKey() const;
  bool isAuthorKey() const;
  CommentKey* commentKey();
  AuthorKey* authorKey();
  VersKey* versKey();
  DateKey* dateKey();
  void setCommentKey(bool isOn);
  void setVersKey(bool isOn);
  void setDateKey(bool isOn);
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
  void SetAuthorKey(bool isOn);
private:
  bool _isVersKey;
  bool _isDateKey;
  bool _isAuthorKey;
  bool _isCommentKey;
  VersKey     _versKey;
  DateKey     _dateKey;
  AuthorKey   _authorKey;
  CommentKey _commentKey;
} titleSec;

class SiteSec {
public:
  struct NameKey {
  public:
    string str() const;
    string name() const;
    unsigned int index;
    unsigned int number;
    int          f0;
    int          f1;
  };
  struct CoordsKey {
    string  str() const;
    unsigned int index;
    string lat;
    string lon;
    double alt;
  };
  struct DeclinationKey {
    string str() const;
    unsigned int index;
    double declination;
  };
  string str() const;
  bool isCoordsKey() const;
  bool isNameKey() const;
  bool isDeclinationKey() const;
  unsigned int nItems() const;
  NameKey* nameKey(unsigned int item);
  CoordsKey* coordsKey(unsigned int item);
  DeclinationKey* declinationKey(unsigned int item);
  void setCoordsKey(bool isOn);
  void setNameKey(bool isOn);
  void setDeclinationKey(bool isOn);
  void resize(unsigned int nItems);
private:
  bool _isNameKey;
  bool _isCoordsKey;
  bool _isDeclinationKey;
  vector<NameKey> _nameKey;
  vector<CoordsKey> _coordsKey;
  vector<DeclinationKey> _declinationKey;
  unsigned int _nItems;
} siteSec;

struct ProjSec {
public:
  struct NameKey {
    string str() const;
    string name;
    string owner;
  };
  struct CommentKey {
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
    string str() const;
    string text;
  };
  string str() const;
  bool isNameKey() const;
  bool isCommentKey() const;
  void setNameKey(bool isOn);
  void setCommentKey(bool isOn);
  NameKey* nameKey();
  CommentKey* commentKey();
private:
  bool _isNameKey;
  bool _isCommentKey;
  NameKey _nameKey;
  CommentKey _commentKey;
} projSec;

class FileSec {
public:
  struct NameKey {
    string str() const;
    string name;
    unsigned int run;
    unsigned int totalEvents;
    double sampleFreq;
  };
  struct DateKey {
    string str() const;
    time_t start;
    unsigned int startUs;
    time_t stop;
    unsigned int stopUs;
  };
  struct CorrKey {
    string str() const;
    double subUsCorr;
  };
struct BandKey {
    string str() const;
    string id;
    double lpFreq;
    double hpFreq;
    string text;
  };
  string str() const;
  bool isNameKey() const;
  bool isDateKey() const;
  bool isBandKey() const;
  bool isCorrKey() const;
  void setNameKey(bool isOn);
  void setDateKey(bool isOn);
  void setBandKey(bool isOn);
  void setCorrKey(bool isOn);
  NameKey* nameKey();
  DateKey* dateKey();
  BandKey* bandKey();
  CorrKey* corrKey();
private:
  bool _isNameKey;
  bool _isDateKey;
  bool _isBandKey;
  bool _isCorrKey;
  NameKey _nameKey;
  DateKey _dateKey;
  BandKey _bandKey;
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
  CorrKey _corrKey;
} fileSec;

class StatusSec {
public:
  struct StatusKey {
    string str() const;
    string text;
  };
  string str() const;
  bool isStatusKey() const;
  void setStatusKey(bool isOn);
  StatusKey* statusKey();
private:
  bool _isStatusKey;
  StatusKey _statusKey;
  } statusSec;

class DataSec {
public:
  struct ChanKey {
  public:
    string  str() const;
    unsigned int index;
    unsigned int site;
    unsigned int chan;
    double dist;
    double rot;
    double tilt;
    double scale;
    unsigned int totalModules;
  };
  struct ChanAuxKey {
    string  str() const;
    unsigned int index;
    double dcOffset;
    double eResG;
    double eResNG;
    double eResPG;
  };
  struct CommentKey {
    string str() const;
    unsigned int index;
    string text;
  };
  struct Module {
    string str() const;
    string file;
    string mode;
    unsigned int id;
  };
  string str() const;
  unsigned int nItems() const;
  bool isChanKey() const;
  bool isChanAuxKey() const;
  bool isCommentKey() const;
  void setChanKey(bool isOn);
  void setChanAuxKey(bool isOn);
  void setCommentKey(bool isOn);
  void resize(unsigned int nItems);
  ChanKey* chanKey(unsigned int item);
  ChanAuxKey* chanAuxKey(unsigned int item);
  CommentKey* commentKey(unsigned int item);
  vector<Module>* modules(unsigned int item);
private:
```

```
  bool _isChanKey;
  bool _isChanAuxKey;
  bool _isCommentKey;
  vector<ChanKey> _chanKey;
  vector<ChanAuxKey> _chanAuxKey;
  vector<CommentKey> _commentKey;
  vector<vector<Module>> _modules;
  unsigned int _nItems;
  } dataSec;
};
```

3.4.2 Definition of the XTR interface in FORTRAN and C

3.4.2.1 *Function XTRfile*

When a file exists, it can only be opened for reading. New data cannot be appended to existing files and the attempt to open an existing file in `CREAT`-mode will result in an error. Only one file can be opened at any time.

FORTRAN

Syntax	
integer	function XTRfile(fn, mode)
character*(*)	fn
integer	mode
Input Variables	
fn	file name
mode	XTR_CREAT: create new file XTR_EXIST: open existing file XTR_CLOSE: close file
Output Variables	
Return Value ERR OPEN NEW FILE	 cannot open file
ERR OPEN OLD FILE	file does not exist
ERR CLOSE FILE	file cannot be closed
ERR NOERROR	no error
Global Variables	
character*256	XTRfile
integer	XTRunit

In FORTRAN, the opened file is accessed using the global variable `XTRunit`. Starting with `unit 1`, the API searches the next unused FORTRAN file-unit and allocates it to open the file. The actual file name is stored in `XTRfile`.

C

Syntax	
int	XTR file(char* filename, int mode)
Input Variables	
filename	file name
mode	XTR_CREAT: create new file XTR_EXIST: open existing file

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

	XTR_CLOSE: close file
Output Variables	
Return Value	
ERR OPEN NEW XTR	cannot open file
ERR OPEN OLD XTR	file does not exist
ERR CLOSE XTR	cannot close file
ERR NOERROR	no error
Global Variables	
FILE* xtrstream	

In C, the opened file is addressed via the global variable `xtrstream`.

3.4.2.2 Function XTRdata

In read mode, sections, keywords, and key-indices are searched and the contents of the data items is transferred to the caller. Specify one of the defined section names, a keyword and if applicable a key-index in the appropriate function parameters and call *XTRdata*. Depending on the data type, *XTRdata* will return the data items in the *sdata*, *idata* and *fdata* variables, in the same sequence as defined in the file and in chapter 3.3.

Extract files are rather simple, sequential files. Therefore, the files are always searched from the beginning and are rewound for each call to XTRdata.

Writing extract files is done in two steps. Firstly, a new section name is appended to the file. Then keywords, if necessary key-indices, together with all the data items are written. The API takes care that everything is in the correct format and sequence. However, keywords are defined only in their sections and it is the programmer's responsibility to specify the correct keys together with the right values for the data items.

When using the API, always specify the section and key names, without any brackets, quotes or equal signs. Examples for programming the API in FORTRAN and C and sample `XTR` files can be found chapters 3.4.4 and 3.4.5.

FORTRAN

Syntax	
integer	function XTRdata(mode, section, key, sdata, idata, fdata)
integer	mode
character*(*)	section
character*(*)	key
character*(*)	sdata(5)
integer	idata(*)
real	fdata(*)
Input Variables	
mode	XTR_GET_KEY: find a key XTR_GET_KEYINT: find a key-index XTR_WRT_KEY: write a key or key-index XTR_WRT_SECTION: write a section
section	Name of the section
Key	name of the key
idata(1)	key-index, if applicable
I/O Variables	

sdata	string-type data items
idata	integer-type data items
fdata	real-type data items
Return Value	
ERR_SECTION_NF	section not found
ERR_KEY_NF	key not found
ERR_WRITE_FILE	write error
ERR_NOERROR	no error
Global Variables	
character*256	XTRfile
integer	XTRunit

C

Syntax	
Int	function XTR data(int mode, const char* section, const char* key, char* *sdata, long *idata, double *fdata)
Input Variables	
Mode	XTR_GET_KEY: find a key
	XTR_GET_KEYINT: find a key-index
	XTR_WRT_KEY: write a key or key-index
	XTR_WRT_SECTION: write a section
section	name of the section
Key	name of the key
idata[0]	key-index, if applicable
Input/Output Variables	
Sdata	string-type data items
Idata	integer-type data items
Fdata	real-type data items
Return Value	
ERR_SECTION_NF	section not found
ERR_NOTSECTION	invalid section specified
ERR_KEY_NF	key not found
ERR_NOTKEY	invalid key specified
ERR_WRITE_FILE	write error
ERR_INCORRECT_USE	incorrect use of function
ERR_NOERROR	no error
Global Variables	
character*256	XTRstream

3.4.3 C++ Source Code Example

```
// Example Program
int main( int argc, char *argv[] )
{
  const double version = 1.0;
  EmXtrFile xtr;
  try {
    std::cout << "*********************************************************************\n";
    std::cout << "*                             emtoasc                               *\n";
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
    std::cout << "*                    Version: ";
    std::cout << setprecision(1) << fixed << version;
    std::cout << " - " << __DATE__ << "                      *\n";
    std::cout << "*                             GFZ Potsdam                          *\n";
    std::cout << "********************************************************************\n";
    std::cout << "* emtoasc converts Emerald .raw binary data files to  *\n";
    std::cout << "* ASCII data files. Usage: emtoasc [xtr filename]                  *\n";
    std::cout << "********************************************************************\n";
    std::cout << "\n";
    std::cout.flush();

// needs 2 parameters from the commandline. The first is the program name.
// The 2nd paramter has is the name of the .xtr file associated with the data file
// to beconverted.
    if (argc == 1) {
      throw EmException(EmError::Usage, "No XTR file specified");
    } else
// expect the second parameter to be the XTR filename
// load() throws exception, if unable to load a .xtr file
    if (argc == 2) {
      xtr.load(std::string(argv[1]));
    } else {
      throw EmException(EmError::Usage, "Too many command line arguments");
    }

// The purpose of this output is to show how to access the XTR data fields
// (just a few examples here)
    std::cout << "XTR filename: " << xtr.fileName() << "\n";
    std::cout << "filename from contents: " << xtr.fileNameFromContents() << "\n";
    if (xtr.fileSec.isNameKey()) {
      std::cout << "File name: " << xtr.fileSec.nameKey()->name << "\n";
      std::cout << "Run number: " << xtr.fileSec.nameKey()->run << "\n";
// rates in xtr files underly the conversion that negative numbers have unit Hz
// while positive numbers have unit second. Use the conversion routines to handle.
      std::cout << "SampleFreq: "<<emConv2Freq(xtr.fileSec.nameKey()->sampleFreq)<<" Hz\n";
    }
    if (xtr.titleSec.isDateKey()) {
      std::cout << "Date: " << xtr.titleSec.dateKey()->str() << "\n";
    }
    if (xtr.dataSec.isChanKey()) {
      for (unsigned int i = 0; i < xtr.dataSec.nItems(); ++i) {
        std::cout << (i+1) << ". Chan FileIndex:" << xtr.dataSec.chanKey(i)->index << "\n";
        std::cout << (i+1) << ". Chan SiteIndex:" << xtr.dataSec.chanKey(i)->site << "\n";
        std::cout << (i+1) << ". Chan ChanIndex:" << xtr.dataSec.chanKey(i)->chan << "\n";
        std::cout << (i+1) << ". Chan ScaleFactor:" << xtr.dataSec.chanKey(i)->scale<<"\n";
      }
    }
    std::cout << "\n\n";
// To change a field in the .XTR, just change the variable and save the file
// for example increase the run number
    if (xtr.fileSec.isNameKey()) {
      xtr.fileSec.nameKey()->run++;
    }
    xtr.save();

// Now load the .raw file with the binary data
    EmRawFile raw;
    raw.open(emGetFileName(xtr.fileName(), true, false) + ".RAW");
    std::cout << "RAW filename: " << raw.file() << "\n";
    std::cout << "EventCount: " << raw.generalHeader().totalEvents << "\n";
    std::cout << "RecordCount: " << raw.generalHeader().totalRecs << "\n";
    std::cout << "RecordLength: " << raw.generalHeader().recLen << "Bytes\n";
    std::cout << "StartTime: " << emTimeToStr(raw.eventHeader().startTime(), true) << "\n";
    std::cout << "StopTime: " << emTimeToStr(raw.eventHeader().stopTime(), true) << "\n";
    std::cout.flush();
```

```
// start reading
// Open the file, where the ASCII data should go in
    const std::string asciiFileName = emGetFileName(xtr.fileName(), true, false) + ".ASC";
    ofstream asciiStream(asciiFileName);
    asciiStream.exceptions(ofstream::badbit);
    bool ok = asciiStream.is_open();

    vector<float> sample;
// Write only the first 100 Samples
    unsigned int nSamples = 100;
    while (ok && raw.readNextRecord(sample)) {
      for (const auto& f : sample) {
         asciiStream << f << " ";
      }
      asciiStream << "\n";
      ok = asciiStream.good();
     ok &= (--nSamples >= 0);
    }

    asciiStream.close();
    raw.close();
    std::cout << "\n Done.\n\n";
    std::cout.flush();

    std::cout << "\n\n emtoasc successfully finished\n";
    std::cout << "*********************************************************************\n";
    std::cout.flush();
    return 0;
  } catch (EmException& e) {
    std::cout << "\n\n Error: emtoasc failed\n";
    std::cout << " " << e.what() << "\n";
    std::cout << "*********************************************************************\n";
    std::cout.flush();
    return e.errorCode();
  } catch (...) {
    std::cout << "\n\n Error: emtoasc failed\n";
    std::cout << " UnknownError\n";
    std::cout << "*********************************************************************\n";
    std::cout.flush();
    return static_cast<int>(EmError::Unknown);
  }
}
```

3.4.4 FORTRAN Source Code Example

This FORTRAN sample program illustrates reading some parameters from an extract file. Please note using the include file `em_xtr.fi` is necessary as it contains definition of some global variables.

```
      program xtrtestf
c
      include ’em xtr.fi’                ! xtr-include file
      integer XTRdata, XTRfile           ! API- functions
      external XTRdata, XTRfile
c
      integer i, err, idat(5), nsite, ncomp, nchan
      real rdat(5)
      character*4 compnm(5), sitenm(5)
      character*15 key, section, xtrfn
      character*40 sdat(5)
c
      data xtrfn/’10ORB000.XTR’/    ! xtr- filename
c
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
      err=XTRfile(xtrfn, XTR_EXIST) ! open existing xtr - file
      section='FILE'                ! get data file name
      key = 'NAME'
      err=XTRdata( XTR_GET_KEY, section, key, sdat, idat, rdat)
      write(*,*)'data file: ',sdat(1)
      write(*,*)
      section = 'SITE'
      nsite = idat(1)
      key = 'ITEMS'                 ! get number of sites
      err = XTRdata( XTR_GET_KEY, section, key, sdat, idat, rdat)
      nsite = idat(1)
      write(*,'(1x,a,i2)')'number of sites:',nsite
      do i=1, nsite
        key='NAME'                         ! get site names
        idat(1)=i
        err=XTRdata( XTR_GET_KEYINT, section, key, sdat, idat, rdat)
        write(*,*)'site name: ',sdat(1)
        sitenm(i)=sdat(1)                  ! store site names
      enddo
      write(*,*)
c
      section='CHANNAME'
      key = 'ITEMS'                 ! get number of channel names
      err = XTRdata(XTR_GET_KEY, section, key, sdat, idat, rdat)
      ncomp = idat(1)
      write(*,'(1x,a,i2)')'number of component names:',ncomp
c
      do i=1, ncomp
        key='NAME'                         ! get channel names
        idat(1)=i
        err=XTRdata(XTR_GET_KEYINT, section, key, sdat, idat, rdat)
        write(*,*)'component name: ',sdat(1)
        compnm(i)=sdat(1)                  ! store channel names
      enddo
      write(*,*)
c
      section='DATA'
      key = 'ITEMS'                        ! get number of channels
      err=XTRdata(XTR_GET_KEY, section, key, sdat, idat, rdat)
      nchan = idat(1)
      write(*,'(1x,a,i2)')'number of data channels:',nchan
      write(*,*)'channel site component rotation scaling modules'
      do i=1, nchan
        key='CHAN'                         ! get channel info
        idat(1)=i
        err=XTRdata(XTR_GET_KEYINT, section, key, sdat, idat, rdat)
        write(*,1000)idat(1), sitenm(idat(2)), compnm(idat(3)),rdat(2),
     &               rdat(4), idat(4)
      enddo
      err=XTRfile( xtrfn, XTR_CLOSE)
1000  format(i5,4x,2(a5,4x),f6.3,3x,f6.3,4x,i5)
      end
```

Below is the screen output produced by the above FORTRAN source. The data originated from a dense magnetic variation mapping experiment, where vertical magnetic fields were recorded at 5 sites. The horizontal magnetic fields from site 10*OR* were used as reference fields for the other sites.

```
data file: 10ORB000.RAW
number of sites: 5
site name: 10EE
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
site name: 10EO
site name: 10OR
site name: 10WO
site name: 10WW
number of component names: 3
component name: Hx
component name: Hy
component name: Hz
number of data channels: 7
channel site  component  rotation  scaling  modules
   1    10OR     Hx         .000     .025        1
   2    10OR     Hy       90.000     .025        1
   3    10WW     Hz         .000    -.025        1
   4    10EE     Hz         .000    -.025        1
   5    10OR     Hz         .000    -.025        1
   6    10WO     Hz         .000    -.025        1
   7    10EO     Hz         .000    -.025        1
```

3.4.5 C Source Code Example

The following source code shows how to use of the API to read and write extract files in C. Please note, that EM XTR.H is required. The code demonstrates only principles, the contents of the generated extract file is incomplete and not very useful for normal MT usage.

```
#include<stdio.h>
#include<string.h>
#include ”EM_XTR.H”

int XTR file( char*, int);
int XTR data( int, char*, char*, char* *, long *, double *);
double fdata[5];
long idata[5];
char *sdata[5];

int main (void)
{
  int i,numsites;
  char b0[MAXLINELEN],b1[80],b2[40],b3[MAXKEYLEN],b4[MAXKEYLEN];
  sdata[0]=b0; sdata[1]=b1; sdata[2]=b2; sdata[3]=b3; sdata[4]=b4;
/* write XTR file, open file */
  XTR_file(”test.xtr”,XTR CREAT);
/* write title - section */
  XTR_data(XTR_WRT_SECTION, "TITLE", "", sdata, idata, fdata);
  strcpy(sdata[0], "Oliver Ritter");
  XTR_data(XTR WRT KEY, "TITLE", "AUTHOR", sdata, idata, fdata);
  strcpy(sdata[0], "XTRTESTC");
  fdata[0]=1.0;
  XTR_data(XTR_WRT_KEY, "TITLE", "VERSION", sdata, idata, fdata);
  strcpy(sdata[0], "wee EMERALD extract file test program");
  XTR_data(XTR_WRT_KEY, "TITLE", "COMMENT", sdata, idata, fdata);
/* write site - section */
  XTR data(XTR_WRT_SECTION, "SITE", " ", sdata, idata, fdata);
/* --> write number of sites: ITEMS- section */
  idata[0]=2L;
  XTR_data(XTR_WRT_KEY, "SITE", "ITEMS", sdata, idata, fdata);
/* --> write first site */
  strcpy(sdata[0], "SIT1");
  idata[1]=101L;
  idata[2]=0L;
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
  idata[3]=-1L;
  XTR_data(XTR_WRT_KEY, "SITE", "NAME", sdata, idata, fdata);
/* --> write second site */
  strcpy(sdata[0], "SIT2");
  idata[1]=102L;
  idata[2]=0L;
  idata[3]=-1L;
  XTR_data(XTR_WRT_KEY, "SITE", "NAME", sdata, idata, fdata);
/* close xtr-file */
  XTR_file(”test.xtr”,XTR_CLOSE);
/* ----------------------------------------------------------------*/
/* read XTR file, open file */
  XTR_file(”test.xtr”,XTR_EXIST);
/* read title section */
  XTR_data(XTR_GET_KEY, "TITLE", "VERSION", sdata, idata, fdata);
  printf("<== created by: %s version: %g\n",sdata[0],fdata[0]);
  XTR_data(XTR_GET_KEY, "TITLE", "AUTHOR", sdata, idata, fdata);
  printf("<== author: %s\n",sdata[0]);
/* read site - section, get number of sites */
  XTR_data(XTR_GET_KEY, "SITE", "ITEMS", sdata, idata, fdata);
  numsites=(int)idata[0];
/* --> read sites */
  for (i=1; i<=numsites; i++){
    idata[0] = (long)i;
    XTR_data(XTR_GET_KEYINT, "SITE", "NAME", sdata, idata, fdata);
    printf("<== %d. site: %s - site no.: %ld\n",i,sdata[0],idata[1]);
  }
/* close XTR file */
  XTR_file("test.xtr",XTR CLOSE);
  return(ERR NOERROR);
} /* main */
```

The sample program above produces the following dummy-extract file.

```
[TITLE]
’AUTHOR=’ ’Oliver Ritter’
’VERSION=’ ’XTRTESTC’ 1.00000
’COMMENT=’ ’wee EMERALD extract file test program’
[SITE]
’ITEMS=’ 2
’NAME=’ 1 ’SIT1’ 101 0 -1
’NAME=’ 2 ’SIT2’ 102 0 -1
```

The screen output, produced by the above program while reading the extract file, should look like this:

```
<== created by: XTRTESTC version: 1
<== author: Oliver Ritter
<== 1. site: SIT1 - site no.: 101
<== 2. site: SIT2 - site no.: 102
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

4 EMERALD Data Files

This chapter describes the EMERALD type data files, which are normally binary. To ensure compatibility with earlier versions of FORTRAN, data files are internally organized in records (length is specified in bytes). Records allows random access in FORTRAN. The Windows- type or Little Endian binary representation of numbers is assumed.

4.1 Overview

All EMERALD data files consist of three principal building blocks:

1. The <u>general header</u> (GH) contains information about file structure, such as word length or number of data per record.
2. The <u>event header</u> (EH) contains information specific for the block of data that follows, such as start and end time of an event.
3. The <u>data section</u> contains data in the form of rows and columns of data items.

There is one GH, but one or many EHs in every file. Each EH is followed by a data section. Both, GH and EH are written as a sequence of ASCII characters into the binary files. The data items in the data sections are written binary (or ASCII).

The files are organized in records of a fixed length. Between files, the length of the records may vary. The record length is calculated according to: (*word length* $*$ *number of rows*). The word length depends on the data type; single precision real numbers have a word length of 4 bytes.

Depending on the record length GH and EH extend over one or several records. EH, GH, and the data sections always start on a new record.

While all data in one file must be of the same type and word length, the number of data (records) following an EH can be different. If the data is written as formatted numbers (strings), they must have the same string length.

~~To store additional information in the files, the general header can be extended. The extended GH consists of user defined strings, separated by blanks. The maximum string length for is set to 1024 bytes.~~

4.1.1 The General Header in FOTRAN and C

Below is the definition of the GH in FORTRAN notation:

FORTRAN

Type	Name	Description
integer	GHwrdl	length of one data item in bytes
character*4	GHftyp	identifier for file type (see text)
character*6	GHvers	version of this GH (see text)
integer	GHncha	number of data items per record, e.g. number of channels
integer	GHrecl	= GHwrdl * GHncha
integer	GHtotr	total number of records in this file
integer	GHfEHr	record of first EH in file
integer	GHnev	total number of EHs in this file

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

character*4	GHproc	processing identifier (see text)
integer	GHext	number of data items stored in the extended header (see text)
character	GHstr*(?)	contains the extended header. Maximum length of the string is defined in GH MAXSTR

The format of the general header was revised a couple of times, mostly to accommodate for larger data files. `GHvers` (FORTRAN) or `gh.version` (C, see below) must be defined by the calling program. The current revision of the GH is version 5.0.

Extension of the GH is an obsolete feature which means `GHext` should always be equal to 0.

```
      write (str,100x,err=999) GHrecl, GHftyp, GHwrdl, GHvers,
     &                         GHproc, GHncha, GHtotr, GHfEHr,
     &                         GHnev, GHext
c --- for GHvers < 4.0 (since 5.2.1996)
1000  format(i4.4,1x,a3,1x,i3.3,1x,a5,1x,a3,1x,i3.3,1x,i8.8,1x,
     &       i4.4,1x,i5.5,1x,i2.2,2x)
c --- for GHvers < 5.0 (since 7.11.2001)
1001  format(i4.4,1x,a3,1x,i3.3,1x,a5,1x,a3,1x,i3.3,1x,i8.8,1x,
     &       i4.4,1x,i7.7,1x,i2.2,2x)
c --- for GHvers < 6.0 (since 1.7.2003)
1002  format(i4.4,1x,a3,1x,i3.3,1x,a5,1x,a3,1x,i3.3,1x,i9.9,1x,
     &       i4.4,1x,i7.7,1x,i2.2,2x)
```

C

Type	Name	Description
typedef	struct GH{	
int	word_length	length of one data item in bytes
char	file_type[4]	identifier for file type (see text)
char	version[6]	version of this GH (see text)
int	num_channels	number of data items per record, e.g. number of channels
int	record_length	=gh.totalrecs*gh.num channels
long	totalrecs	total number of records in this file
long	first_EH_record	record of first EH in file
long	num_events	total number of EHs in this file
char	processing_ident[4]	processing identifier (see text)
int	Extended	number of data items stored in the extended header (see text)
}GH		
char	char ghstring[?]	contains the extended header. Maximum length of the string is defined in GH_MAXSTR

The contents of the general header in C is written as follows:

```
#define FMT ”%0*d %*s %0*d %*s %*s %0*d %0*ld %0*ld %0*ld %0*d ”

/* for GH vers < 4.0 (since 5.2.1996) */
#define GHWL ( 3 )                   /* gh.word_length */
#define GHFT ( 3 )                   /* gh.file_type */
#define GHVS ( 5 )                   /* gh.version */
#define GHNC ( 3 )                   /* gh.num_channels */
#define GHRL ( 4 )                   /* gh.record_length */
#define GHTR ( 8 )                   /* gh.totalrecs */
#define GHFE ( 4 )                   /* gh.first_EH_records */
#define GHNE ( 4 )                   /* gh.num_events */
#define GHPT ( 3 )                   /* gh.processing_ident */
#define GHEX ( 3 )                   /* gh.extended */
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
/* 7 digits for event counter GH vers < 5.0 (since 7.11.2001) */
#define GHNE4 ( 6 )                  /* gh.num_events */

/* 9 digits for records: GH vers < 6.0 (since 1.7.2003) */
#define GHTR5 ( 9 )                  /* gh.totalrecs */
iret=fprintf(dastream, FMT,
             GHRL,gh.record_length,
             GHFT, gh.file_type,
             GHWL, gh.word_length,
             GHVS, gh.version,
             GHPT, gh.processing_ident,
             GHNC, gh.num_channels,
             GHTR/5, gh.totalrecs,
             GHFE, gh.first_EH_record,
             GHNE/4, gh.num events,
             GHEX, gh.extended);
```

Computer codes use file type identifiers to decide how to handle the data. This parameter contains 3 uppercase letters with the following meaning:

- first letter:
  - ’A’: ASCII data
  - ’B’: binary data
- second letter:
  - ’C’ : complex value
  - ’F’ : floating point value
  - ’I’ : integer value
  - ’R’ : floating point (real) value
- third letter is a repetition of the word length parameter:
  - ’1-F’ : hex number specifying the length of the formatted string data (A-type)
  - ’1-F’ : hex number specifying the word length binary data (B-type)
  - ’0’ : word / string length is 10
  - ’*’ : if word or string length exceeds 16

The data file and the API must be the same revision. Check the `#define VERSION` statement in `em_file.h` and `parameter VERSION` in `em_file.fi`. Currently, EMERALD -type data files are of version 2.

The 3-letter processing identifiers contain some information on the data file contents. They are usually also used for the filename extensions (see section 4.2 for further details).

~~The number of items stored in the extended general header is stated to ease reading the extended GH block. It is assumed, that the extended GH is organized as one string, which is divided by blanks, to separate the individual data items~~.

4.1.2 The Event Header in FORTRAN and C

FORTRAN

Type	Name	Description
integer	EHtst	start time, elapsed seconds since 1/1/1970

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

integer	EHtstm	elapsed micro-seconds since EHtst
integer	EHtsp	stop time, elapsed seconds since 1/1/1970
integer	EHtspm	elapsed micro-seconds since EHtsp
integer	EHreif	record of the current EH
integer	EHrneh	record of next EH in file
integer	EHrpeh	record of previous EH in file
integer	EHrnod	the number of records (data-rows) to follow this EH
integer	EHrsod	the first record of data after this EH
double precision	EHdvl1	EH value1 (see text)
double precision	EHdvl2	EH value2 (see text)
double precision	EHdvl3	EH value3 (see text)
integer	EHext	reserved, always 0

Changes of the GH also have side effects on the format of the EH. The current version of the GH is version 5.0. `EHext` is not used.

```
      write (str,100x,err=999) EHtst, EHtstm, EHtsp , EHtspm,
     &                         EHdvl1, EHdvl2, EHdvl3, EHreif, EHrneh,
     &                         EHrpeh, EHrnod, EHrsod, EHext
c --- for GHvers < 4.0 (since 5.2.1996)
 1001 format(2(i10.10,1x,i6.5,1x),3(g11.5,1x),5(i6.6,1x),i3.3,2x)
c --- for GHvers < 5.0 (since 7.11.2001)
 1002 format(2(i10.10,1x,i6.5,1x),3(g11.5,1x),5(i8.8,1x),i3.3,2x)
c --- for GHvers < 6.0 (since 1.7.2003)
 1003 format(2(i10.10,1x,i6.5,1x),3(g11.5,1x),5(i9.9,1x),i3.3,2x)
```

C

Type	Name	Description
typedef	struct EH{	
typedef	struct EHTIME{	
int	start	start time, elapsed seconds since 1/1/1970
int	startms	elapsed micro-seconds since EHtst
int	stop	stop time, elapsed seconds since 1/1/1970
int	stopms	elapsed micro-seconds since EHtsp
}EHTIME		
Typedef	struct RECS{	
int	EH_in_file	record of the current EH
int	next_EH	record of next EH in file
int	previous_EH	record of previous EH in file
int	num_of_data	the number of records (data-rows) to follow this EH
int	start_of_data	the first record of data after this EH
}RECS		
typedef	struct DATA{	
double	value1	EH value1 (see text)
double	value2	EH value2 (see text)
double	value3	EH value3 (see text)
}DATA		

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
  int                   EHext                    reserved, always 0
}EH
```

In C notation, the EH is constructed as follows:

```
   #define EHTST       ( 10 )       /* start time */
   #define EHTSTM      ( 6 )        /* start time ms */
   #define EHTSP       ( 10 )       /* stop time */
   #define EHTSPM      ( 6 )        /* stop time ms */

   #define EHD1M       ( 11 )
   #define EHD1E       ( 4 )
   #define EHD2M       ( 11 )
   #define EHD2E       ( 4 )
   #define EHD3M       ( 11 )
   #define EHD3E       ( 4 )

   #define RCEF        ( 6 )        /* RECS structure */
   #define RCEN        ( 6 )
   #define RCEP        ( 6 )
   #define RCND        ( 6 )
   #define RCSD        ( 6 )

   #define RCEF3       ( 8 )        /* RECS structure */
   #define RCEN3       ( 8 )
   #define RCEP3       ( 8 )
   #define RCND3       ( 8 )
   #define RCSD3       ( 8 )

   #define RCEF5       ( 9 )        /* RECS structure */
   #define RCEN5       ( 9 )
   #define RCEP5       ( 9 )
   #define RCND5       ( 9 )
   #define RCSD5       ( 9 )

#define FMT2 "%0*ld %0*ld %0*ld %0*ld" \
             "%+#0*.*lG %+#0*.*lG %+#0*.*lG" \
             "%0*ld %0*ld %0*ld %0*ld %0*ld"\
             "%0*d"
iret=fprintf(dastream, FMT2,
                       EHTST, eh.ehtime.start,
                       EHTSTM, eh.ehtime.startms,
                       EHTSP, eh.ehtime.stop,
                       EHTSPM, eh.ehtime.stopms,
                       EHD1M, EHD1E, eh.data.value1,
                       EHD2M, EHD2E, eh.data.value2,
                       EHD3M, EHD3E, eh.data.value3,
                       RCEF/3/5, eh.recs.EH_in_file,
                       RCEN/3/5, eh.recs.next_EH,
                       RCEP/3/5, eh.recs.previous_EH,
                       RCND/3/5, eh.recs.num_of_data,
                       RCSD/3/5, eh.recs.start_of_data,
                       EXT, eh.extended);
```

The event header consist of three principal components to describe the data section that follows.

- Four integer values define start and stop times of the subsequent data section. The date and time values are specified as universal time (UT), a presentation where the seconds elapsed since 1/1/1970 are counted. For fine-tuning and to synchronize high

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

frequency data, the one second time intervals can be divided further into parts of a million of a second.

- The record section of the EH supports navigation in the files. They help to move backwards and forwards in the file, from one EH to another, without having to read the data in between.
- The last part of the EH consists of supplemental number values (floating point). These can be chosen freely. Typically they depend on the data that is stored (see chapter 4.2).

function name	**FORTRAN file name**	**C file name**	**short description**
EMfile	emfilef.for	emfilec.c	to open and close a new or existing file
GHmake	ghmakef.for	ghmakec.c	to generate and update the GH of a file
GHget	ghgetf.for	ghgetc.c	to read the contents of a general header from a file.
~~GHadd~~		~~ghaddc.c~~	~~to add an element to the extended GH (obsolete).~~
EHmake	ehmakef.for	ehmakec.c	to write an EH at the current position into a data file.
EHget	ehgetf.for	ehgetc.c	to read the contents of an EH from a file.

Table 2: Overview of the EMERALD file API for FORTRAN and C

4.1.3 The Data Section

The easiest way to understand how to read and write the data sections, will be to study the example programs given in chapters 3.2.2, 4.3.3, and 4.3.4. Although the EMERALD- file API does not include functions to read and write data, the file headers provide all the information necessary to access the data in the correct way.

There are only a few of the header parameters that must be provided by the user. For the general header, the *word length* of one data item and the number of data *items per row* are required to calculate the record length of the file. The *file type* identifier, the *version* of the GH and the *processing identifier* should contain meaningful values, because they are normally examined by the programs that read the data files. The information about the *record length*, the *total number of records* of the file and the *number of events* are calculated by the API.

The most critical event header parameter is the *number of data records* that must be specified by the user. The other record parameters are normally handled by the API. Be very careful if you use `EHmake` in the direct write mode, since the contents of the EH elements is not checked.

It is the responsibility of the programmer to calculate the correct *start- and stop times* and to give meaningful information for the EH *values*.

4.2 Naming Convention and Definitions for the EMERALD Data Processing.

The EMERALD data processing uses predefined file extensions for time series data (RAW / TD), Fourier spectra (FD / CAL) and cross- and auto spectra (SP). More details are given below.

For EM processing the variable EH values (see chapter 4.1.2) often describe a frequency or a period. Since round off errors are easily introduced when transforming from time domain to frequency domain, the following definition is used: positive values describe a frequency (dimension *Hz)* and negative values a period (dimension *seconds*).

4.2.1 Raw Data (.RAW)

.RAW files contain time series data as produced by the instruments or after digital filtering. The data is a continuous stream of time series data. Data are typically stored as 4 byte long floating point values stored in binary format. For an MT setup the channel sequence could be Bx, By, Bz, Ex, Ey. If all of the data in the file are continuous, the file contains only the GH and one EH. The EH-values describe: sampling rate, low-pass cut-off frequency, high-pass cut-off frequency.

Over the years several naming conventions have been used, initially limited by the 8.3 character limitation for filenames imposed by the MS-DOS operating system:

Filename convention:

SSUBBRCC.RAW

Example: `40S1FA01.RAW`

Meaning of characters:

SS: site number (2 digits)

U: unit. Valid characters are [S | H | M]. S: period [s]; H: Frequency [Hz], M: a mix of [Hz] and [s]. Unit is related to the next two characters (BB).

BB: Frequency band identifier. Valid characters are [0..9, A-F] to compose a hexadecimal number. The first letter refers to the low-pass filter setting, the second letter to the high-pass filter setting. All values are multiples of 2. The letter ‘A’ translates to:
$2^{0Ax} = 2^{10} = 1024$. The combination of letters ‘M24’ describe the band-pass: $2^2 = 4\ [Hz]$ to $2^4 = 16[s]$. The letter ‘F’ used as a high-pass value means that no high-pass filter was used, i.e. in the example above ‘S1F’ means a low-pass-only filter with a cut-off period of 2 s was used.

R: Run number. Valid characters are [0..9, A-Z]. Used to indicate a different run was started for that site, e.g. after a sensor was changed.

CC: Counter. Valid characters are [0..9]

Approximately used:

1995 – 2010

Filename convention:

SSSS_LPnnnnnUU_HPnnnnnUU_Rnnn_Wnnn.RAW

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

Example: `0008_LP00200Hz_HP00000s_R006_W024.RAW`

Meaning of characters:

SSSS: site number (4 digits)

LP: Low-pass filter.

HP: High-pass filter.

nnnnnUU: nnnn: 5 digit integer number, valid characters for 'n' are [0..9]. UU: unit [Hz | s] to identify frequency band.

Rnnn: 3 digit Run number. Valid characters for 'n' are [0..9]. Used to indicate a different run was started for that site, e.g. after a sensor was changed.

Wnnn: 3 digit time Window counter, typically used for data recorded in scheduled or burst modes. Valid characters for 'n' are [0..9].

Approximately used:

2010 – 2015

Filename convention:

SSSS_LPnnnnnUU_HPnnnnnUU_Dyyyyddd_Thhmmss.RAW

Example: `0008_LP00200Hz_HP00000s_D2015126_T140000.RAW`

Meaning of characters:

SSSS: site number (4 digits)

LP: Low-pass filter.

HP: High-pass filter.

nnnnnUU: nnnn: 5 digit integer number, valid characters for 'n' are [0..9]. UU: unit [Hz | s] to identify frequency band.

Dyyyyddd: 7 digit date information. Valid characters for 'y/d' are [0..9]. 4 digit year 'yyyy' and 3 digit day of year 'ddd' (001 = January.1st of that year)

Thhmmss: 6 digit time information, 2 digit hour 'hh', 2 digit minute 'mm' and 2 digit second 'ss'. Valid characters for 'h/m/s' are [0..9].

Approximately used:

2015 – Present

4.2.2 Time Domain Data (.TD_)

.TD files contain time series data with time segments which have differing lengths or are discontinuous. Data are typically stored as 4 byte long floating point values stored in binary format. .TD files typically contain the GH and several EHs. The EH-values describe: sampling rate, low-pass cut-off frequency, high-pass cut-off frequency

For the file naming convention, see chapter 4.2.1.

4.2.3 Frequency Domain Data (.FD_)

.FD files contain Fourier coefficients, ever before or after correcting for instrument responses (see chapter 4.2.5). Fourier coefficients are typically stored as 8 byte long complex numbers stored in binary format. The Fourier transform is typically applied to time segments of a fixed length (e.g. with 2^n samples) and hence, *.FD* files typically contain the

GH and several EHs. The EH-values describe: lowest frequency, frequency spacing, low-pass cut-off frequency.

For the file naming convention, see chapter 4.2.1.

4.2.4 Cross- and Auto-Spectra (.SP_)

.SP files contain averaged Fourier coefficients, before or after correcting for instrument responses (see chapter 4.2.5). Cross- and auto-spectra are stored as 4 byte long floating point numbers in binary format. Auto spectra are real numbers while cross-spectra are complex numbers. To store the data efficiently, the data are stored in matrix form. The example below shows this for 5-channel MT example:

$$\begin{bmatrix} <BxBx> & \mathrm{Re}<BxBy> & \mathrm{Re}<BxBz> & \mathrm{Re}<BxEx> & \mathrm{Re}<BxEy> \\ \mathrm{Im}<ByBx> & <ByBy> & \mathrm{Re}<ByBz> & \mathrm{Re}<ByEx> & \mathrm{Re}<ByEy> \\ \mathrm{Im}<BzBx> & \mathrm{Im}<BzBy> & <BzBz> & \mathrm{Re}<BzEx> & \mathrm{Re}<BzEy> \\ \mathrm{Im}<ExBx> & \mathrm{Im}<ExBy> & \mathrm{Im}<ExBz> & <ExEx> & \mathrm{Re}<ExEy> \\ \mathrm{Im}<EyBx> & \mathrm{Im}<EyBy> & \mathrm{Im}<ExBz> & \mathrm{Im}<EyEx> & <EyEy> \end{bmatrix}$$

<BxBx>:= $\sum_{i=1}^{N} Bx_i\, Bx_i^*$ is the auto-spectra of Bx. <BxBy>:=$\sum_{i=1}^{N} Bx_i\, By_i^*$ is the cross-spectra between Bx and By. The * denotes the complex conjugate. The auto-spectra are stored on the diagonal of the matrix. The real parts of the cross-spectra are stored in the upper triangle of the matrix, the imaginary parts of the cross-spectra are stored in the lower triangle of the matrix. *.SP* files typically contain the GH and several EHs. The EH-values describe: centre frequency, frequency band width, number degrees of freedom.

For the file naming convention, see chapter 4.2.1.

4.2.5 Calibration Data (.CAL)

.CAL files contain Fourier coefficients to correct for instrument responses in the frequency domain. The layout, i.e. the frequency distance between any two values, must match those of corresponding time series segment, i.e. with 2^n samples. Calibration data are typically stored as 8 byte long complex numbers stored in binary format. *.CAL* files typically contain the GH and several EHs. The EH-values describe: lowest frequency, frequency spacing, low-pass cut-off frequency.

For the file naming convention, see chapter 4.2.1.

4.3 The Application Program Interface for EMERALD Data Files

4.3.1 The EMERALD Data File Interface in C++

```
// ---------------------------------------------------------------
// ----------------------- Emerald Data Files-----------------------
// ---------------------------------------------------------------

// Binary Data Files
class EmRawFile
{
public:
  EmRawFile();
```


Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

```
// General Header File Information (one per file)
  struct GeneralHeader {
    string str() const;
    unsigned int recLen;
    string type;
    unsigned int wordLen;
    string vers;
    string proc;
    unsigned int wordsPerRec;
    unsigned long totalRecs;
    unsigned long firstEvent;
    unsigned long totalEvents;
  };
// Event Header (arbitrary number of event headers per files)
// in time series data (RAW) there is usually only ONE event
  struct EventHeader {
    time_t start;
    unsigned int startUs;
    time_t stop;
    unsigned int stopUs;
    double f0, f1, f2;
    unsigned long recInFile;
    unsigned long totalRecs;
    unsigned long dataStartRec;
  };

// Open/Close file for reading and writing
  void open();
  bool isOpen() const;
  void close();
  void create(unsigned int wordLen, unsigned int wordsPerRec, const string& type, const
string& proc);

// change information of current event header
  void setEventHeaderStartTime(double t);
  void setEventHeaderStopTime(double t);
  void setEventHeaderData(double f0 = 0.0, double f1 = 0.0, double f2 = 0.0);

// jump to next/prev event
  bool loadNextEventHeader();
  bool loadPrevEventHeader();

// read next sample (jump to a sample and start reading from there)
  void setNextReadSample(unsigned long number);
  template <class T>
  bool readNextRecord(vector<T>& rec);

// append a new event header to file, append a sample at the end of the file
  void appendEventHeader();
  template <class T>
  bool appendRecord(const vector<T>& rec);

  unsigned long recordsWritten() const;
  string file() const;

// read only access to event header and general header
  const EventHeader& eventHeader() const;
  const GeneralHeader& generalHeader() const;
};
```

4.3.2 The EMERALD Data File Interface in FORTRAN and C

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

The EMERALD data file API consists of functions to read and write the GH and EH in FORTRAN and C. It is left to the programmer to supply applications with the code to write and read the data sections.

Most information between the general and event headers and the application are transferred as global variables. These global variables are declared in external files, which must be included in all routines that make use of the EMERALD file API. The FORTRAN and C include files are called `em_file.fi` and `em_file.h`, respectively.

`EMfile` is always the first and the last function that any program calls. The first call is to open an existing or new file and finally, to close it. If the application is to read a data file, `GHget` is called next to retrieve the `GH` from a previously opened file. The whole file can then be read with subsequent calls to `EHget`, followed by the application specific data-read routine.

To write to a new file, the program should call the `GHmake` function. Similar to `EMfile`, `GHmake` must be called a second time to update the `GH`, after all the data with has been written. `EH(s)` are generated with subsequent calls to `EHmake`. After an `EH` has been generated, the application can write a section of data to the file. If the number of data items is not known in advance, `EHmake` can be called for a second time, after the last data item has been written. The `EH` will then be updated by the API (see section 4.5.4).

~~To use an extended GH, the header string variable and the extended elements counter must be set before GHmake is called. Usually, this is achieved by subsequent call of GHadd. GHget returns the extended header, but the contents is lost, as soon as the application activates another EMERALD data file.~~

All functions return integer values. If no error occurred, the functions return `ERR_NOERROR (= 0)`. You can check the include files to get the integer values of an error code. However, in most cases, the application program will be halted by the API error handler if an error occurs. The user will receive an information why and where the error handler was triggered.

See chapter 4.6 for example programmes in FORTRAN and C.

4.3.2.1 Function EMfile

`EMfile` is called to open or to close an EMERALD data file. The data file may exist or may be a new file. More than one data file can be opened by the same application.

FORTRAN

Syntax	
integer	function EMfile(fn, mode)
character	fn*(*)
integer	mode
Input Variables	
fn	file name
mode	EMF_CREAT: create new file EMF_EXIST: open existing file EMF_CLOSE: close file

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

Output Variables	
Return Value	
ERR_OPEN_NEW_FILE	cannot open file
ERR_OPEN_OLD_FILE	file does not exist
ERR_CLOSE_FILE	file cannot be closed
ERR_NOERROR	no error
Global Variables	
character*256	DAfile
integer	DAunit
COMMON	/EMFN/ DAfile, DAunit

In FORTRAN, the opened file is accessed via the global variable `DAunit`. Starting with unit 1, the API searches the next unused file unit and allocates it to open the file. The actual file name is stored in `DAfile`. If more than one file is opened, only the name of the most recently opened file is kept.

Up to `MAXFILEBUF` files can be open at the same time. To increase this number, `EM_file.fi` must be modified.

C

Syntax	
int	EM file(char* filename, int mode)
Input Variables	
filename	file name
mode	CREAT: create new file EXIST: open existing file CLOSE: close file
Output Variables	
Return Value	
ERR_OPEN_NEW_FILE	cannot open file
ERR_OPEN_OLD_FILE	file does not exist
ERR_CLOSE_FILE	file cannot be closed
ERR_NOERROR	no error
Global Variables	
FILE*	dastream
int	dahandle

The opened file is addressed via the global variable `dastream.` `dahandle` is used internally. Up to `MAXFILEBUF` files can be open at the same time. To increase this number, `EM_file.h` must be modified.

4.3.2.2 Function GHmake

FORTRAN

Syntax	
integer	function GHmake(mode)
integer	mode
Input Variables	
mode	GH_INIT: generate GH for new file GH_SET: update GH on new file, after all data has been written

	GH_WRITE: direct write of GH
Output Variables	
Return Value	
ERR_WRITE_GH1	cannot write GH after a call with GH_WRITE
ERR_WORD_LENGTH	invalid word length
ERR_NUM_CHAN	invalid number of channels
ERR_WRITE_GH2	cannot write GH after a call with GH_INIT
ERR_GHREAD_EH	cannot read EH while updating GH
ERR_WRITE_GH3	cannot write GH while updating GH
ERR_NOERROR	no error
Global Variables	
COMMON	/GHBLOCK/ GHwrdl, GHftyp, GHvers, GHncha, GHrecl, GHtotr, GHfEHr, GHnev, GHproc, GHext, GHstr

C

Syntax	
int	GHmake(int tell)
Input Variables	
tell	GH_INIT: generate GH for new file GH_SET: update GH on new file, after all data has been written GH_WRITE: direct write of GH
Output Variables	
Return Value	
ERR_WRITE_GH1	cannot write GH after a call with GH_WRITE
ERR_WORD_LENGTH	invalid word length
ERR_NUM_CHAN	invalid number of channels
ERR_WRITE_GH2	cannot write GH after a call with GH_INIT
ERR_GHREAD_EH	cannot read EH while updating GH
ERR_WRITE_GH3	cannot write GH while updating GH
ERR_NOERROR	no error
Global Variables	
GH	gh;
char	ghstring[MAXGHSTR];

4.3.2.3 Function GHget

FORTRAN

Syntax	
integer	function GHget()
Input Variables	
Output Variables	
Return Value	
ERR_READ_ITEMS_GH	error reading GH
ERR_NOERROR	no error
Global Variables	
COMMON	/GHBLOCK/ GHwrdl, GHftyp, GHvers, GHncha, GHrecl, GHtotr, GHfEHr, GHnev, GHproc, GHext, GHstr

C

Syntax	
int	GHget(int tell)
Input Variables	
tell	reserved
Output Variables	
Return Value	
ERR_READ_ITEMS_GH	error reading GH
ERR_READ_ITEMS_GHSTR	error reading extended GH
ERR_NOERROR	no error
Global Variables	
GH	gh;
char	ghstring[MAXGHSTR];

4.3.2.4 *Function EHmake*

FORTRAN

Syntax	
integer	function EHmake(mode, event, npts)
integer	mode
integer	event
integer	npts
Input Variables	
mode	reserved
event	!= 0 : write EH direct, at record *event* = 0 : write next EH
npts	< 0 : write EH incomplete, number of data items not known in advance > 0 : write EH complete, npts records of data to follow
Output Variables	
Return Value	
EH_COMPLETE	EH is completely written, no error
EH_UNCOMPLETE	EH is not completely written, warning
ERR_WRITE_EH1	direct write error
ERR_WRITE_EH2	write error of incomplete EH
ERR_WRITE_EH3	write error of complete EH
Global Variables	
COMMON	/EHBLOCK/EHtst, EHtstm, EHtsp , EHtspm, EHreif, EHrneh, EHrpeh, EHrnod, EHrsod, EHdvl1, EHdvl2, EHdvl3, EHext

C

Syntax	
Int	EHmake(int tell, long event, long npts)
Input Variables	
tell	reserved
event	!= 0 : write EH direct, at record event = 0 : write next EH
npts	< 0 : write EH incomplete, number of data items not known in advance > 0 : write EH complete, npts records of data to

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

	follow
Output Variables	
Return Value	
EH_COMPLETE	EH is completely written, no error
EH_UNCOMPLETE	EH is not completely written, warning
ERR_WRITE_EH1	direct write error
ERR_WRITE_EH2	write error of incomplete EH
ERR_WRITE_EH3	write error of complete EH
Global Variables	
EH	eh;

4.3.2.5 Function EHget

FORTRAN

Syntax	
integer	function EHget(irec)
integer	irec
Input Variables	
irec	FIRST_EVENT: get first EH THIS_EVENT: read current EH NEXT_EVENT: get next EH PREV_EVENT: get previous EH > 0 : get EH direct, at irec
Output Variables	
Return Value	
ERR_NO_PREV_DATA	no previous data available
ERR_RECORD_INVALID	irec invalid
Global Variables	
COMMON	/EHBLOCK/ EHtst, EHtstm, EHtsp , EHtspm, EHreif, EHrneh,EHrpeh, EHrnod, EHrsod, EHdvl1, EHdvl2, EHdvl3, EHext

C

Syntax	
Int	EHget(int tell, long record)
Input Variables	
Tell	reserved
record	FIRST_EVENT: get first EH THIS_EVENT: read current EH NEXT_EVENT: get next EH PREV_EVENT: get previous EH LAST_EVENT: get last EH > 0 : get EH direct, at record
Output Variables	
Return Value	
ERR_NO_PREV_DATA	no previous data available
ERR_RECORD_INVALID	record invalid
Global Variables	
EH	eh;

4.3.3 FORTRAN Source Code Exemple

Scientific Technical Report STR15/08 Data, Deutsches GeoForschungsZentrum GFZ. http://doi.org/10.2312/GFZ.b103-15082

The following program first creates and then reads the data file test.dat. Please note this version of the GH is now outdated. The actual version is version 5.0.

```
      program emtestf
      include 'em file.fi'
c
      integer EHget, GHget, EMfile, GHmake, EHmake
      external EHget, GHget, EMfile, GHmake, EHmake
      integer NO_EVENTS, NO_CHANS, NO_PTS
      parameter (NO_EVENTS=2, NO_CHANS=5, NO_PTS=10)
      integer mode, ierror
      integer npts, event, i, j, k, kk, irec, idat(NO_CHANS)
      character fn*12, buf*40
c
      data GHext,EHext,GHvers(1:5) /3,0,'02.00'/
      data EHdvl1,EHdvl2,EHdvl3,EHtst /1.0,4.0,-100.0,10000000/
c
      do kk=1,GH MAXSTR                    ! fill ext. GH with blanks
        GHstr(kk:kk)= ' '
      enddo
      GHstr(1:22)= 'sample extended header' ! obsolete, don't use!!!
c
      fn = 'test.dat'               ! file for data storage
      GHftyp(1:3) = 'AI8'                  ! ascii integer data
      GHproc(1:3) = 'DAT'                  ! file extension
      GHwrdl = 8                           ! word length: 8 bytes
      GHncha = NO_CHANS                    ! number of channels
c
      mode = 0                      ! dummy value
      event = 0                            ! write next event
      npts = NO_PTS                        ! number of samples
c-
c first, create and write EMERALD type data file
c-
      ierror = EMfile( fn, EMF_CREAT )
      ierror = GHmake( GH_INIT )
      k=0
      do i=1,NO_EVENTS
        EHtsp = EHtst + (npts) - 1
        ierror = EHmake( mode, event, npts )
        do j=1,npts
          k=k+1
          irec=EHrsod+j-1
          write(buf,1000)((k+10000*kk), kk=1,GHncha)
          write(DAunit,rec=irec) buf
        enddo
        EHtst = EHtsp + 1
      enddo
      ierror = GHmake( GH_SET ) !.ne. 128
      ierror = EMfile( fn, EMF_CLOSE )
c
c now, read and display its contents
c
      ierror = EMfile( fn, EMF_EXIST )
      ierror = GHget()
      do i=1, GHnev
        ierror = EHget ( NEXT_EVENT )
        write(*,*)'event no:',i,' data items:',EHrnod
        do j=1,EHrnod
          irec=EHrsod+j-1
```

```
          read(DAunit,rec=irec) buf
          read(buf,1000) (idat(kk),kk=1,GHncha)
          write(*,*) (idat(kk),kk=1,GHncha)
        enddo
      enddo
      ierror = EMfile( fn, EMF_CLOSE )
c
 1000 format(5(i7.7,1x))
      end
```

Test data file

Below is the test data file created with the FORTRAN code above. Please note, the entire file is binary. For the purpose of this documentation all characters are printable. GH and EH are extended with the '_' character to complete the record (40 characters in this example). Extended GH should not be used anymore.

```
0040 AI8 008 02.00 DAT 005 00000029 0004
0002 003 ______________________________
sample extended header ________________
0010000000 00000 0010000009 00000      1
.0000       4.0000      000004 000017 00
0000 000010 000007 000 ________________
0010001 0020001 0030001 0040001 0050001
0010002 0020002 0030002 0040002 0050002
0010003 0020003 0030003 0040003 0050003
0010004 0020004 0030004 0040004 0050004
0010005 0020005 0030005 0040005 0050005
0010006 0020006 0030006 0040006 0050006
0010007 0020007 0030007 0040007 0050007
0010008 0020008 0030008 0040008 0050008
0010009 0020009 0030009 0040009 0050009
0010010 0020010 0030010 0040010 0050010
0010000010 00000 0010000019 00000      1
.0000       4.0000      000017 000030 00
0004 000010 000020 000 ________________
0010011 0020011 0030011 0040011 0050011
0010012 0020012 0030012 0040012 0050012
0010013 0020013 0030013 0040013 0050013
0010014 0020014 0030014 0040014 0050014
0010015 0020015 0030015 0040015 0050015
0010016 0020016 0030016 0040016 0050016
0010017 0020017 0030017 0040017 0050017
0010018 0020018 0030018 0040018 0050018
0010019 0020019 0030019 0040019 0050019
0010020 0020020 0030020 0040020 0050020
```

4.3.4 C Source Code Example

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include ”em file.h”                  /* open,close EMERALD files */
#define GHVERSION ”02.00”             /* gh.version (outdated, actual=5.0) */
/*-------------- EMERALD API functions -----------------*/
int GH_make( int );                   /* create/verify/overwrite GH */
int EH_make( int, long, long );       /* create/verify/overwrite EH */
int EM_file( char *, int );           /* open and close data file */
int GH_add( int, char* );             /* obsolete! add element to EH */
/*------------------ local functions -------------------*/
int DA_rw_testpattern ( void );       /* writes a test pattern as data */
```

```
/* ---------------------- main -----------------------*/

void main( void )
{
  char* fn="test.raw";                    /* filename for time series data */
  long npts = 10L;                        /* length of time series segments */
  long event = 0L;                        /* next event */
  long increment;                         /* sampling rate in 10-6 seconds */
  int i;
  char tmp[80];
/* ----------------- GENERAL HEADER --------------------*/
  strcpy(gh.file type, "BI4");            /* (b)inary (i)nt*(4) data */
  strcpy(gh.processing_ident,"RAW");      /* RAW data */
  strcpy(gh.version, GHVERSION);          /* 02.00 */
  gh.extended = 0;                        /* is modified by EM add */
  gh.word length     = 4;                 /* 4 bytes */
  gh.num channels = 5;                    /* 5 data channels */
/* ----------------- EVENT HEADER ----------------------*/
  eh.data.value1 = 1.0;                   /* sampling rate Hz */
  eh.data.value2 = 2.0;                   /* lowpass cut off */
  eh.data.value3 = -100.0;                /* highpass cut off */
  increment = (long)(eh.data.value1 * 1000000);
  eh.ehtime.start = 10000000L;            /* start time: UNIX style UT */
  eh.ehtime.startms = 500;                /* start time: microseconds */
  EM_file( fn , CREAT);
  strcpy(tmp, "GH;Extended;element;1");   /* should not be used anymore */
  GH_add( DEFAULT MODE, tmp);
  strcpy(tmp, "GH_Extended_element_2");
  GH_add( DEFAULT MODE, tmp);
  GH_make( GH INIT );
  event = 0L;
  for (i=1; i<=2; i++) {
    eh.ehtime.stop = eh.ehtime.start + (eh.ehtime.startms +
                    ((npts-1L)*increment) / 1000000L);
    eh.ehtime.stopms = (eh.ehtime.startms +
                      ((npts-1L)*increment)) % 1000000L;
    EH_make( DEFAULT MODE, event, npts);
    DA_rw_testpattern( );
    printf("event %03d: start %ld :ms %ld -- stop %ld :ms %ld\n",i,
      eh.ehtime.start, eh.ehtime.startms, eh.ehtime.stop,eh.ehtime.stopms);
    eh.ehtime.start = eh.ehtime.stop +
                     ((eh.ehtime.stopms + increment) / 1000000L);
    eh.ehtime.startms = (eh.ehtime.stopms + increment) % 1000000L;
  }
  GH_make( GH_SET );                      /* checking file consistency */
  EM_file( fn, CLOSE );                   /* close data file */
}/* end main */

int DA_rw_testpattern( void )
{
  int i, k;
  long *array, *start; float fa[7];
  fpos_t recpos;

  if (gh.num_channels >= 1000 || gh.num_channels <= 0 ) return(1);
  if (eh.recs.num_of_data >= 10000L || eh.recs.num_of_data <= 0L ) return(2);
  if ( (array = (long *) calloc( (size_t)gh.num_channels, sizeof(long)) )
      == 0) return(3);
  start = array;
  recpos = (fpos_t) (gh.record_length * (eh.recs.start_of_data - 1) );
  if ( fsetpos(dastream, &recpos) != 0) return(4);
```

```
  for (k=1; k <= (int) eh.recs.num_of_data; k++) {
    for (i=1; i <= gh.num_channels; i++) {
      *array++ = (unsigned long) ( ((((k%1000)%100)/10)+48) +
                 (0x00000100 * ((k%10)+48) ) +
                 (0x00010000 * (((i%100)/10)+48) ) +
                 (0x01000000 * ((i%10)+48) ) );
    }
    array = start;
    fwrite( array, sizeof(*array), (size_t) gh.num_channels, dastream);
  }
  free( array );
  return(0);
}
```

Output of exemplary C code

```
0020 BI4 004 02.00 R
AW 005 00000038 0007
 0002 005 _________
GH;Extended;element;
1;GH Extended elemen
t 2; ______________
+010000000 000500 +0
10000509 000500 +000
0000001 +0000000002
-0000000100 000007 0
00023 000000 000010
000013 000 ________
01010102010301040105
02010202020302040205
03010302030303040305
04010402040304040405
05010502050305040505
06010602060306040605
07010702070307040705
08010802080308040805
09010902090309040905
10011002100310041005
+010000510 000500 +0
10001019 000500 +000
0000001 +0000000002
-0000000100 000023 0
00039 000007 000010
000029 000 ________
01010102010301040105
02010202020302040205
03010302030303040305
04010402040304040405
05010502050305040505
06010602060306040605
07010702070307040705
08010802080308040805
09010902090309040905
10011002100310041005
```

ISSN 2190-7110