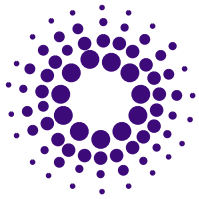
EUROPEAN SYNCHROTRON RADIATION FACILITY
INSTALLATION EUROPEENNE DE RAYONNEMENT SYNCHROTRON



May 20, 1994

The ESRF Data Format

Version 1.1

Peter Daly
Experiments Division Programming Group
European Synchrotron Radiation Facility
38043 GRENOBLE cedex
France

This document describes the ESRF Data Format; a file storage system for the European Synchrotron Radiation Facility.

Copyright Notice

Printed copies of this manual are available (at cost) by writing to:

Experiments Division Programming Group
European Synchrotron Radiation Facility
BP 220,
38043 GRENOBLE cedex
France

Telephone: (+33) 76.88.20.00 Fax: (+33) 76.88.25.42

Terms and Conditions of Distribution

All Rights Reserved

1. This software was written by the Experiments Division Programming Group (EXPG) at the European Synchrotron Radiation Facility (ESRF). It is one of the goals of the ESRF to have the results of work which they support made available to the public and disseminated as widely as possible. This material is being made available to the public in furtherance of this objective.
2. ESRF asks, but does not require, that a notice similar to the one below appears in any software that is developed using this code:

   The File Format libraries used in this software were developed by the European Synchrotron Radiation Facility and are available from them via the Public Domain.

3. As part of ESRF/EXPG's two goals of preserving the free status of all of our free software (and derivatives), and of promoting the sharing and reuse of software generally, you are permitted and encouraged to use, copy, modify, and distribute this software and its documentation for any purpose and without fee. Provided that the a notice to 'ESRF/EXPG' appear in all copies, and that you also do the following:

   a) cause the modified files to carry prominent notices stating that you changed the files and the date of any change

   b) for the whole of any work that you distribute or publish, that contains these routines or any part thereof, either with or without modifications, then any subsequent licence that you charge to all third parties must exclude the ESRF/EXPG software and such ESRF/EXPG software can only be included under the terms of this Conditions of Distribution (except that you may choose to grant warranty protection to some or all third parties, at your option).
4. You may not copy, modify, sublicence, distribute or transfer the software except as expressly provided under either this agreement or as in the case of Software that is in itself written by others and used in the library by the Copyright notices of the original authors. Any attempt otherwise to copy, modify, sublicence, distribute or transfer the Software is void, and will automatically terminate your rights to use the Software under this Agreement.
5. For the software considered as being of ESRF/EXPG, (and for items used that are written by others that do not have an explicit statement of warranty then the following warranty statement also applies) these are supplied under the following conditions.

No Warranty

Because the software is licenced free of charge, there is no warranty for the software, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "**as is**" without warranty of any kind, either expressed or implied, including, but not limited to,

the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

6. For items of software written by others and used by EXPG if the authors have given a statement of copyright and warranties then those notices apply to those items software.
7. You are permitted to copy and or modify this Conditions of Distribution Notice for inclusion in other software.

1.0 Structure and Concepts

1.1 Introduction

This document describes a data format for storage and retrieval of data at the ESRF. The format has the properties described below. It is envisaged that newer versions of the software will be backwardly compatible with older versions.

- it should be flexible enough to cover all types of data collected at the ESRF
- it should allow multiple sets of disparate data
- it should provide as much information about the data itself and any information the user should require with it
- all the calls to read/write the data should be callable from both C and Fortran
- it should be extensible

The basic principle behind the format is that a user who can define his data in a suitable way should be able to obtain that data regardless of the machine or format it is written in. Given the fact that there is no definition of what data actually *is* the format must be flexible enough to cope with different user requirements. The data should be describable in a standard way, if this is possible. For example, a 16-bit image would need the X and Y dimensions, the byte order of the image and it's type to describe the image itself.

1.2 Data Format Definitions

1.2.1 Logical File Structure

The data file has a *logical* structure, which is a view of the data file irrespective of it's physical disk layout, as follows: a ***Global Header Section*** that describes the properties belonging to *all* data within the file and a number of ***Data Blocks***, each with it's own ***Header Section*** that describes properties local to this particular ***Data Section***, these local values will displace any global value in the Global Header Section. That is to say, if a global header section specifies an image size of 512x512 and a particular image in the file specifies 1024x1024 then the

1024x1024 size is used; if an image has no information about it's dimensions then the 512x512 value will be used.

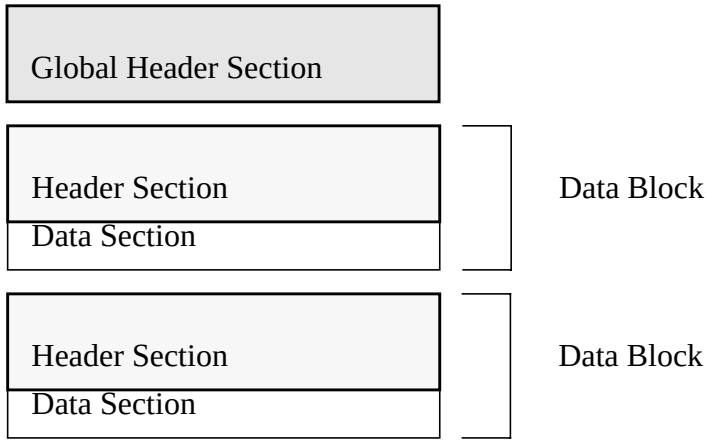


Figure 1: An overview of the data format

In fact there can be more than one global header section providing that the user can identify for himself which global header belongs to which header sections. Files are parsed when opened and the *initial* Global Header Section is the one that contains the VersionNumber keyword (if this does not exist no Global Header Section is defined). All the headers are written in ASCII for readability, and each property in a header is described by a statement of the form

KEYWORD = value ;

the KEYWORD identifies the property, the '=' acts as a separator between the keyword and it's value(s), the semi-colon terminates the statement, thus a statement can extend over several lines, the rest of the line (up to the newline) is ignored and can act as a comment; whitespace is allowed between any of the elements. Blank lines are ignored. Values can consist of single or multiple items (arrays). The significant length of a keyword is currently 64 characters, although the actual length can be longer if the user wishes.

For each logical header it is the user's responsibility to ensure that some keyword (or group of keywords) can be used to identify the header, failure to do this will mean that a particular header may be unavailable for use. For example, images may be stored with Image = 1, Image = 2, etc.

In the data blocks, the Data Section contains *binary*[1] data described in it's header. The Data Section may be empty (i.e. no binary data) but if it does exist it *must* immediately follow the Header Section. Within each Data Section the data may be compressed with a choice of available techniques, random access to any Data Section is allowed. Whitespace is allowed between data blocks to align the start of a block on a convenient boundary (if appropriate).

It should be noted that the format does not require any particular order to the data blocks within the file and implies no particular size of storage element. The Data Sections are stored in whatever size storage element is required by the user, typical values will be Short Integers (16 bit data), Integers (32 bit data), Real Values (IEEE) and some can also be defined as signed or unsigned as required.

Furthermore, there is no requirement that a Data Section read from a file and returned as, for example, REAL is actually stored in the file in the same format that is required. This implies that conversion between the different storage elements takes place.

When reading or writing a piece of data the user may choose to identify the correct place to put it either by performing a search of the headers or by specifying a header number. For example, if we wish to read the title of image 3 then we must specify what we want (i.e. the title) and how to find that information - we can either specify the search condition is ‘Image = 3’ or (if we know which header this information is to be found) by specifying the header number for this item. However, user’s may not always know this value an cannot assume a particular value for a header since the system itself assigns integers as necessary. In general, the search condition ‘Image = 3’ would be used.

1.2.2 Physical File Structure

Although the logical structure is all one needs to know at the user level, this section describes the technique of writing information to the file and the reasons for the choice of header format. Furthermore, users can write data files in this format, simply with print statements, provided they remain faithful to the header identifier. At the disk level the structure of the file follows a tagged list; each header has an identity number of the form:

HeaderID = EH:000002:000003:000001;

the numbers are, respectively, the current header number; the header number of the next portion of the header and the header number of the previous portion of the header (the EH stands for ESRF Header and is purely arbitrary). A value of *zero* in the next header field shows that this header is the last in this group and a *zero* in the previous field shows that this is the first portion of the group.

This technique is used rather than byte offsets from the start of the file so that the headers themselves are *editable* by an editor such as *emacs*. Fixed length fields are used (for this statement *only*) since each header *must* have an identifier and variable length identifiers would create problems with overflow. Each header may contain the next strip of data in the binary section. Also in each

1. binary here means that the space between headers is treated as a block of data that the user can interpret at will. For some common storage requirements (e.g. unsigned short integer images) the data will be written in a standard way so the user need not bother about the details, but in some cases the user will simply receive a block of data with no particular emphasis placed on it.

header is a *Size* keyword to describe the physical size of the data stored. Both the *HeaderID* and the *Size* keyword are managed by the software and should *not be changed by the user*.

This also implies that a file may have X physical headers but Y logical headers (where Y <= X) since there may be several physical headers to one logical header. As an example, imagine we have the following HeaderID's in a file:

HeaderID = EH:000001:000000:000000 ;

HeaderID = EH:000002:000005:000000 ;

HeaderID = EH:000003:000000:000000 ;

HeaderID = EH:000004:000000:000005 ;

HeaderID = EH:000005:000004:000002 ;

We can see that although there are 5 physical headers in the file, logically there are only 3, since headers 2, 4 and 5 are part of the same logical header. In this case header 2 is (in sequence) made up of headers 2, 5 and 4. The management of this logical versus physical view is provided by the software.

This physical structure allows the user to read/write arbitrary part of a data section and/or header without the penalty of file reorganisation. When new header information is to be written the software caches it in memory until the file is closed to prevent unnecessary fragmentation of the headers - users can explicitly flush the headers with a FlushTable command in the Write Data routine.

1.3 Header Section Structure

The header sections of the data file are described by ASCII statements delimited by the curly braces '{' and '}'. These braces act as start-of-header and end-of-header markers. The keywords chosen are case *in*sensitive, but the *value* will have it's case preserved.

It is also possible to have a *site file* where various properties can be defined for a particular site. For example, at the ESRF the site file may contain the definitions of the instruments in some sense meaningful to the experiment. If this file is delimited with the curly braces mentioned above then it will be parsable by the same routines that read the data file.

1.3.1 Example of a Header

Below is an example of a header.

```
{
; This is an example header. Comments in the header are ignored as are blank lines.
HeaderID = EH:000002:000006:000003          ; this Identifier should not be changed by hand
```

```
VersionNumber = 1.0                     ; version number of software
ByteOrder = LowByteFirst                ; Little ended data
DataType = UnsignedShortInteger         ; 2 bytes per pixel
Size = 2817492                          ; Compressed image size in bytes
Dim_1= 1187                             ; X-pixels
Dim_2 = 1187                            ; Y-pixels
Compression = RunLengthEncoded          ; compression method used
DetectorName = MAR Scanner              ; Detector used
Title = “TEST IMAGE PLATE DATA”         ; self-explanatory
Cell = 105.77 105.77 153.37 90 90 120   ; Cell a b c alpha beta gamma
SampleName = “Unknown”                  ; self-explanatory
ProposalNumber =                        ; this value is issued by the ESRF
}
```

The example shows the following points:

- this header is number 2, the next logical portion of the header section is number 6 and the previous portion of the header was number 3, thus there are many physical headers to this logical one. The other portions of this logical header would have other suitable keywords and descriptions required by the user.
- comment lines are allowed by prefacing with a semi-colon. The comment extends up to the newline blank lines in the header are ignored,
- some symbolic names are pre-defined e.g. LowByteFirst and HighByteFirst describe the way that integers are stored (the ‘endedness’ of the machine),
- multiple values (arrays) are allowed (e.g. the Cell statement below),
- some keywords can be initialised but have no value defined until a later time (e.g. the ProposalNumber keyword),

the image in this file is 1187x1187 pixels and is compressed - the Size keyword records the size of this section of the compressed image

1.4 Pre-defined Names for common keywords

The table in Appendix 2 describes the pre-defined keywords for some common statements. Some keywords are required to describe a particular Data Section whilst others are highly recommended. For example, the HeaderID is *required*, but the ByteOrder keyword is not required (since it will default to the machine type you are using) but is *highly recommended* so that Data Sections can be swapped between different machines with no loss of generality. The *Size* keyword is compulsory if a header has a Data Section.

1.5 Access to Data Sections

Since the Data Sections may contain any data the user must define how he wishes to identify the Data Sections - access is object-oriented. The user must define the keywords, their values and their types in order to choose a particular

Data Section. A Data Section *may* need more than one keyword to access it (e.g. imagine a user who wants an image described by Image = 1 and Date = some date).

In fact, the Data Sections may contain disparate types of data and the user can choose which keys to access those sections he wants. For example, a data file may contain Data Sections that contain calibration information, which the user will identify by the keyword CalibrationScan (with some integer value) and image data that he wishes to identify by the keyword Image (and a corresponding integer to specify it).

1.5.1 Sub-sections of Data

It can be seen from the physical file layout that there is no requirement for all the data of, for example, an image be recorded in one Data Section. The data can be divided up into strips (of different sizes) and that each strip can be compressed with a different compression technique. This is managed by the software, however, the user must specify the strip number so that data is not overwritten accidentally.

Furthermore, the interfaces below provide for the user to specify a sub-portion of the data rather than obtain the whole block - again, imagine an image where we only want a small portion of it.

1.5.2 Defined data types for data stored

The keyword DataType is used to describe the data in the data section. It can take any of the symbolic shown in the table below. In the case of ComplexByte, ComplexShortInteger, ComplexInteger and ComplexLongInteger the values are treated as signed.

DataType descriptors

SignedByte	UnsignedByte	ComplexByte
SignedShort	UnsignedShort	ComplexShortInteger
SignedInteger	UnsignedInteger	ComplexInteger
SignedLong	UnsignedLong	ComplexLongInteger
FloatValue		ComplexFloat
DoubleValue		ComplexDouble

1.5.3 Defined types for keywords

When a user wishes to read a header item or search for a header he must specify 3 items of information: the keyword, it's value and it's type. For example, to find a data section described by the statement

Image = 1;

the user would need to specify the keyword “image”, the value (in this case 1) and it’s type for a suitable comparison to be made - currently, the types can be IntegerValue, FloatValue or StringValue. We call these keyword descriptors. In the example above the type would, of course, be IntegerValue.

1.6 Interfaces to Manipulate the Data file

Below are the interface routines to manipulate the data file. In general the routines themselves will return the (integer) symbolic value RoutineSucceeded for success or RoutineFailed for failure. Most routines to access the data have three arguments:

- a channel number to specify which file to use, this is an integer to a (private) array - each element of this array is a structure that records various properties of the file opened,
- a control block structure to set values and return results. This is currently a block with 6 values. We will look at several different cases below,
- a returned integer error value to describe success or failure of the routine. This value gives a more explicit description of the reason the symbolic name RoutineFailed was returned. A routine is provided so that the user can be given a description of the error.

Various flag values have been defined to make the use of the library easier. They are explicitly listed below - see the appropriate call definition for use of these flags:

Flag	Used for
ReadData	Specifies Data Section is to be read
ReadHeader	Specifies a Header Section is to be read
InquireData	Specify that we want to know the size and type of the data without the overhead of reading it from the disk
WriteData	Specifies Data Section is to be written
WriteHeader	Specifies a Header Section is to be written
SearchCondition	Tells the interface then argument 5 is a block of conditions rather than a simple header number. This allows searches for different matches and composite keys
FlushTable	Explicitly flush the buffers rather than cache some headers until the files are closed. Can cause fragmentation of the headers.
AllocateSpace	will cause the library to use malloc to create the necessary space. If this is omitted the routines will take the start address as the target for the data and the sizes as inputs.

2.0 Compression

2.1 Introduction

It is envisaged that different compression techniques will be made available to the user using a variety of techniques so that the size of the file can be reduced as much as possible. It goes without saying, however, that users who wish to use this facility do so with the penalty of their data taking longer to read/write.

2.2 Diffraction Image Data Compression

The Diffraction Image Data Compression technique - using the symbolic name DiffDataCompress has been donated by Jan Peter Abrahams, MRC, Laboratory of Molecular Biology, Hills Road, Cambridge. The description given below is from comments in the code itself.

This file contains functions capable of compressing and decompressing images. It is especially suited for X-ray diffraction patterns, or other image formats in which orthogonal pixels contain “grey-levels” and vary smoothly across the image. Clean images measured by a MAR-research image plate scanner containing two bytes per pixel can be compressed by a factor of 3.5 to 4.5.

Since the images are encoded in a byte-stream, there should be no problem concerning big- or little ended machines: both will produce an identical packed image.

Compression is achieved by first calculating the differences between every pixel and the truncated value of four of its neighbours. For example:

the difference for a pixel at img[x, y] is:

img[x, y] - (int) (img[x-1, y-1] + img[x-1, y] + img[x-1, y+1] + img[x, y-1]) / 4

After calculating the differences, they are encoded in a packed array. A packed array consists of consecutive chunks which have the following format:

- Three bits containing the logarithm base 2 of the number of pixels encoded in the chunk.
- Three bits defining the number of bits used to encode one element of the chunk. The value of these three bits is used as index in a lookup table to get the actual number of bits of the elements of the chunk.
- The truncated pixel differences.

3.0 The C Interface

3.1 Introduction

In the following descriptions, each function is described using normal C syntax. The return type is given in the definition and then the functions arguments are described as INPUT (input only), OUTPUT (output only) or INOUT (both input and output). There then follows a brief description of the purpose of the call.

In the C interface a pre-defined typedef is provided called ArgBlk. This can be used to generate the Structure argument used by the calls (see the C examples section). This is currently defined as:

```
typedef struct ArgumentBlock {
int   ConvertType,               /* what data type */
      ConvertFlag;               /* what to do */
void  *StartData,                /* memory location for data/header */
      *Sizes,                    /* sizes of the data or header value */
      *Offsets,                  /* sub-section of data or header type */
      *ControlBlock              /* search condition or header number */
} ArgBlk;
```

As an example, to fill in this Structure to Read the data defined by Image = 1; the user would write the following code. Note that we use a search condition to find the data:

```
int *Block[4], ImageNumber = 1;
ArgBlk *Args = (ArgBlk *) malloc (sizeof (argBlk));
                                 /* set up the search condition */
Block[0] = 1;                    /* 1 condition */
Block[1] = (int *) “image”;      /* Keyword */
Block[2] = (int *) &ImageNumber; /* Value */
Block[3] = IntegerValue;         /* type */
                                 /* now set up the argument structure */
Args->ConvertType = UnsignedShort;/* return data type */
/* use malloc and search for data by using a condition */
Args->ConvertFlags = ReadData & AllocateSpace & SearchCondition;
Args->StartData = (void *) NULL; /* start address returned here */
Args->Sizes = (void *) NULL;     /* sizes returned here */
Args->Offsets = (void *) NULL;   /* no sub-image required */
Args->ControlBlock = (void *) Block;/* pointer to search conditions */
```

3.2 int ExpgOpenDataFile (char *Filename, int *DataErrorValue)

INPUT:	char *Filename
OUTPUT:	int *DataErrorValue
INOUT:	None
FUNCTION:	open the Data File

This routine will open up the appropriate file and return a channel number (an integer provided by the calls, which is an index into a private data structure - users should not try to access this structure directly).

In the event of a failure the channel number will be the value of the Symbolic name RoutineFailed.

The Filename is first checked as an environment variable; otherwise it is taken as a literal pathname. This allows the user to specify whatever filename he chooses outside the program, if he so wishes. For example, the call:

```
int stream, ErrorValue;
stream = ExpgOpenDataFIle (“IMAGE1”, &ErrorValue)
```

will look for an environment variable IMAGE1 and use it’s value. If this does not exist then the file opened will be IMAGE1. In the ExpgOpenDataFile routine the opening of the file causes all the headers to be read and kept in a private data structure in memory. The maximum number of streams that can be open at any time is (arbitrarily) the symbolic name MaxFiles. In all the following calls int Stream refers to the value returned by these calls.

The routine will also use a simple test for an ESRF Data Format File in that the first non-blank element must be an open curly bracket - if it finds anything else it will assume the file is not in the ESRF format and return with DataErrorValue set to the appropriate error number.

3.3 int ExpgCloseDataFile (int Stream, int *DataErrorValue)

INPUT:	int Stream
OUTPUT:	int *DataErrorValue
INOUT:	None
FUNCTION:	close the Data File

This routine will close the appropriate file and free the channel for other calls. All the data structures created by a call to ExpgOpenDataFile with this stream number are freed.

In the event of failure, the user should check the possible reasons for this in his code and should **not** use the freed channel number again since this may lead to corruption of the file(s).

Note: if the file has been *updated* or has had new information *written* to it then this call *must* be used to ensure the buffers are flushed to it because this call searches the internal tables to find headers that have not been written to the file and writes them out. If you only *read* from a file then this call is not explicitly needed but it is good practice to include a close with each open statement you use.

3.4 int ExpgReadDataFile (int Stream, void *Structure, int *DataErrorValue)

INPUT:	int Stream
OUTPUT:	int *DataErrorValue
INOUT:	void *Structure
FUNCTION:	performs various read operations on the data.

The Structure (defined in the introduction above) is used to pass parameters in and out of the routine. See the examples in the next section.

If you open a data file and *only* make read operations on it then you do not need to explicitly call ExpgCloseDataFile, since the file itself will not have changed.

3.4.1 Example

To fill in this Structure to Read a title from the data block referenced by Image = 1; the user would write the following code. Note that we use a search condition to find the data:

```
int *Block[4], ImageNumber = 1, ErrorValue, ReadError;
ArgBlk *Args = (ArgBlk *) malloc (sizeof (argBlk));
                                /* set up the search condition */
Block[0] = 1;                   /* 1 condition */
Block[1] = (int *) “image”;     /* Keyword */
Block[2] = (int *) &ImageNumber;  /* Value */
Block[3] = IntegerValue;        /* type */
                                /* now set up the argument structure */
Args->ConvertType = NoSpecificValue;/* unused for ReadHeader */
Args->ConvertFlags = ReadHeader & SearchCondition;/* use malloc and
search for data */
Args->StartData = (void *) “title”;/* Keyword to read*/
Args->Sizes = (void *) NULL;    /* value returned here */
Args->Offsets = (void *) StringValue;/* type */
Args->ControlBlock = (void *) Block;/* pointer to search conditions */
ErrorValue = ExpgReadDataFile (Stream, Args, &ReadError);/* perform the
read */
```

3.5 int ExpgWriteDataFile (int Stream, void *Structure, int *DataErrorValue)

INPUT:	int Stream
OUTPUT:	int *DataErrorValue
INOUT:	void *Structure
FUNCTION:	performs various write operations on the file.

Writing *data* to the file always causes the data to be written immediately since the libraries have no way of knowing how long this data may exist.

Writing *header information* does not always cause an explicit write to the file (they will be cached in memory) to improve performance and help prevent fragmentation of the file. This header information will only be written either by using the FlushTable command or by closing the file.

If you write header information to the data file, or add new header information then you **must** explicitly call ExpgCloseDataFile to flush the internal tables - failure to do so will mean that the new information is not reflected in the file.

3.5.1 Example

To fill in this Structure to Write an image (with value identified by an integer), the user would write the following code. Note that we use a search condition to find where to write the data, we also assume a stream is already open and the data exists:

```
int *Block[4], ImageNumber = 1, Sizes[3], ErrorValue, WriteError;
ArgBlk *Args;
unsigned short int *Data;
Sizes[0] = 2;                    /* Number of dimensions */
Sizes[1] = 512;                  /* X Dimension */
Sizes[2] = 256;                  /* Y dimension */
                                 /* set up the search condition */
Block[0] = 1;                    /* 1 condition */
Block[1] = (int *) “image”;      /* Keyword */
Block[2] = (int *) &ImageNumber; /* Value */
Block[3] = IntegerValue;         /* type */
                                 /* now set up the argument structure */
Args = (ArgBlk *) malloc (sizeof (argBlk));/* allocate the space */
Args->ConvertType = UnsignedShort;/* return data as this type */
Args->ConvertFlags = WriteData & SearchCondition;/* use malloc and search
for data */
Args->StartData = (void *) Data;  /* start address returned here */
Args->Sizes = (void *) &Sizes;    /* sizes returned here */
Args->Offsets = (void *) NULL;    /* no sub-image required */
Args->ControlBlock = (void *) Block;/* pointer to search conditions */
ErrorValue = ExpgWriteDataFile (Stream, Args, &WriteError);/* write the
data */
```

3.6 char *ExpgDataFormatVersion (void)

INPUT: None
OUTPUT: None
INOUT: None
FUNCTION: returns a string value for the current version of the software.

3.7 void ExpgCatchInterrupt (Flag)

INPUT: int Flag
OUTPUT: None
INOUT: None
FUNCTION: Toggles the interrupt catch facility on or off. If the Flag is True then the signal SIGINT (usually ^C) is caught and all files are closed before any other action. If the Flag is False then the old signal handler (which may be no action) is re-established and no flushing of headers is performed. The default start-up action is OFF.

3.8 char *ExpgReportStatus (int ErrorValue, int DataErrorValue, int DisplayFlag)

INPUT: int ErrorValue
int DataErrorValue
int DisplayFlag

OUTPUT: None
INOUT: None
FUNCTION: give a report on the error that occurred.

The return value is a pointer to a string that contains the Error Message Text.

The value **ErrorValue** is the return value of the routine (generally RoutineFailed), **DataErrorValue** is the returned error code from the routine that the user wishes to report on and **DisplayFlag** is set to True if the user wishes the routine to output the message or False if the routine does not output the message - this can be used, for example, by windows software where the programmer wishes to obtain the string and display it in it's own window.

A typical piece of code may look like (Note that ErrorValue will invariable be RoutineFailed and the real error number is held in DataErrorValue):

```
int ErrorValue, DataErrorValue;
if ((ErrorValue = ExpgOpenDataFile (“myfile”, &DataErrorvalue)) !=
RoutineSucceeded)
      (void) ExpgReportStatus (ErrorValue, DataErrorValue, True);
```

3.9 int ExpgDisplayHeaders (int Stream, int *DataErrorValue)

INPUT: int Stream
OUTPUT: int *DataErrorValue
INOUT: None
FUNCTION: this routine the will display all the headers currently in memory to the standard output device.

3.10 int ExpgGetByteOrder (void)

INPUT: None
OUTPUT: None
INOUT: None
FUNCTION: determine ‘endedness’ of your machine

A support routine to determine the byte order of your machine. Return Values are HighByteFirst (True) or LowByteFirst (False).

This routine describes the way in which integers are stored in the data (either 1234 or 4321).

4.0 C Examples

4.1 Writing Data Blocks

To write data blocks the control structure has all elements as inputs and are used as follows:

Element	Used for	Values
ConvertType	Output Conversion Type	One of the types defined above e.g. UnsignedInteger
ConvertFlags	Control Flags	WriteData AND any of: SearchCondition, JPEG, JPEG_Lossy, RunLengthEncoded, DiffDataCompress
StartData	Address of Start of Data	Start of data in memory
Sizes	Address of array of dimensions	element 0 is number of dimensions, then the sizes themselves
Offsets	Address of an array of offsets to get sub-sections of data	array is 2 * N in size (where N is number of dimensions)
ControlBlock	Address of a control block to specify which data to write or a header number	if SearchCondition is not specified this value may be replaced by a header number

Where:

- Output Conversion Type describes the format of the data as it is to be written to the file, it is one of the DataType descriptors show in the table in section 5.2
- JPEG, JPEG_Lossy, RunLengthEncoded, DiffDataCompress are different compression techniques to apply to the data,
- SearchCondition specifies whether the headers need to be searched to find the correct place to put the data. If present the last argument is a pointer to an array of the form where element 0 is the number of conditions and then we have 3 elements per condition to specify the keyword, it's value and it's type (IntegerValue, StringValue or FloatValue), if SearchCondition is *not* present this last element can contain the header number itself.

4.1.1 Example

Below is a subroutine to write out an image from memory to the data file. We do not show how the image is generated. The parameters of the call are the stream to write the data to (previously opened by a call to ExpgOpenDataFile), the image number (we identify images as Image = 1, Image = 2, etc.), the dimen-

sions of the image, it's type (e.g. UnsignedShort) and it's start address in memory. A typical call may look like this:

```
WriteImage (stream, 1, 512, 768, UnsignedShort, start) /* write out image
1 */
#include "esrf_data_format.h"
int WriteImage (int stream, int Number, int X, int Y, int Type, void
*start)
{
     int i, Sizes[3], *CtrlBlock[4], Count, ErrorValue;
     long Size;
     ArgBlk Arguments;

     Sizes[0] = 2;              /* set up the dimensions */
     Sizes[1] = X;
     Sizes[2] = Y;
     Size = X * Y * DataSize ((void *) Type, True)

     Arguments.ConvertType = Type;/* assign the Output Data Type */
     Arguments.ConvertFlags = WriteData & SearchCondition;/* what we
want to do */
     Arguments.StartData = (void *) start;/* start position of the data
*/
     Arguments.Sizes = (void *) &Sizes;/* the dimensions of the data */
     Arguments.Offsets = (void *) NULL;/* not needed */
     Arguments.ControlBlock = (void *) CtrlBlock;/* specify the search
array */

     Count = 1;
     CtrlBlock[0] = (int *) Count;/* number of conditions */
     CtrlBlock[1] = (int *) "Image";/* Key word */
     CtrlBlock[2] = (int *) &Number;/* Value */
     CtrlBlock[3] = (int *) IntegerValue;/* Type */

     i = ExpgWriteDataFile (stream, (int *) Arguments, &ErrorValue);
     if (i != RoutineSucceeded) (void) ExpgReportStatus (i, ErrorValue,
True);
     return (i);
} /* End of WriteImage */
```

4.2 Writing Header Information

To write header information the control structure is, again, all inputs and has the following fields:

Element	Used for	Values
ConvertType	Currently Unused	
ConvertFlags	Control Flags	WriteHeader (AND SearchCondition if necessary)
StartData	Address of Start of Data	pointer to the Keyword
Sizes	Address of array of dimensions	pointer to its value

Offsets	Type of Data	IntegerValue, FloatValue or StringValue
ControlBlock	Address of a control block to specify which data to write or a header number	if SearchCondition is not specified this value may be replaced by a header number

4.2.1 Example

Below is a section of code to write out an image from memory to the data file. We do not show how the image is generated. Note that in this example we wish to describe the image with the keyword Image = 1; this means that we specify a search condition (of Image = 1). The example assumes that we have already opened a file using ExpgOpenDatFile and the channel number is returned in the integer stream:

```
#include “esrf_data_format.h”
int i, stream, ImageNumber, *CtrlBlock[4], Sizes[3], ErrorValue;
short *data;                /* pointer to the data */
ArgBlk Arguments;           /* define our argument block */

ImageNumber= 1;             /* j records the image number */
Sizes[0] = 2;               /* number of dimensions */
Sizes[1] = 1152;            /* x dimension */
Sizes[2] = 1482             /* y dimension */

CtrlBlock[0] = (int *) 1;   /* number of search conditions */
CtrlBlock[1] = (int *) “Image”;/* keyword */
CtrlBlock[2] = (int *) &ImageNumber;/* value */
CtrlBlock[3] = (int *) IntegerValue;/* type */

Arguments.ConvertType = UnsignedShort;/* output type */
Arguments.ConvertFlags = WriteHeader & SearchCondition;
Arguments.StartData = (void *) data;/* where the data starts */
Arguments.Sizes = (void *) Sizes;/* the sizes */
Arguments.Offsets = (void *) NULL;/* not used in this case */
Arguments.ControlBlock = (void *) CtrlBlock;/* the search block */

i = ExpgWriteDataFile (stream, (int *) Arguments, &ErrorValue);
if (i != RoutineSucceeded) (void) ExpgReportStatus (i, ErrorValue,
True);
```

4.3 Reading Data Blocks

To read data blocks the control structure can have both inputs and outputs according to the various flags - the meaning of the fields are:

Element	Used for	Values
ConvertType	Input Conversion Type	One of the types defined above e.g. UnsignedInteger

ConvertFlags	Control Flags	ReadData AND any of: SearchCondition, AllocateSpace
StartData	Address of Start of Data	Start of data in memory (see below)
Sizes	Address of array of dimensions	element 0 is number of dimensions, then the sizes themselves
Offsets	Address of an array of offsets to get sub-sections of data	array is 2 * N in size (where N is number of dimensions)
ControlBlock	Address of a control block to specify which data to write or a header number	if SearchCondition is not specified this value may be replaced by a header number

Where:

- AllocateSpace is used to tell the routines whether or not space needs to be allocated by the routines themselves. If the flag is present the routines will allocate a suitable space and fill it with data - element 2 will be the *returned* address of this block (i.e. an output) as will element 3. If AllocateSpace is *not* present then the routines will assume that space has been allocated already and will use elements 2 and 3 as inputs to tell the routine where to place the data in memory.

4.3.1 Example

Below is a section of code to write out an image from memory to the data file. We do not show how the image is generated. Note that in this example we wish to describe the image with the keyword Image = 1; this means that we specify a search condition (of Image = 1). The example assumes that we have already opened a file using ExpgOpenDataFile and the channel number is returned in the integer stream:

```
     #include “esrf_data_format.h”
     int i, stream, ImageNumber, *CtrlBlock[4], Sizes[3], ErrorValue;
     short *data;                 /* pointer to the data */
     ArgBlk Arguments;           /* define our argument block */
     ImageNumber= 1;             /* j records the image number */
     Sizes[0] = 2;               /* number of dimensions */
     Sizes[1] = 1152;            /* x dimension */
     Sizes[2] = 1482             /* y dimension */
     CtrlBlock[0] = (int *) 1;   /* number of search conditions */
     CtrlBlock[1] = (int *) “Image”;/* keyword to search for */
     CtrlBlock[2] = (int *) &ImageNumber;/* value of the image number */
     CtrlBlock[3] = (int *) IntegerValue;/* its type for the compare rou-
tine */
     Arguments.ConvertType = UnsignedShort;/* Data to be written out as
this type */
     Arguments.ConvertFlags = ReadData & SearchCondition;
     Arguments.StartData = (void *) data;/* where the data starts */
     Arguments.Sizes = (void *) Sizes;/* the sizes */
     Arguments.Offsets = (void *) NULL;/* not used in this case */
     Arguments.ControlBlock = (void *) CtrlBlock;/* the search block */
```

```
      i = ExpgReadDataFile (stream, (int *) Arguments, &ErrorValue);
      if (i != RoutineSucceeded) (void) ExpgReportStatus (i, ErrorValue,
True);
```

4.4 Inquiring about Data Blocks

It is possible to read information about a data block without actually reading the data. That is we can inquire about the various sizes and dimensions of the data. In this case the fields are:

Element	Used for	Values
ConvertType	Output Data Type	Returned type of data
ConvertFlags	Control Flags	InquireData (AND SearchCondition if necessary)
StartData	Currently Unused	
Sizes	Address of array of dimensions	an array with a value for each dimension (see below)
Offsets	Currently Unused	
ControlBlock	Address of a control block to specify which data to write or a header number	if SearchCondition is not specified this value may be replaced by a header number

Where:

- element 3 has N + 2 parts (where N is the number of dimensions found). the first element is the number of dimensions, then the dimensions themselves and, finally, the size of the data in this header.

4.4.1 Example

Below is a section of code to find the details of an image specified by Image = 1;

```
      #include “esrf_data_format.h”
      int i, stream, ImageNumber, *CtrlBlock[4], *Sizes, ErrorValue;
      ArgBlk Arguments;           /* define our argument block */
      ImageNumber= 1;             /* j records the image number */
      CtrlBlock[0] = (int *) 1;   /* number of search conditions */
      CtrlBlock[1] = (int *) “Image”;/* keyword to search for */
      CtrlBlock[2] = (int *) &ImageNumber;/* value of the image number */
      CtrlBlock[3] = (int *) IntegerValue;/* its type for the compare rou-
tine */
      Arguments.ConvertType = NoSpecificValue;/* the data type returned
here */
      Arguments.ConvertFlags = InquireData & SearchCondition;
      Arguments.StartData = (void *) NULL;/* unused */
      Arguments.Sizes = (void *) NULL;/* sizes returned here */
      Arguments.Offsets = (void *) NULL;/* unused */
      Arguments.ControlBlock = (void *) CtrlBlock;/* the search block */
      i = ExpgReadDataFile (stream, (int *) Arguments, &ErrorValue);
```

```
     if (i != RoutineSucceeded) (void) ExpgReportStatus (i, ErrorValue,
True);
     Sizes = (int *) Arguments->Sizes/* Values we want */
```

4.5 Reading Header Information

To read header information the control structure has the fields below, Elements 2 and 3 give the result required:

Element	Used for	Values
ConvertType	Currently Unused	
ConvertType	Control Flags	ReadHeader (AND SearchCondition if necessary)
StartData	Address of Keyword	pointer to the Keyword
Sizes	Address of value	returned pointer to its value
Offsets	Count	returned count of number of matching headers
ControlBlock	Address of a control block to specify which data to write or a header number	if SearchCondition is not specified this value may be replaced by a header number

4.5.1 Example

Below is a section of code to find the title of an image specified by Image = 1;

```
     #include “esrf_data_format.h”
     int i, stream, ImageNumber, *CtrlBlock[4], ErrorValue;
     ArgBlk Arguments;           /* define our argument block */
     ImageNumber= 1;             /* j records the image number */
     CtrlBlock[0] = (int *) 1;   /* number of search conditions */
     CtrlBlock[1] = (int *) “Image”;/* keyword to search for */
     CtrlBlock[2] = (int *) &ImageNumber;/* value of the image number */
     CtrlBlock[3] = (int *) IntegerValue;/* its type for the compare rou-
tine */
     Arguments.ConvertType = NoSpecificValue;/* the data type returned
here */
     Arguments.ConvertFlags = ReadHeader & SearchCondition;
     Arguments.StartData = (void *) “title” ;/* keyword */
     Arguments.Sizes = (void *) NULL;/* value returned here */
     Arguments.Offsets = (void *) StringValue;/* type*/
     Arguments.ControlBlock = (void *) CtrlBlock;/* the search block */
     i = ExpgReadDataFile (stream, (int *) Arguments, &ErrorValue);
     if (i != RoutineSucceeded) (void) ExpgReportStatus (i, ErrorValue,
True);
```

5.0 The Fortran Interface

5.1 Introduction

Warning: the Fortran interface does not have access to the AllocateSpace option in the Control Block because Fortran 77 does not have dynamic memory allocation. It is the user's responsibility to ensure that data returned by the routines (including strings) is placed in a suitably dimensioned array - failure to reserve sufficient space will cause the program to generate memory overflow errors and the program will crash.

The Fortran interface is under development and uses an INCLUDE file called esrf_data_format.inc. This file defines various values that can be used for a replacement of symbolic names. It also defines the functions below. Each routine has two names; the long, more descriptive name and (for those who insist on absolute Fortran 77) a shorter name, conforming to the Fortran 77 rules under the sub-heading ALTERNATIVE; this may be important for compilers who will not allow long names or for those machines where name space conflicts may occur - see the note for HP users below.

Note for those who need to compile the library: if you wish to install the Fortran interface when you compile the library it uses a package called cfortran.h written by Burkhard Burrow - this is **not** part of the software and users will have to obtain the package by anonymous ftp from zebra.desy.de (internet number: 131.169.2.244). Installation of this package is straightforward and should use the ANSI C implementation.

Note also that since strings may be merged into the data section (at the discretion of the compiler) users should explicitly terminate their strings with a null character i.e. use MYSTRING//CHAR(0) to ensure the C routines underneath correctly obtain the length of the string.

HP users: In the interface itself the long name has been compiled with the +ppu option and requires that the user also compiles his program with this option. If this causes a problem (e.g. calling other libraries may generated undefined symbol errors from the loader) then use the shorter name that has **not** been compiled with this problem

In the Fortran interface a six element integer array provides a structure to pass various arguments to the routines. Some examples are given with the routines others follow in the next section.

5.2 INTEGER FUNCTION ExpgOpenDataFile (Filename, DataErrorValue)

ALTERNATIVE:	INTEGER FUNCTION EXPODF (Filename, DataErrorValue)
INPUT:	CHARACTER *(*) Filename
OUTPUT:	INTEGER DataErrorValue
INOUT:	None
FUNCTION:	open the Data File

This routine will open up the appropriate file and return a channel number (an integer provided by the calls- this stream number should **not** be confused with a Fortran Unit number and it is not interchangeable with it.).

In the event of a failure the channel number will be the value of the Symbolic name RoutineFailed.

The Filename is first checked as an environment variable; otherwise it is taken as a literal pathname. This allows the user to specify whatever filename he chooses outside the program, if he so wishes. For example, the call:

```
include 'esrf_data_format.inc'
integer stream, ErrorValue;
stream = ExpgOpenDataFIle ('IMAGE', ErrorValue)
```

will look for an environment variable IMAGE1 and use it's value. If this does not exist then the file opened will be IMAGE1. In the ExpgOpenDataFile routine the opening of the file causes all the headers to be read and kept in a private data structure in memory. The maximum number of streams that can be open at any time is (arbitrarily) the symbolic name MaxFiles. In all the following calls int Stream refers to the value returned by these calls.

The routine will also use a simple test for an ESRF Data Format File in that the first non-blank element must be an open curly bracket - if it finds anything else it will assume the file is not in the ESRF format and return with DataErrorValue set to the appropriate error number.

5.3 INTEGER FUNCTION ExpgCloseDataFile (Stream, DataErrorValue)

ALTERNATIVE:	INTEGER FUNCTION EXPCDF (Stream, DataErrorValue)
INPUT:	INTEGER Stream
OUTPUT:	INTEGER DataErrorValue
INOUT:	None
FUNCTION:	close the Data File

This routine will close the appropriate file and free the channel for other calls. All the data structures created by a call to ExpgOpenDataFile with this stream number are freed.

In the event of failure, the user should check the possible reasons for this in his code and should **not** use the freed channel number again since this may lead to corruption of the file(s).

Note: if the file has been *updated* or has had new information *written* to it then this call *must* be used to ensure the buffers are flushed to it because this call searches the internal tables to find headers that have not been written to the file and writes them out. If you only *read* from a file then this call is not explicitly needed but it is good practice to include a close with each open statement you use.

5.4 INTEGER FUNCTION ExpgReadDataFile (Stream, Structure, DataErrorValue)

ALTERNATIVE: INTEGER FUNCTION EXPRDF (Stream, Structure, DataErrorValue)
INPUT: INTEGER Stream
OUTPUT: INTEGER *DataErrorValue
INOUT: INTEGER Structure[6]
FUNCTION: performs various read operations on the data.

The Structure (defined in the introduction above) is used to pass parameters in and out of the routine. See the examples in the next section.

If you open a data file and *only* make read operations on it then you do not need to explicitly call ExpgCloseDataFile, since the file itself will not have changed.

5.4.1 Example

To fill in this Structure to Read a title from the data block referenced by Image = 1; the user would write the following code. Note that we use a search condition to find the data:

```
*
      INCLUDE 'esrf_data_format.inc'
*
      INTEGER   ISTREAM, IRETERR, IRDERR, INUM, IBLOCK(6), ICOND(40)
      CHARACTER*80 KEYWRD(8), MESG*200
*
      INUM = 1
      KEYWRD(1) = 'Image' // CHAR(0)
      KEYWRD(2) = 'Title' // CHAR(0)
*
* Set up Search Condition
*
      ICOND(1)  = 1
      ICOND(2)  = EXPLOC (KEYWRD(1))
      ICOND(3)  = EXPLOC (INUM)
      ICOND(4)  = IntegerValue
*
* Set up control structure
*
      IBLOCK(1) = 0
      IBLOCK(2) = ReadHeader .AND. SearchCondition
      IBLOCK(3) = EXPLOC (KEYWRD(2))
      IBLOCK(4) = EXPLOC (MESG)
      IBLOCK(5) = StringValue
      IBLOCK(6) = EXPLOC (ICOND)
*
* Perform the operation
*
      IRETERR = EXPRDF (ISTREAM, IBLOCK, IRDERR)
      IF (IRETERR .NE. RoutineSucceeded) THEN
       MESG = EXPREP (IRETERR, IRDERR, .TRUE.)
      ELSE
       PRINT *, KEYWRD(3), MESG
      ENDIF
*
```

5.5 INTEGER FUNCTION ExpgWriteDataFile (Stream, Structure, DataErrorValue)

ALTERNATIVE: INTEGER FUNCTION EXPWDF (Stream, Structure, DataErrorValue)
INPUT: INTEGER Stream
OUTPUT: INTEGER *DataErrorValue
INOUT: INTEGER Structure[6]
FUNCTION: performs various write operations on the file.

Writing *data* to the file always causes the data to be written immediately since the libraries have no way of knowing how long this data may exist.

Writing *header information* does not always cause an explicit write to the file (they will be cached in memory) to improve performance and help prevent fragmentation of the file. This header information will only be written either by using the FlushTable command or by closing the file.

If you write header information to the data file, or add new header information then you **must** explicitly call ExpgCloseDataFile to flush the internal tables - failure to do so will mean that the new information is not reflected in the file.

5.5.1 Example

To fill in this Structure to Write an image (with value identified by an integer), the user would write the following code. Note that we use a search condition to find where to write the data, we also assume a stream is already open and the data exists:

```
*
      INCLUDE 'esrf_data_format.inc'
*
      INTEGER   ISTREAM, IRETERR, IRDERR, INUM, IBLOCK(6), ICOND(40),
SIZES(3)
      INTEGER*2 IDATA(2000000)
      CHARACTER*80 KEYWRD(8), MESG*200
*
      INUM = 1
      KEYWRD(1) = 'Image' // CHAR(0)
*
* Set up the sizes of the data
*
      SIZES(1) = 2
      SIZES(2) = 512
      SIZES(3) = 256
*
* Set up Search Condition
*
      ICOND(1)  = 1
      ICOND(2)  = EXPLOC (KEYWRD(1))
      ICOND(3)  = EXPLOC (INUM)
      ICOND(4)  = IntegerValue
*
* Set up control structure
*
```

```
      IBLOCK(1) = UnsignedShort
      IBLOCK(2) = WriteData.AND. SearchCondition
      IBLOCK(3) = EXPLOC (IDATA(1))
      IBLOCK(4) = EXPLOC (SIZES(1))
      IBLOCK(5) = 0
      IBLOCK(6) = EXPLOC (ICOND)
*
* Perform the operation
*
      IRETERR = EXPWDF (ISTREAM, IBLOCK, IRDERR)
      IF (IRETERR .NE. RoutineSucceeded) THEN
                                  MESG = EXPREP (IRETERR, IRDERR, .TRUE.)
                                  PRINT *, MESG
      ENDIF
*
```

5.6 CHARACTER*(*) FUNCTION ExpgDataFormatVersion ()

ALTERNATIVE: CHARACTER*(*) FUNCTION EXPDFV ()
INPUT: None
OUTPUT: None
INOUT: None
FUNCTION: returns a string value for the current version of the software.

5.7 SUBROUTINE ExpgCatchInterrupt (Flag)

ALTERNATIVE: SUBROUTINE EXPCI ()
INPUT: INTEGER Flag
OUTPUT: None
INOUT: None
FUNCTION: Toggles the interrupt catch facility on or off. If the Flag is True then the signal SIGINT (usually ^C) is caught and all files are closed before any other action. If the Flag is False then the old signal handler (which may be no action) is re-established and no flushing of headers is performed. The default start-up action is OFF.

5.8 CHARACTER*(*) FUNCTION ExpgReportStatus (ErrorValue, DataErrorValue, Flag)

ALTERNATIVE: CHARACTER*(*) FUNCTION EXPREP (ErrorValue, DataErrorValue, DisplayFlag)
INPUT: INTEGER ErrorValue
INTEGER DataErrorValue
INTEGER Flag
OUTPUT: None
INOUT: None
FUNCTION: give a report on the error that occurred.

give a report on the error that occurred. The return value is a string with the appropriate message. The value **ErrorValue** is the return value of the routine (generally RoutineFailed), **DataErrorValue** is the returned error code from the routine that the user wishes to report on and **Flag** is set to .TRUE. if the user

wishes the routine to output a message or .FALSE. if the user does not want the routine to output a message. A typical piece of code may look like:

```
      include 'esrf_data_format.inc'
      integer ErrorValue, DataErrorValue
      character*200 mesg
      ErrorValue = ExpgOpenDataFile ('myfile", DataErrorvalue)
      if (ErrorValue .ne. RoutineSucceeded mesg = ExpgReportStatus (Error-
Value, DataErrorValue, .True.)
```

5.9 INTEGER FUNCTION ExpgDisplayHeaders (Stream, DataErrorValue)

ALTERNATIVE: INTEGER FUNCTION EXPHDR (Stream, DataErrorValue)
INPUT: INTEGER Stream
OUTPUT: INTEGER DataErrorValue
INOUT: None
FUNCTION: this routine the will display all the headers currently in memory to the standard output device.

5.10 LOGICAL FUNCTION ExpgGetByteOrder ()

ALTERNATIVE: LOGICAL FUNCTION EXPGBO ()
INPUT: None
OUTPUT: None
INOUT: None
FUNCTION: determine 'endedness' of your machine

A support routine to determine the byte order of your machine. Return Values are HighByteFirst (True) or LowByteFirst (False).

This routine describes the way in which integers are stored in the data (either 1234 or 4321).

5.11 INTEGER FUNCTION ExpgLocation (ILOC)

ALTERNATIVE: INTEGER FUNCTION EXPLOC (ILOC)
INPUT: INTEGER LOC
OUTPUT: None
INOUT: None
FUNCTION: return an integer address for a variable

This function returns an integer value that is the address of it's argument. It performs the same function as %LOC () in VAX Fortran. It is used to assign the Control Block to the appropriate element of the Structure argument.

6.0 Fortran Examples

6.1 Writing Data

A simple program fragment to read an image in a file:

```
*
       INCLUDE 'esrf_data_format.inc'
*
       INTEGER IVALUE, IRETURN, IRESULT, INUM, II,
      +        IBLOCK(6), ICOND(40), ISIZE(10)
       CHARACTER*80 KEYWRD, MESG*200, FNAME
       INTEGER*2 IDATA(2000000)
*
      INUM = 1
      KEYWRD = 'Image' // CHAR(0)
*
* Set up Search Condition
*
       ICOND(1) = 1
       ICOND(2) = EXPLOC (KEYWRD)
       ICOND(3) = EXPLOC (INUM)
       ICOND(4) = IntegerValue
*
*     Read the data back as SignedShorts
*
       IBLOCK(1) = SignedShort
       IBLOCK(2) = WriteData .AND. SearchCondition
       IBLOCK(3) = EXPLOC (IDATA)
       IBLOCK(4) = EXPLOC (ISIZE)
       IBLOCK(5) = 0
       IBLOCK(6) = EXPLOC (ICOND)
*
       IRETURN = EXPWDF (IVALUE, IBLOCK, IRESULT)
       IF (IRETURN .NE. RoutineSucceeded) THEN
       MESG = EXPREP (IRETURN, IRESULT, .TRUE.)
       PRINT *, MESG
       ENDIF
*
```

6.2 Writing Headers

A simple program fragment to read a title from an image in a file:

```
*
       INCLUDE 'esrf_data_format.inc'
*
       INTEGER IVALUE, IRETURN, IRESULT, INUM, II,
      +        IBLOCK(6), ICOND(40), ISIZE(10)
       CHARACTER*80 KEYWRD, MESG*200, FNAME
       INTEGER*2 IDATA(2000000)
*
       INUM = 1
       KEYWRD = 'Image' // CHAR(0)
       FNAME = ‘Title’//CHAR(0)
       MESG = ‘This is an example Title’//CHAR(0)
*
```

```
* Set up Search Condition
*
      ICOND(1) = 1
      ICOND(2) = EXPLOC (KEYWRD)
      ICOND(3) = EXPLOC (INUM)
      ICOND(4) = IntegerValue
*
*     Read the data back as SignedShorts
*
      IBLOCK(1) = NoSpecificValue
      IBLOCK(2) = WriteHeader .AND. SearchCondition
      IBLOCK(3) = EXPLOC (FNAME)
      IBLOCK(4) = EXPLOC (MESG)
      IBLOCK(5) = StringValue
      IBLOCK(6) = EXPLOC (ICOND)
*
      IRETURN = EXPRDF (IVALUE, IBLOCK, IRESULT)
      IF (IRETURN .NE. RoutineSucceeded) THEN
      MESG = EXPREP (IRETURN, IRESULT, .TRUE.)
      PRINT *, MESG
      ENDIF
*
```

6.3 Inquiring About Data

A simple program fragment to find the size and data type of an image in a file:

```
*
      INCLUDE 'esrf_data_format.inc'
*
      INTEGER   IVALUE, IRETURN, IRESULT, INUM, II,
     + IBLOCK(6), ICOND(40), ISIZE(10)
      CHARACTER*80 KEYWRD(8), MESG*200
*
      INUM      = 1
      KEYWRD(1) = 'Image' // CHAR(0)
*
* Set up the search condition
*
      ICOND(1)  = 1
      ICOND(2)  = ExpgLocation (KEYWRD(1))
      ICOND(3)  = ExpgLocation (INUM)
      ICOND(4)  = IntegerValue
*
* Set up the control structure
*
      IBLOCK(1) = 0
      IBLOCK(2) = InquireData .AND. SearchCondition
      IBLOCK(3) = 0
      IBLOCK(4) = ExpgLocation (ISIZE)
      IBLOCK(5) = 0
      IBLOCK(6) = ExpgLocation (ICOND)
*
      IRETURN = ExpgReadDataFile (IVALUE, IBLOCK, IRESULT)
      IF (IRETURN .NE. RoutineSucceeded) THEN
      MESG = ExpgReportStatus (IRETURN, IRESULT, .TRUE.)
      ELSE
      PRINT *, 'Data Type ', IBLOCK(1)
      PRINT *, 'Dimensions ', ISIZE(1)
```

```
      DO 100 II = 2, ISIZE(1) + 1
      PRINT *, 'Dim ', ISIZE(II)
 100  CONTINUE
      PRINT *, 'Total Size ', ISIZE(ISIZE(1) + 2)
      ENDIF
*
```

6.4 Reading Data

A simple program fragment to read an image in a file:

```
*
      INCLUDE 'esrf_data_format.inc'
*
      INTEGER IVALUE, IRETURN, IRESULT, INUM, II,
     +        IBLOCK(6), ICOND(40), ISIZE(10)
      CHARACTER*80 KEYWRD, MESG*200, FNAME
      INTEGER*2 IDATA(2000000)
*
     INUM = 1
     KEYWRD = 'Image' // CHAR(0)
*
* Set up Search Condition
*
      ICOND(1) = 1
      ICOND(2) = EXPLOC (KEYWRD)
      ICOND(3) = EXPLOC (INUM)
      ICOND(4) = IntegerValue
*
*     Read the data back as SignedShorts
*
      IBLOCK(1) = SignedShort
      IBLOCK(2) = ReadData .AND. SearchCondition
      IBLOCK(3) = EXPLOC (IDATA)
      IBLOCK(4) = EXPLOC (ISIZE)
      IBLOCK(5) = 0
      IBLOCK(6) = EXPLOC (ICOND)
*
      IRETURN = EXPRDF (IVALUE, IBLOCK, IRESULT)
      IF (IRETURN .NE. RoutineSucceeded) THEN
      MESG = EXPREP (IRETURN, IRESULT, .TRUE.)
      PRINT *, MESG
      ENDIF
*
```

6.5 Reading Headers

A simple program fragment to read a title from an image in a file:

```
*
      INCLUDE 'esrf_data_format.inc'
*
      INTEGER IVALUE, IRETURN, IRESULT, INUM, II,
     +        IBLOCK(6), ICOND(40), ISIZE(10)
      CHARACTER*80 KEYWRD, MESG*200, FNAME
      INTEGER*2 IDATA(2000000)
*
      INUM = 1
```

```
      KEYWRD = 'Image' // CHAR(0)
      FNAME = ‘Title’
*
* Set up Search Condition
*
      ICOND(1) = 1
      ICOND(2) = EXPLOC (KEYWRD)
      ICOND(3) = EXPLOC (INUM)
      ICOND(4) = IntegerValue
*
*     Read the data back as SignedShorts
*
      IBLOCK(1) = NoSpecificValue
      IBLOCK(2) = ReadHeader .AND. SearchCondition
      IBLOCK(3) = EXPLOC (FNAME)
      IBLOCK(4) = EXPLOC (MESG)
      IBLOCK(5) = StringValue
      IBLOCK(6) = EXPLOC (ICOND)
*
      IRETURN = EXPRDF (IVALUE, IBLOCK, IRESULT)
      IF (IRETURN .NE. RoutineSucceeded) THEN
      MESG = EXPREP (IRETURN, IRESULT, .TRUE.)
      ENDIF
      PRINT *, MESG
*
```

7.0 System Limits

The following define statements are used in the header file esrf_data_format.h to specify limits and header syntax:

#define MaxFiles 20

The maximum number of files that can be open at any one time. It's value is purely arbitrary and can be changed by the installer of the library.

#define MaxKeyLen 64

The significant length of a keyword. Keywords can be of any length but only the first MaxKeyLen characters are recognised as distinct. Can be changed by the installer of the library.

#define BufferSize 512

The size of the input buffer. The files are parsed by reading blocks of this size and then parsing through them in memory. This should allow the file opening to be quick and can be matched to the block size of your disks.

#define HeaderStart 1

Header numbers will begin at this value. Negative header numbers are not allowed, neither is a header number of zero.

#define MaximumMatch 2048

The maximum number of matches that can be made on a file of data. This has serious implications and this limit may be removed in future releases. It is arbitrarily set - this means that a search for the keyword Image (to determine the number of images in a file) cannot be greater then this define i.e. you can't have more than MaximumMatch images in a file.

#define MaxDimensions BufferSize

The maximum number of dimensions for a block of data. It arbitrarily uses the value of BufferSize but since this value is unlikely to change (or if it does it will be increased) the limit s not really important.

8.0 Error Codes and Messages

8.1 Error Messages

The following is a list of retruned values from the calls and a fuller description of each message.

RoutineSucceeded

is the return value of a call that has performed it's function correctly. This value is guaranteed to be zero all other error values are non-zero.

RoutineFailed

the routine could not perform it's function correctly and the last argument to the call will contain one of the error values listed below.

CouldNotMallocMemory

the malloc system call failed. This is a serious error and the program should not continue.

CouldNotFreeHeaders

when closing the file, the system failed to free the header tables it had created. This is not a serious error but the user should beware that this channel number should not be re-used if at all possible.

NoMoreStreamsAvailable

the maximum number of files are open. Close some files or abandon the program.

CouldNotOpenFile

An error occurred with the fopen system call. This may be caused by the user having incorrect permissions on a file that exists or no write permission to the directory where a new file is created.

EndOfFileDetected

There was an unexpected end of file found during the opening and parsing of the file. The program should not continue

CouldNotFindHeader

A warning that the file does not contain any headers

BadSizeDefinition

An error with writing data, the system found a size that was negative.

BadDataBlock

An error occurred during a fread/fwrite call that produced a mismatch between the number of items to read/write and the actual number.

CouldNotFindKeyword

This is not an error but a warning that the system failed to find your given keyword.

NotESRFDataFile

The file does not start with a ‘{‘ as it’s first non-blank character so it’s unlikely to be an ESRF Data File.

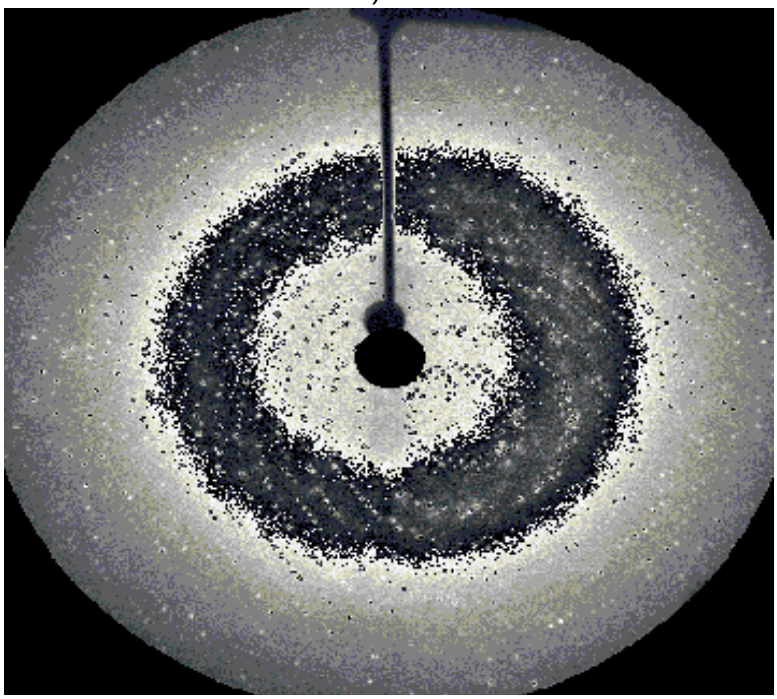
9.0 An interface to *XV*

9.1 Introduction

Xv is written by John Bradley (e-mail: bradley@cis.upenn.edu) and is described by him as an “interactive image display for the X Window System”. The current version has a license fee of $25. An unregistered copy can be obtained from the server ftp.upenn.cis.edu (IP Number 130.91.6.8) as the file pub/xv/xv-3.00a.tar.Z.

It is quite a useful tool for looking at a variety of image formats (GIF, TIFF, JPEG, etc. even PostScript documents!) and has been modified by the author of the ESRF Data Format to read our format as well. The patches will shortly be made available on our ftp server expga.esrf.fr. However, it is not recommended that it be used to write ESRF Data Format files (although this is possible) since Xv itself can only use (internally) 8- and 24-bit image data so some loss of resolution may occur.

For example the image below was generated by reading and ESRF Data Format File and converting it to TIFF for input to FrameMaker (the desktop publishing system used to create this documentation)



Appendix A Obtaining the Software

The library source files, header files and some example programs are available for the following machines from the file server expga.esrf.fr (Internet number 160.103.2.141) as follows:

~ftp/src/esrf_data_format.tar.Z - source code, header files and Makefiles

~ftp/src/esrf_examples.tar.Z - example test files; these are rather large so have been split from the main source

If you need to compile the Fortran interface, you will also need a package called cfortran.h obtainable from the ftp site is zebra.desy.de (131.169.2.244)

Appendix B Common Keywords

In order to standardise on keyword names as far as possible the following is a list of pre-defined keywords that should be used for their said purpose:

Keyword	Properties and/or use
HeaderID	provides a header number and other information (parent header, next header) to link headers together. This keyword is **COMPULSORY** for every header,
VersionNumber	a string value indicating the version of the software that wrote the file,
ByteOrder	symbolic values HighByteFirst or LowByteFirst, used to describe the 'endedness' of integers and IEEE floating point numbers,
DataType	symbolic values UnsignedByte, SignedByte (both 8 bit), UnsignedShort, SignedShort (both 16 bit), UnsignedInteger, SignedInteger (both 32 bit), Real (32 bit IEEE or 32-bit VMS), DoubleValue (64 bit IEEE or 64-bit VMS). See the section on pre-defined types,
RealFormat	specify the format of floating point numbers. Can be set to VMS, IEEE or ConvexReal (the default is IEEE if omitted),
Size	Size of the Data Section following this header - may be omitted for Size = 0,
Dim_1,Dim_2,Dim_3, etc.	the size of each dimension of the data actually stored. The maximum number of dimensions is set in the header file define MaxDimensions (currently 512) and follow the C language ordering,
Compression	symbolic names NoSpecificValue (no compression), DiffDataCompress, JPEG, JPEG-Lossy (for JPEG lossy compression), RunLengthEncoded and other techniques as they are implemented,
Date	the date the file was written, any format common DD/MM/YY, MM/DD/YY, DD-MON-YY, etc.
Time	the time the file was written, any common format hh:mm zone e.g. 23:40 CET,
Image	an integer to specify a particular image e.g. Image = 1;

ImageType	a symbolic name to describe the Image type. Current values are: 8bit (8 bit greyscale or colour palette images), 24bit (true colour images), RawData (used to describe raw diffraction data). Note that using RawData does not imply any size of pixel element - that is the purpose of the DataType keyword,
DetectorName	a string to identify the detector type used, the current values are NoSpecificValue (for a detector that is not yet known), Film (for photographic film), MAR (MAR Research Image Plate), RAXIS (Image Plate), MOLD (Molecular Dynamics Image Plate), FUJI (Image Plate), each of these known detectors will, in general, have several other keywords defined that describe the properties of the detector - see the list later,
Title	a title for the header,
SampleName	a string to identify the sample used,
ProposalNumber	value assigned by ESRF from original beam time application

The following subsidiary keywords are used to describe the properties of various image types. The main ones deal with the description of Image Plates and Film detectors.

Subsidiary Keyword for Image Plates and Film Detectors

Subsidiary Keyword	Properties and/or use
DataLowLimit	the lower limit for a valid pixel reading,
DataHighLimit	the upper limit for a valid pixel reading,
BadPixelMarker	offset used to mark a data point as invalid. the pixel value + BadPixelMarker makes this pixel an invalid data item,
ZeroOffset	number to subtract from standard formula to compensate for the zero offset of the detector,
CorrectionFunction	symbolic name for the different types of nonlinearity corrections to apply to the raw data; values are: Polynomial (1 byte), Linear (2 byte), ExponentialWord (2 byte), ExponentialByte (1 byte) and PieceWiseLinear (2 byte, RAXIS),
HeaderSize	the size of the header in the raw output from the detector,
ScaleFactor	scale factor to apply to the data; typically 1 but not for RAXIS,

Subsidiary Keyword for Image Plates and Film Detectors

Log10Scale	for Linear CorrectionFunctions this is zero; otherwise it is the constant in the exponent for Exponential CorrectionFunctions,
PadParameters	two integer values, the number of records and the number of bytes respectively; some detectors require a skip of n bytes every m records, for example a FUJI BA100 requires an 8 byte skip every 4 records and is described as PadParameters = 4 8; a FUJI BA101 requires a 2 byte skip every 4 records and is described as PadParameters = 4 2;
FilmType	describes different types of film; currently NoSpecificValue, Film2 or Film4. This will change the way various corrections are applied

Appendix C Future Plans and Updates

In coming versions of the software several improvements are planned (although there is no definite release date for these improvements):

1. A RecordDescriptor keyword that will provide a record based view of the data. In this the user will be able to describe tabular data with a statement like:

```
RecordDescriptor = IntegerValue[3] StringValue[80] RealValue;
```

   to describe a line of the data as 2 integer values then an 80 character string followed by a real number.

2. A SQL-like parser may be provided so that a user may specify something like:

```
SELECT DataValues FROM StreamNumber = 1 WHERE Image = 1;
or
SELECT Header Title as StringValue FROM StreamNumber = 1 WHERE Image = 1;
or
WRITE Title as StringValue TO StreamNumber = 1 WHERE Image = 1;
```

Appendix D Changes to the Library

D.1 Changes between Version 1.0 and Version 1.1 of the Library

There is a clash with some definitions in X11R4 and X11R5 so the following have had their names changed:

Convex	becomes ConvexReal
NoValue	becomes NoSpecificValue

Appendix E Colophon

This manual was produced using the *FrameMaker* 4.0 Desktop Publishing package. It is mostly set in 10pt Helvetica font, Section headings are in Helvetica Bold font at 12, 10, 9 and 8pt. The examples of code are set in 8pt Courier.

Diagrams (as such) were imported as TIFF format files from various sources.