The KerGIS Correspondence Library

(Work in progress—handle with care)
$Id: libcorr.w,v 1.35 2006/11/16 19:15:24 tlaronde Exp

Thierry LARONDE tlaronde@polynum.com

Abstract

The core functions of KerGIS—and its *raison d'être*—are geometrical ones. Non geometrical attributes (typically text ones) are peripheral and hence shall be kept distinct. But this does not mean that a way to associate KerGIS geometrical elements with non geometrical informations has to be ignored.

To handle the |correspondence| between partitions and these attributes is the purpose of the `libcorr`.

	Section	Page
Licence	1	2
Overview	2	3
How the attributes are passed	3	4
Definitions	4	5
Properties	5	6
Limits of the present implementation	8	7
Public interface to the library	9	8
Opening and closing	10	9
Managing images	16	10
Memory management	19	11
The *C_Datum* information structure	20	12
The *C_Data* structure	36	16
The *C_Att* information structure	38	17
The *C_Image* information structure	40	18
Syntax of the head and data description files	41	19
Implementation	45	21
Opening and closing	46	22
Reading and writing the head and data description files	55	27
Image manipulations	64	33
Fields conversions: `iconv` and other related transformations	66	35
Memory allocation	79	40
The `dbf` back-end	83	42
The `psql` back-end	84	43
The `stdin` back-end	85	44
The `stdout` back-end	86	45
Informations	87	46
Warnings	90	47
Error handling	93	48
Index	96	50

1. Licence.

2. Overview.

The *corr* library is the major component dealing with non-geometrical properties associated with geometrical objects.

Once an object has been assigned to a group (it may be the sole one in this group, in which case we have a singleton), this number—group id— can be linked with other non geometrical properties.

A correspondence is a relation between a *partition*—handled by the `libpart` and UD facilities—and a *dataset*. This is called a correspondence since it is in no way a function. An *image* of a group—identified by its *groupid*—can have 0, 1 or more attributes.

Since KerGIS is at a crossroads of many sources of informations, the importation and exportation of informations must be considered with great care.

The primary concern is accuracy, meaning here: not loosing information by accessing data. As a corollary, the ability to store this correct imported information in a special format (for backup or exchange): the *KerGISAttributesText* format, or `kat`, will be provided.

The `libcorr` provides a defined API for programs accessing attributes.

Importing/exporting from/to a `libcorr` known back-end format to another one is now indeed a 1:1 problem: the back-end imports and exports in the KerGIS format, meaning that once a back-end is written the importation/exportation to other known formats is ready since everything passes through the correspondence library.

Since the `libcorr` can be used as a filter—for example when exporting KerGIS geometrical elements in another format—, the library supports the notion of an *anonymous* correspondence: in this case, the library will just connect to one or more back-ends, without creating a *head* element that is the description of the correspondence stored in the database.

This very same property of the library to be a proxy imposes the design of a format allowing description of widest range of data and the mandatory property of not loosing accuracy when converting **to** the KerGIS format. Indeed, the description of the attributes is made as a description of words of a virtual machine with variable length integers or IEEE 754 floats, and the values are passed as **UNICODE strings** encoded in *UTF-8*.

3. How the attributes are passed.

Back-ends convert to/from KerGIS format.

The attributes are passed as an array of *C_Att* structures. On the lower level, in the *C_Att*, the fields are passed as bytes strings that since all values are passed encoded in *UTF-8*, that is UNICODE represented as a bytes (octets) stream: the back-end retrieve the values from the back-end storage in its *encoding* and the higher level `libcorr` routines convert to *UTF-8* (KerGIS). Symetrically, the higher level converts from *UTF-8* to *encoding* before feeding the back-end.

Except when the logical type of the value is `C_RAW_L`, values coming from the back-ends shall have leading and trailing blanks removed: these are usually formatting candy and have nothing to do with the raw values.

Since an image can be defined as a huge amount of attributes, when the caller is asking for an image (read functions: *C_get_image*) the image is passed with a *nb_atts_here* of at most *chunk_size*, which is defined in the *Head* structure. If it was not set, it defaults to `C_MAX_NB_ATTS`. If there are more (the function returned an image with *multiplicity* $\neq$ *nb_atts_here*, the caller must recall the get image function recursively until he has gathered all the desired attributes (last chunk will have *multiplicity* $\equiv$ *offset* $+$ *nb_atts_here*.

This limit is not used when writing: we take what amount *nb_atts_here* is given.

⟨ Public macros 3 ⟩ ≡
#define `C_MAX_NB_ATTS` 100

See also sections 22, 23, 25, 26, 29, 31, and 39.

This code is used in section 9.

4. Definitions.

The `libcorr` establishes a correspondence between a partition of a geometrical set and an attributes set (that we will call simply: *dataset* in what follows).

The partition of a geometrical set is described in the KerGIS *libpart*.

The dataset is **not** a random collection of unspecified objects. The dataset is a set of elements that we call: *attribute*s, that are implemented in a structure *C_Att*, and that are all instances of a unique structure *C_Data* taking the form of an ordered list of *C_Datum*s. This ordered list of *C_Datum*s is the description and definition of the dataset. *C_Datum* is a description of a component of a *C_Data* and is **not** and element of the dataset. An *C_Att* is an instance of the *C_Data*. The *C_Image* associated with a geometrical group is then an array of *C_Att*s, the number of *C_Att*s being called the *multiplicity*. The *multiplicity* is the **total** number of attributes composing the image of the group. But an Image structure can hold less atts at a time (being a window in the actual image) of *nb_atts_here* number of attributes.

The `libcorr` deals with elements of the partition on the one side (a group), and elements of the dataset on the other side.

The *Correspondence* is defined by the graph mapping groups of some defined partition to attributes.

There can be several distinct partitions of a geometrical set. There can be several distinct correspondences between a given partition and distinct datasets. These correspondences are elements of the gis database and are identified by a name in this base.

5. Properties.

The first goal is *accuracy*. The data structure shall allow an access to some information sources without loss of accuracy, and to import in the KerGIS attributes format these sources without loosing accuracy from the original. **This does not mean that standard routines provided by the library will be able to handle infinite precision of floating point or unbounded ranges of integers for example**. This does mean that one of the storage formats —the KerGIS Attributes Text— will be able to encode all the elements from the source. The accuracy may be lost if this `kat` format is converted in another format. But a transformation of this `kat` back to the original format shall lead to equality if not identity (the same informations even if the two binary files are not exactly the same).

6. *portability* is the second goal, to let a gisdatabase be shared between differing hosts (not the same endianess, not the same word size and so on).

So the programs will convert from a fixed format data to machine dependent one.

In what follows, the following C types are used with the following meaning (in case a binary version of the data files is implemented), i.e. with the minimal sizes guaranteed by ISO C:

short : *int16_t* a wyde;

long : *int32_t* a tetra;

7. *efficiency* comes after, even if it is important it is not first.

A great deal of the efficiency will be achieved by the implementation of the drivers to the DBMs. The `libcorr` is actually what is described: a way to link geometrical groups with attributes. It is in no way a mean to manipulate efficiently pure attributes: this shall be better done by the dedicated DBMs.

8. Limits of the present implementation.

The present implementation will be limited to provide some essential needs. The definition of structures are attempts and nothing is carved into stone at the moment.

The immediate application of the `libcorr` was the development of the `dBASE` converters.

The basic data structures are described, but the management of the data, including reliable and efficient storage, and at least one query language have yet to be done.

The text translations from some locale to UNICODE/UTF-8 and back is done using *POSIX iconv* functions.

On system supporting the `iconv XSI` option, the functions are in the libc. For other systems, there is the need to link against an external implementation (for example GNU `libiconv`).

9. Public interface to the library.

We are going to describe the visible top of the iceberg first: what callers are supposed to see and use.

The naming and API conventions are the KerGIS ones: useful values are returned by putting the result in a memory place (parameters are passed as addresses in this case), the returning value being an **int** specifying the status: `G_SUCCESS` which is 0 for OK, and not 0 for an error.

```
⟨ corr.h 9 ⟩ ≡
#ifndef KERGIS_CORR_H
#define KERGIS_CORR_H
#define CORR_MAJOR 0
#define CORR_MINOR 1
#define CORR_REVISION 0
#include <math.h>      /* floor() */
#include <stdint.h>      /* POSIX exact number of bits types */
#include <iconv.h>      /* POSIX/XSI iconv conversion functions */
#include "ksys/cdefs.h"
#include "ksys/types.h"      /* id_kt */
#include "ksys/sys/iconv_defs.h"
#include "part.h"
  ⟨ Errors macros 93 ⟩
  extern const struct G_Msgs *const C_Errors;
  ⟨ Warnings macros 90 ⟩
  extern const struct G_Msgs *const C_Warnings;
  ⟨ Infos macros 87 ⟩
  extern const struct G_Msgs *const C_Infos;
  ⟨ Public macros 3 ⟩
  ⟨ Public structures 21 ⟩
  ⟨ Public prototypes 13 ⟩
#endif      /* KERGIS_CORR_H */
```

10. Opening and closing.

The user visible side of the `libcorr` has to deal with *C_Image*s: we gather attributes from the back-ends to create a unique image belonging to a defined *group* id.

At the core level of the `libcorr` will be handled the matching between group identifiers and attributes.

A user can request the image of a defined group, or set the image associated with a defined group. The attributes flags will tell if the attributes have to be created or deleted (add or mark as dead).

11. The mode for opening are the one supported by *G_open*. The policy is the *G_open* one too.

If the correspondence is *anonymous*—i.e. *Corr→name* $\equiv \Lambda$— the *head* informations will not be stored in the database: the `libcorr` will function as a filter.

Otherwise, when a link to a dataset is created, the definition of the dataset is stored in the `corr.head` sub-element.

Later, when the dataset is accessed, the information about the definition of the dataset is retrieved from there.

The *Head* stores information about the *partition* and the meta-properties of the *dataset* (how to access it).

The *Data* stores the description of the dataset.

12. Whether the session is a *transaction* one or not is back-end dependent. For example, the *psql* back-end handles transactions (if everything went fine, at closing time the transaction is committed; else it is rollbacked). But the *dbf* back-end is record oriented: every record is updated on its own.

13. The *C_open* function does what was advertised: build the correspondence identifier with the *partition* and *dataset* members, open it and fill the *C_Corr* structure.

⟨ Public prototypes 13 ⟩ ≡

```
int C_open(struct C_Corr *Corr, int oflag);
```

See also sections 14, 15, 16, 17, and 19.

This code is used in section 9.

14. When building a correspondence, a description in the form of a file holding labelled values can be used not only by the `libcorr` but altogether by external programs. Hence routines to read these files.

Reading the *C_Data* description is only necessary when building another dataset, or for the KerGIS Attributes Text format (yet to be implemented). So *C_read_data* will be used whether internally by the back-end dependent opening routines, or by a userland program creating a new dataset using the KerGIS text description.

⟨ Public prototypes 13 ⟩ +≡

```
int C_read_head(FILE *fp, struct C_Corr *Corr);
int C_write_head(FILE *fp, struct C_Corr *Corr);
int C_read_data(FILE *fp, struct C_Corr *Corr);
int C_write_data(FILE *fp, struct C_Corr *Corr);
```

15. Closing, from the caller's standpoint is quite simple. This **does not** free the structure. For this, call (afterwards...) *C_free_corr*.

⟨ Public prototypes 13 ⟩ +≡

```
int C_close(struct C_Corr *Corr);
```

16. Managing images.

Obviously *C_get_image* will not work if *oflag* $\equiv$ `O_WRONLY` when opening the data file.

C_get_image can be used with the special group `P_GROUP_ALL`. This group will cause the `C_DUMPING` flag to be set, and nothing else, except continue dumping will be allowed in the mean time.

Setting the correct *offset* is the responsability of the caller. *Image¬offset* is not updated at return, since it indicates the starting edge of the window.

⟨ Public prototypes 13 ⟩ +≡

int *C_get_image*(**const struct** *C_Corr* ∗*corr*, **struct** *C_Image* ∗*Image*);

17. Whether when modifying attributes, inserting or deleting, all is done with the *Image* structure. Flags are specified at the attribute level for the action to be taken. If *att.id* $\equiv 0$, the attribute is inserted. if *att.id* $\neq 0 \wedge$ `C_IS_MODIFIED`(*att*), the corresponding attribute is modified. Otherwise, **nothing is done**.

Since on return from other functions, the *att.id* can be set (for example if one is retrieving images from another correspondence), one needs to clear it before feeding another correspondence if the image is to be inserted.

`P_GROUP_ALL` can be used with *C_set_image*. It will cause the `C_RESTORING` flag to be set, and nothing else will be allowed during the mean time.

⟨ Public prototypes 13 ⟩ +≡

int *C_set_image*(**const struct** *C_Corr* ∗*Corr*, **struct** *C_Image* ∗*Image*);

18. Dumping and restoring can be done using the `P_GROUP_ALL` special group.

19. Memory management.

There are several pointers inserted in the structures, that are set to differing length strings. To avoid memory leak, let routines handle allocation.

An *C_Image* and an *C_Att* will always see there internals freed before filling then with new data. If one wants to save the data, it must be copied elsewhere first.

⟨ Public prototypes 13 ⟩ +≡

```
int C_alloc_image(struct C_Image **Image);
int C_free_image(const int nb_datums, struct C_Image *Image);
int C_clean_atts(const int nb_datums, struct C_Image *Image);
int C_alloc_atts(const int nb_datums, struct C_Image *Image, int nb_atts);
```

20. The *C_Datum* information structure.
We start by the bottom.

A *C_Datum* is three-fold:

1 at the raw level, the information is encoded as a vector of numbers;
2 at the administrative level (handling by the programs accessing directly the data) some properties have to be remembered (access time, permissions, range of values, etc.);
3 at the user level, the *C_Datum* has a *name*, and the numerical values have to be interpreted: the *C_Datum* has some logical meaning (hopefully).

21. To ensure the accuracy and the generality, we will describe the *C_Datum*s à la computer: a vector of words representing numbers.

The number of words will be encoded in a member *size*. More later about that. What is of interest right now is the way we are going to encode the word.

Since the size of the words on distinct computers varies, this raw level for us will have a singularity: the description is made for some hypothetical engine whose *C_Word* is of variable length, and is of two distinct flavors: whether `C_INTEGER_W` or `C_FLOAT_W`. The float type is always described à la IEEE 754. This is coded with the first bit of the *flags*.

For `C_INTEGER_W` type, the sign is considered and is coded with the 2nd bit of the *flags*.

Whether the machine stores this 'Word' natively or not (whether the machine is able to deal with a binary representation of this word, or has to interpret an ascii representation) is indicated by the 3th bit of the *flags*: if set, binary representation.

How the field of bits described has to be interpreted is put in the flags *type*.

```
⟨ Public structures 21 ⟩ ≡
#define C_INTEGER_W 0 /* default */
#define C_FLOAT_W °1
#define C_UNSIGNED_W 0 /* default */
#define C_SIGNED_W °2
#define C_ASCII_W 0 /* default: interpreted */
#define C_BINARY_W °4
  struct C_Word {
    unsigned char flags;
    unsigned short significand; /* sign included if relevant */
    unsigned short exp_max; /* K + 1 */
  };
```

See also sections 30, 34, 35, 36, 37, 38, 40, and 48.

This code is used in section 9.

22. Since there are some classical values, we will provide some already filled structures.

```
⟨ Public macros 3 ⟩ +≡        /* integers */
#define C_SET_W(Word, f, s, e)Word.flags ← (unsigned char) f;
  Word.significand ← (unsigned short) s;
  Word.exp_max ← (unsigned short) e;
#define C_UINT8_W(Word)C_SET_W  (Word, 0, 8, 0)
#define C_INT8_W(Word)C_SET_W  (Word, 2, 8, 0)
#define C_UINT16_W(Word)C_SET_W  (Word, 0, 16, 0)
#define C_INT16_W(Word)C_SET_W  (Word, 2, 16, 0)
#define C_UINT32_W(Word)C_SET_W  (Word, 0, 32, 0)
#define C_INT32_W(Word)C_SET_W  (Word, 2, 32, 0)
#define C_UINT64_W(Word)C_SET_W  (Word, 0, 64, 0)
#define C_INT64_W(Word)C_SET_W  (Word, 2, 64, 0)      /* floats */
#define C_IEEE_754_SINGLE_W(Word)C_SET_W  (Word, 3, 24, 8)
#define C_IEEE_754_DOUBLE_W(Word)C_SET_W  (Word, 3, 53, 11)
```

23. Given some number (i.e. some range), it may be usefull to be able to compute the bits needed for encoding a given number expressed as digits (the *significand* and the *exp_max*: the *nb_bits*, the actual number of bits used on some actual machine is another story).

To set the values for a numeric signed exact type with 10 significant digits (SQL would say: precision) and 4 digits after the decimal separator (SQL would say: scale):

First compute the number of bits needed to encode a 9 significant digits (and add one for the sign), noting that we need to find the maximum power of 2 used to reach this value, which will tell us that it can be represented by $n + 1$ bits (since the bit one represents 2^0), adding a bit to encode the sign (may be two complement or not the result is that one more bit is needed if signed):

$$significand = \lfloor lg(10^9) \rfloor + 1 + [signed] = \lfloor 9/log2 \rfloor + 1 + 1 = 31$$

We can note that if we request more than 9 digits of accuracy, an *int32_t* is not able to handle it since, for 10 digits unsigned we have:

$$mant = \lfloor 10/log2 \rfloor + 1 + [signed] = 34$$

Since there are some computations that will be done quite frequently, we can write a macro definition to compute the number of bits needed to represent a number of n digits (taking into account an optional sign).

```
⟨ Public macros 3 ⟩ +≡
#define C_DIGITS_TO_BITS(nb_of_digits, sign)
      (unsigned short)  ((int) floor((((double)(nb_of_digits)) * M_LN10/M_LN2)) + 1 + sign)
#define C_WORD_TO_DIGITS(Datum)
      (int)ceil  ((Datum.Word.significand + Datum.Word.exp_max) * M_LN2/M_LN10 + 4)
    /* 2 signs, dot and 'E' */
```

24. We have describes what are values stored (the raw level). But these values have—hopefully—a meaning for some user.

The way a value is stored (as a vector of numbers i.e. *Word*s) must belong to the black box: the user doesn't care. What the user inputs, and what is displayed on output is an **interpretation** of the values stored as numbers. The member of the structure *C_Logical* holding these informations will be named *type*. *type* is not exactly a field of bits, since some related types share a bit. But some bits are reserved for some kind of types (example: text), so that a test can be easily implemented.

The type will be encoded as a byte (*uint8_t*), and all values < 128 are whether defined by KerGIS or reserved.

All values with the seventh bit set (≥ 128) can be used to encode an interpretation that doesn't fit with the defined ones and will not be used by KerGIS.

25. A raw vector of bits/bytes is not encoded (or, if we are under interpretation of exporting in `kat` it is stored as a string consisting of 2 chars ASCII representing a hexadecimal value).

```
⟨ Public macros 3 ⟩ +≡
#define C_RAW_L 0 /* uninterpreted: an array of numbers BLOB */
```

26. All strings are encoded in *UTF-8*and have the least significant bit set.

```
⟨ Public macros 3 ⟩ +≡
#define C_IS_STRING(Datum) (Datum.Logical.type & °1 )
#define C_TEXT_L °1 /* strings */
#define C_DATE_L °3 /* a string */
#define C_URI_L °11 /* a text string to be interpreted */
```

27. `C_URI_L` is indeed a text encoded in *UTF-8* but that has to be interpreted according to the protocol specified.

28. `C_DATE_L` is a string in ISO 8601 format (YYYY-MM-DDTHH:MM:SS.DD...).
TODO: handle this correctly.

29. There are numericals values too, including `C_LOGICAL_L` that is not doomed to be only 2 bits (the number of bits is defined via the **C_Word** structure).

The times information are generally encoded whether as a number of units (generally seconds) from a conventional point in time (on UNIX the 'epoch' 1970-01-01 00:00:00), or as a string (broken down time).

Note too that `C_DECIMAL_L` is an exact type that shall be encoded as an integer. The scale and other values are described on the administrative level.

```
⟨ Public macros 3 ⟩ +≡ /* numerical values shared the third bit */
#define C_IS_NUMERIC(Datum) (Datum.Logical.type & °4 )
#define C_TIME_L °4 /* a number of seconds relative to the epoch */
#define C_INTEGER_L °6
#define C_LOGICAL_L °14 /* boolean for example */
#define C_DECIMAL_L °16 /* fixed decimal separator */
#define C_FLOAT_L °24 /* a floating point number */
```

30. At the higher level, for the user the *C_Datum* must have a *name* (that will be encoded in *UTF-8*).
We can now describe the *C_Logical* structure.

```
⟨ Public structures 21 ⟩ +≡
  struct C_Logical {
    char type;
    char *name; /* UTF-8 encoded */
  };
```

31. These *C_Datum*s have some stored values, and some meaning. Now the problem arises to manage this: this is the administrative level.

We have already said that we will handle vector of words. So it is time to introduce the *size*. We do not support unspecified length: the maximum shall always be specified. But for KerGIS proper, everything is stored as variable length.

Some *flags* have to be specified, telling us whether Λ is allowed or not, whether a *default_value* is provided or not, whether this is auto-incrementing (a special case of variable default), and whether all values must be distinct or not (distinct is considered for not Λ values).

```
⟨ Public macros 3 ⟩ +≡
#define C_NULL_ALLOWED_A 1
#define C_HAS_DEFAULT_A 2
#define C_ALL_DISTINCT_A 4
#define C_AUTO_INCREMENT_A (C_HAS_DEFAULT_A | C_ALL_DISTINCT_A | 8)
```

32. A *default_value* (always encoded whether in ASCII—for a number—or in *UTF-8*) can be provided.

33. Secondly, the values stored have to be interpreted. But there are some common manipulations that we may ease by providing the informations without needing to define specific functions.

For example, numbers can be scaled (this is especially true about numeric values that are exact numbers with a fixed decimal point one could say “only for output”).

So we will consider the general case of an **integer** *shift* (a constant to add), a *scale* (a power of 10 since it is on the user side, power of 2 being on the computer’s one) and a *unit*. The effective number is then computed as follows:

$$number = shift + stored\_value * unit * 10^{scale}$$

By default, *shift* $\leftarrow 0$, *unit* $\leftarrow 1$ and *scale* $\leftarrow 0$. All these values are of signed integer type.

In the case of `C_AUTO_INCREMENT_A`, by using the *shift* and *unit* one can achieve the result: the stored value will start at 0 (hence at *shift*) and will be increased by 1 hence leading to a step of $unit * 10^{scale}$ being *unit* if *scale* $\equiv 0$.

34. Hence the result.

```
⟨ Public structures 21 ⟩ +≡
  struct C_Admin {
      /* input/output */
    unsigned short flags;
    unsigned long size;      /* the effective number is computed with these */
    long shift;
    long unit;
    long scale;
    char *default_value;
  };
```

35. Finally, composing the three levels gives the *C_Datum* structure.

```
⟨ Public structures 21 ⟩ +≡
  struct C_Datum {      /* user level */
    struct C_Logical Logical;      /* admin */
    struct C_Admin Admin;      /* raw level */
    struct C_Word Word;
  };
```

36. The *C_Data* structure.

The *C_Data* describes a list of *C_Datum*s and is the definition of the dataset for us. Its structure is then simple.

- the number of *C_Datum*s *nb_datums*;
- the total number of *C_Att*s *nb_att*;
- a vector of *C_Datum*s;

⟨ Public structures 21 ⟩ +≡

```
struct C_Data {
  unsigned short nb_datums;
  unsigned long nb_atts;
  struct C_Datum *Datums;
};
```

37. Beside the data, there is the highest level dealing with meta-organization of the data: the way to access it, the partition to link with, the encoding, the place and some candy for user interaction.

This is dealt with via the *C_Head* structure that has an immediate counterpart in the `GISDATABASE`: a text file to be found under the `CORR_HEAD_SUBELT` offering the user the capacity to store values for accessing some data source without having to type it other and other when accessing it.

As for other elements stored in the directory based KerGIS database (`GISDATABASE`), it is identified by its *name*.

When invoking *C_open*, the *name* member shall be set to the name identifier of the correspondence. If *name* $\equiv \Lambda$, an anonymous correspondence is wanted, that is no *Head* will be written in the gisdatabase.

The structure has, as members, information that are set in the header file. Hence the *C_Head* structure. On opening, how to open must be set, whether by specifying a *name* identifying a text version of the *C_Head* in the KerGIS database, or by filling all the values.

On return, the *Data* and *handler* are filled.

⟨ Public structures 21 ⟩ +≡

```
#define C_NB_HEAD_VALUES 13
  struct C_Head {
    char *driver;
    char *partition;
    char *cluster;
    char *dbase;
    char *table;
    char *encoding;
    char *datum_group;
    char *datum_group_name;
    char *group_empty_as;
    char *group_all_as;
    char *null_as;
    char separator;
    unsigned long chunk_size;      /* negociate this amount of atts at a time */
  };
  struct C_Corr {
    int handler;
    char *name;
    struct P_Map Mapping;
    struct C_Head Head;
    struct C_Data Data;
  };
```

38. The *C_Att* information structure.

The *C_Att* is an instance of the data, and a component of the image. The actual values of each *C_Datum* is returned as a vector of words, the number of words being given. Hence the *length* must be interpreted according to the definition of the *C_Datum*, interpreted or not interpreted, text value or not etc.

```
⟨ Public structures 21 ⟩ +≡
  struct C_Att {
    unsigned long id;      /* the identifier, set by caller on request */
    unsigned short flags;
    char **fields;      /* a field is an instance of a datum */
  };
```

39. We need at least to be able to handle the case of the deleted and modified attributes.

```
⟨ Public macros 3 ⟩ +≡
#define C_ATT_DELETED °1
#define C_IS_DELETED(Att) (Att→flags & °1)
#define C_ATT_MODIFIED °2
#define C_IS_MODIFIED(Att) (Att→flags & °2)
```

40. The *C_Image* information structure.

The image is the image of this *group*, and is composed of a *multiplicity* of *C_Att*s (total) but may be retrieved by chunks, hence the *offset* (in the image) and the actual *nb_atts_here* in this. If *offset* $\equiv 0$ and *nb_atts_here* $\equiv$ *multiplicity*, you have the whole image.

```
⟨ Public structures 21 ⟩ +≡
  struct C_Image {
    id_ktgroup;
    unsigned long multiplicity;
    unsigned long offset;
    unsigned long nb_atts_here;
    struct C_Att *Atts;      /* an array of C_Att */
  };
```

41. Syntax of the head and data description files.

The head and data are simple ascii (ascii for the labels, text for the values) file.

All values not given with have the *Default_head* or *Default_data* ones.

Since the whole is a description of a single entity (the correspondence), the parser will be unique even if the head and data files are stored separately (think of a matter of subfiles in some filesystems if you want).

This description will be translated obviously in `yacc` and `lex` files.

42. Comments are not token starting by a '#' up to the end of line.

43. There are 4 major types of tokens:

1. a `NAME` is a reserved word and defined à la C;
2. a `STRING` is enclosed in double quotes, does not contain a double quote or a new line;
3. an `INT` is a signed integer;
4. a char is given à la C i.e. enclosed in single quotes. Escaped sequences are allowed.

The reserved word Λ can be used in some cases in place of `NAME` or `STRING`. Hence the introduction of the rules $\langle name \rangle$ and $\langle string \rangle$.

44. The syntax, first for head, then for data.

⟨*statements*⟩→ /* empty */ | ⟨*statements*⟩ ⟨*statement*⟩

⟨*statement*⟩→ ⟨*driver*⟩ | ⟨*partition*⟩ | ⟨*cluster*⟩ | ⟨*dbase*⟩ | ⟨*table*⟩ | ⟨*encoding*⟩ | ⟨*datum_group*⟩ | ⟨*datum_group_name*⟩ | ⟨*group_empty_as*⟩ | ⟨*group_all_as*⟩ | ⟨*chunk_size*⟩ | ⟨*null_as*⟩ | ⟨*separator*⟩ | ⟨*nb_datums*⟩ | ⟨*nb_atts*⟩ | ⟨*datums*⟩

⟨*driver*⟩→ DRIVER **':'** ⟨*name*⟩

⟨*partition*⟩→ PARTITION **':'** ⟨*string*⟩

⟨*cluster*⟩→ CLUSTER **':'** ⟨*string*⟩

⟨*dbase*⟩→ DBASE **':'** ⟨*string*⟩

⟨*table*⟩→ TABLE **':'** ⟨*string*⟩

⟨*encoding*⟩→ ENCODING **':'** ⟨*string*⟩

⟨*datum_group*⟩→ DATUM_GROUP **':'** ⟨*string*⟩

⟨*datum_group_name*⟩→ DATUM_GROUP_NAME **':'** ⟨*string*⟩

⟨*group_empty_as*⟩→ GROUP_EMPTY_AS **':'** ⟨*string*⟩ /* may be a SQL statement */

⟨*group_all_as*⟩→ GROUP_ALL_AS **':'** ⟨*string*⟩/* may be a SQL statement */

⟨*chunk_size*⟩→ CHUNK_SIZE **':'** INT

⟨*null_as*⟩→ NULL_AS **':'** ⟨*string*⟩

⟨*separator*⟩→ SEPARATOR **':'** CHAR

⟨*nb_datums*⟩→ NB_DATUMS **':'** INT

⟨*nb_atts*⟩→ NB_ATTS **':'** INT

⟨*datums*⟩→ DATUMS **':'** ⟨*datums_list*⟩

⟨*datums_list*⟩→ ⟨*datum*⟩ | ⟨*datums_list*⟩ ⟨*datum*⟩

⟨*datum*⟩→ ⟨*logical*⟩ ⟨*admin*⟩ ⟨*word*⟩

⟨*logical*⟩→ INT STRING

⟨*admin*⟩→ ⟨*admin_flags*⟩ INT INT INT INT ⟨*string*⟩

⟨*admin_flags*⟩→ INT ⟨*admin_flags*⟩ **'|'** INT

⟨*word*⟩→ ⟨*word_flags*⟩ INT INT

⟨*word_flags*⟩→ INT | ⟨*word_flags*⟩ **'|'** INT

⟨*string*⟩→ NULL_VALUE | STRING

⟨*name*⟩→ NULL_VALUE | NAME

45. Implementation.

Here we will describe the actual implementation of the functions. We start by the private header.

⟨ _corr.h 45 ⟩ ≡

```
#ifndef _KERGIS_CORR_H
#define _KERGIS_CORR_H
#include <assert.h>
#include <errno.h>
#include "ksys/endian.h"
#include "gis.h"
#include "corr.h"
#include "corr/dbf.h"
#include "corr/psql.h"
#include "corr/stdin.h"
#include "corr/stdout.h"
  ⟨ Private macros 51 ⟩
  ⟨ Private structures 49 ⟩
  ⟨ Private prototypes 55 ⟩
#endif  /* _KERGIS_CORR_H */
```

46. Opening and closing.

Depending on the *mode*, the file is opened if it exists or created. The policy on opening is handled by *G_open*.

The library suppports the concept of an anonymous correspondence, that is a transient, ephemeral object that will not be written in the GISDATABASE. To obtain this, one just gives the *NULL* pointer as a name.

If the files have to be created, they are created in the current mapset.

If the files already exist, they are searched according to SEARCH_PATH.

```
⟨ Set mapset and creation flag 46 ⟩ ≡
  mapset ← Λ;
  if (mode ≡ G_O_WRONLY) creation ← 1;
  else creation ← 0;
  if (Corr→name ≠ Λ ∧ mode ≠ G_O_WRONLY) {
    mapset ← G_find_file2(CORR_HEAD_SUBELT, Corr→name, "");
    if (mapset ≡ Λ) {
      if (mode ≡ G_O_RDONLY) return G_ENOENT;
      else creation ← 1;
    }
  }
  if (mapset ≡ Λ) mapset ← G_mapset( );
```

This code is used in section 47.

47. There are several files opened. First, the **C_Head** is filled, whether taking the values from the element of the gisdatabase if a *name* was given or from the values passed. If everything went fine, the **C_Corr** is filled and the open cascade to the driver routine.

```
⟨ open.c 47 ⟩ ≡
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "_corr.h"
  int C_open(struct C_Corr *Corr, int mode)
  {
    int status ← G_SUCCESS;      /* return value */
    int fd ← −1;
    char *mapset;      /* pointer to internal GISLIB name, don't free! */
    int creation;      /* G_O_RDWR case */
    Corr→handler ← −1;
    ⟨ Set mapset and creation flag 46 ⟩
    ⟨ Allocate and init new _C_Corr; goto end if no handler left or problem 52 ⟩
    ⟨ Open head if not anonymous and init DFile 53 ⟩
    ⟨ Set partition conversion 60 ⟩
    ⟨ Request driver to init or goto end 61 ⟩
    ⟨ Init fields conversions 67 ⟩
  end:
    if (status) (void) C_close(Corr);
    return status;
  }
```

48. The internal handling is done via an internal structure *_C_Corr*, that is accessed according to the *handler* set in the *Corr* struct that will be used on every call of the library.

Since the back-ends will handle the dirty stuff, we define a structure to communicate with them.

The *flags* are under the responsability of the back-ends. The least significant 16 bits are reserved for the `libcorr`, higher are used by the back-ends.

```
⟨ Public structures 21 ⟩ +≡
#define C_DUMPING °1
#define C_RESTORING °2
#define C_MODIFIED °4
#define C_ICONV_INVALID °10
#define C_DATE_INVALID °20
#define C_INVALID_TRANSACTION °40
#define C_FLAGS_SHIFT 16 /* back-end flags are shifted left this amount */
  struct C_DFile {
    int handler; /* internal to the driver */
    unsigned long flags;
    int mode;
    struct P_Map *Mapping;
    struct C_Head *Head;
    struct C_Data *Data;
  };
```

49.

```
⟨ Private structures 49 ⟩ ≡
  struct _C_Corr {
    struct C_DFile DFile;
    FILE *hf;      /* head description file */
    FILE *df;      /* data description file */
    unsigned long to_be_restored;      /* set to the number of atts to restore */
    unsigned long dump_retrieved;
    int *fields_convs;      /* an array of flags for fields conversions */
    iconv_t cd_in;      /* codeset converter descriptor when reading file */
    iconv_t cd_out;      /* codeset converter descriptor when writing file */
    int (*open)(struct C_DFile *);
    int (*close)(struct C_DFile *);
    int (*get_image)(struct C_DFile *, struct C_Image *);
    int (*set_image)(struct C_DFile *, struct C_Image *);
    int (*ioctl)(struct C_DFile *, unsigned long, void *argp);
  };
#ifdef HIC
#define EXTERN
#else
#define EXTERN extern
#endif      /* we define a maximum number of group of files to handle */
#define _MAX_HANDLERS 16      /* minimum value of OPEN_MAX */
  EXTERN
      int _C_nb_handlers;      /* nb of handlers actually in use */
      EXTERN
          struct _C_Corr *_C_handlers[_MAX_HANDLERS];
#define _HANDLER (_C_handlers[Corr→handler])
#define _DFILE_HANDLER→DFile
#define _FIELDS_CONVS_HANDLER→fields_convs
#define _CD_IN_HANDLER→cd_in
#define _CD_OUT_HANDLER→cd_out
#define _OPEN( )(*_HANDLER→open) (&_HANDLER→DFile)
#define _CLOSE( )(*_HANDLER→close) (&_HANDLER→DFile)
#define _GET_IMAGE(Image)(*_HANDLER→get_image) (&_HANDLER→DFile, Image)
#define _SET_IMAGE(Image)(*_HANDLER→set_image) (&_HANDLER→DFile, Image)
#define _IOCTL(request, argp)(*_HANDLER→ioctl) (&_HANDLER→DFile, request, argp)
```

See also section 69.

This code is used in section 45.

50. HIC will be defined in an object that needs to be here at last, that is in the closing routine object file.

51. Since we need to ensure that the *handler* set is a correct one, we define a macro.

```
⟨ Private macros 51 ⟩ ≡
#define _CHECK_HANDLER if (Corr→handler < 0 ∨ Corr→handler > _MAX_HANDLERS ∨ _C_handlers[Corr→handler] ≡
      Λ) return C_EBADHANDLER
#define _CHECK_DUMPING if (_DFILE.flags & C_DUMPING)return C_EBUILK
#define _CHECK_RESTORING if (_DFILE.flags & C_RESTORING)return C_EBUILK
```

This code is used in section 45.

52. If there is no handler left, we can bail out.

```
⟨ Allocate and init new _C_Corr; goto end if no handler left or problem 52 ⟩ ≡
  if (_C_nb_handlers ≥ _MAX_HANDLERS) return C_EOPENMAX;
  for (Corr→handler ← 0; Corr→handler < _MAX_HANDLERS ∧ _C_handlers[Corr→handler] ≠ Λ; ++Corr→handler)
    ;      /* calloc ensure default values */
  if ((_C_handlers[Corr→handler] ← (struct _C_Corr *) calloc(1, sizeof(struct _C_Corr))) ≡ Λ) {
    Corr→handler ← −1;
    return errno ≪ G_ERROR_SHIFT;
  }
  ++_C_nb_handlers;
```

This code is used in section 47.

53. We will open not a descriptor but a file to be able to use *fgets*.

```
⟨ Open head if not anonymous and init DFile 53 ⟩ ≡
  if (Corr→name ≠ Λ) {
    errno ← 0;
    if ((fd ← G__open(CORR_HEAD_SUBELT, Corr→name, mapset, mode)) < 0) {
      G_msg(G_EOPEN, G_Errors, G_WARNING);
      status ← errno ? errno ≪ G_ERROR_SHIFT : C_EOPEN;
      goto end;
    }
    _HANDLER→hf ← fdopen(fd, G_fd_to_file_mode(mode));
  }
  else _HANDLER→hf ← Λ;      /* anonymous */
  ⟨ Check or init correspondence file 54 ⟩
  _DFILE.Mapping ← &Corr→Mapping;
  _DFILE.Head ← &Corr→Head;
  _DFILE.Data ← &Corr→Data;
  _DFILE.mode ← mode;      /* save mode */
  _CD_IN ← (iconv_t) − 1;
  _CD_OUT ← (iconv_t) − 1;
  ⟨ Init pointers to driver functions 59 ⟩
```

This code is used in section 47.

54. If the anonymous correspondence is requested, the caller must have filled the correct information in the *Head*. The needed informations are back-end dependent.

```
⟨ Check or init correspondence file 54 ⟩ ≡
  if (Corr→name ≠ Λ) {
    struct stat buf;
    if (fstat(fd, &buf) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if (buf.st_size) {
      if ((status ← C_read_head(_HANDLER→hf, Corr))) goto end;
    }
    else if (_DFILE.mode ≡ G_O_RDONLY) {
      status ← C_EEMPTY;
      goto end;
    }
    else      /* creating */
      (void) C_write_head(_HANDLER→hf, Corr);
  }
```

This code is used in section 53.

55. Reading and writing the head and data description files.

The head and data are simple ascii (ascii for the labels, text for the values) file.

All values not given will have the *Default_head* or *Default_data* ones.

Since all these, even if stored in distinct files, are description of the whole, there is no need to try to implement two distinct parsers (this is not easy due to namespace collisions by the way). So here is a description of the syntax.

⟨ Private prototypes 55 ⟩ ≡

```
int _C_yyparse(void);
```

See also sections 74 and 79.

This code is used in section 45.

56.

```
⟨ head.c 56 ⟩ ≡
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "_corr.h"
  extern FILE *_C_yyin;
  extern int _C_lineno;
  struct C_Head *_Head;      /* to communicate with the parser */
  const struct C_Head Default_head ← {Λ, Λ, Λ, Λ, Λ, KT_ICONV_UTF_8, Λ, Λ, Λ, Λ, Λ, '|', C_MAX_NB_ATTS};
  int C_read_head(FILE *fp, struct C_Corr *Corr)
  {
    int status ← G_SUCCESS;
    Corr→Head ← Default_head;
    _Head ← &Corr→Head;
    ⟨ Restart the lexer 57 ⟩
    if (_C_yyparse()) status ← C_EBADHEADFORMAT;
    return status;
  }
  int C_write_head(FILE *fp, struct C_Corr *Corr)
  {
    (void) fprintf(fp, "DRIVER: %s\n", (Corr→Head.driver ≠ Λ) ? Corr→Head.driver : "NULL");
    (void) fprintf(fp, "PARTITION: ");
    if (Corr→Head.partition ≠ Λ) fprintf(fp, "\"%s\"\n", Corr→Head.partition);
    else fprintf(fp, "NULL\n");
    (void) fprintf(fp, "CLUSTER: ");
    if (Corr→Head.cluster ≠ Λ) (void) fprintf(fp, "\"%s\"\n", Corr→Head.cluster);
    else (void) fprintf(fp, "NULL\n");
    (void) fprintf(fp, "DBASE: ");
    if (Corr→Head.dbase ≠ Λ) (void) fprintf(fp, "\"%s\"\n", Corr→Head.dbase);
    else (void) fprintf(fp, "NULL\n");
    (void) fprintf(fp, "TABLE: ");
    if (Corr→Head.table ≠ Λ) fprintf(fp, "\"%s\"\n", Corr→Head.table);
    else fprintf(fp, "NULL\n");
    (void) fprintf(fp, "ENCODING: ");
    if (Corr→Head.encoding ≠ Λ) fprintf(fp, "\"%s\"\n", Corr→Head.encoding);
    else fprintf(fp, "NULL\n");
    (void) fprintf(fp, "DATUM_GROUP: ");
    if (Corr→Head.datum_group ≠ Λ) fprintf(fp, "\"%s\"\n", Corr→Head.datum_group);
    else fprintf(fp, "NULL\n");
    (void) fprintf(fp, "DATUM_GROUP_NAME: ");
    if (Corr→Head.datum_group_name ≠ Λ) fprintf(fp, "\"%s\"\n", Corr→Head.datum_group_name);
    else fprintf(fp, "NULL\n");
    (void) fprintf(fp, "GROUP_EMPTY_AS: ");
    if (Corr→Head.group_empty_as ≠ Λ) (void) fprintf(fp, "\"%s\"\n", Corr→Head.group_empty_as);
    else (void) fprintf(fp, "NULL\n");
    (void) fprintf(fp, "GROUP_ALL_AS: ");
    if (Corr→Head.group_all_as ≠ Λ) (void) fprintf(fp, "\"%s\"\n", Corr→Head.group_all_as);
    else (void) fprintf(fp, "NULL\n");
```

```
    (void) fprintf(fp, "NULL_AS: ");
    if (Corr→Head.null_as ≠ Λ) (void) fprintf(fp, "\"%s\"\n", Corr→Head.null_as);
    else (void) fprintf(fp, "NULL\n");
    (void) fprintf(fp, "SEPARATOR: '%c'\n", Corr→Head.separator);
    (void) fprintf(fp, "CHUNK_SIZE: %lu\n", Corr→Head.chunk_size);
    return G_SUCCESS;
  }
```

57. The library shall handle multiple files. The problem is with the lexer, if something goes wrong: scanning will be stopped in an internal buffer, before the end of the buffer, and modifying *_C_yyin* will not suffice. *lex(1)* will resume the scanning in the buffer before requesting for a new block of data from *_C_yyin*. Hence we need to flush the buffer. Actually, the code is in the `lexer.l` file, and is here for *flex(1)*. But we put here the comments and caveats to record the problem.

```
⟨ Restart the lexer 57 ⟩ ≡
  _C_lineno ← 1;
  _C_yyin ← fp;
```

This code is used in sections 56 and 58.

58. The data is parsed in the same way.

```
⟨ data.c 58 ⟩ ≡
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "_corr.h"
  extern FILE *_C_yyin;
  extern int _C_lineno;
  unsigned short _nb_datums;      /* to verify afterwards */
  struct C_Data *_Data;      /* to communicate with the parser */
  int C_read_data(FILE *fp, struct C_Corr *Corr)
  {
    int status ← G_SUCCESS;
    Corr→Data.nb_datums ← 0;
    Corr→Data.nb_atts ← 0;
    Corr→Data.Datums ← Λ;
    _Data ← &Corr→Data;
    ⟨ Restart the lexer 57 ⟩
    if (_C_yyparse()) status ← C_EBADDATAFORMAT;
    if (_nb_datums ≠ Corr→Data.nb_datums) G_msg(C_WBADNBDATUMS, C_Warnings, G_WARNING);
    return status;
  }
  int C_write_data(FILE *fp, struct C_Corr *Corr)
  {
    int i;
    (void) fprintf(fp, "NB_DATUMS: %d\n", Corr→Data.nb_datums);
    (void) fprintf(fp, "NB_ATTS: %lu\n", Corr→Data.nb_atts);
    (void) fprintf(fp, "DATUMS: \n");
    for (i ← 0; i < Corr→Data.nb_datums; i++) {
      fprintf(fp, "\t%d \"%s\"", Corr→Data.Datums[i].Logical.type, Corr→Data.Datums[i].Logical.name);
      fprintf(fp, "\t%d %lu %ld %ld %ld", Corr→Data.Datums[i].Admin.flags,
          Corr→Data.Datums[i].Admin.size, Corr→Data.Datums[i].Admin.shift,
          Corr→Data.Datums[i].Admin.unit, Corr→Data.Datums[i].Admin.scale);
      if (Corr→Data.Datums[i].Admin.default_value ≡ Λ) fprintf(fp, " NULL");
      else fprintf(fp, " \"%s\"", Corr→Data.Datums[i].Admin.default_value);
      fprintf(fp, "\t%u %hu %hu\n", Corr→Data.Datums[i].Word.flags, Corr→Data.Datums[i].Word.significand,
          Corr→Data.Datums[i].Word.exp_max);
    }
    return G_SUCCESS;
  }
```

59. When we go to end, we invoke the driver specific close function. Everything needs to be set before !

```
⟨ Init pointers to driver functions 59 ⟩ ≡
  if (¬strcmp(Corr→Head.driver, "dbf")) ⟨ Init the dbf back-end 83 ⟩
  else if (¬strcmp(Corr→Head.driver, "psql")) ⟨ Init the psql back-end 84 ⟩
  else if (¬strcmp(Corr→Head.driver, "stdin")) ⟨ Init the stdin back-end 85 ⟩
  else if (¬strcmp(Corr→Head.driver, "stdout")) ⟨ Init the stdout back-end 86 ⟩
  else {
    status ← C_EBADDRIVER;
    goto end;
  }
```

This code is used in section 53.

60. The mapping between group ids and group names is the purpose of the PARTLIB.

The CORRLIB is not designed to write PARTLIB stuff: it only reads if necessary. If the mapping is created while attributes are created too, first open in write mode the mapping with the PARTLIB and update the mapping via the PARTLIB before making work via the CORRLIB if the latter depends on the updated information.

The mapping may be necessary in read cases if DATUM_GROUP_NAME is set. But the mapping is only opened for reading if *Corr→Mapping.name* $\neq \Lambda$.

Of course, the name of the mapping must match the name of the partition set in *Corr→Head*.

```
⟨ Set partition conversion 60 ⟩ ≡
  if (Corr→Head.datum_group_name ≠ Λ ∧ Corr→Mapping.name ≠ Λ) {
    if (strcmp(Corr→Mapping.name, Corr→Head.partition) ≠ 0) {
      status ← C_EPARTNAME;
      goto end;
    }
    if ((status ← P_open(&Corr→Mapping, mode)) ≠ G_SUCCESS) {
      G_msg(status, P_Errors, G_INFO);
      status ← C_EPART;
      goto end;
    }
  }
```

This code is used in section 47.

61. The informations being filled, we set our *Driver* structure to the correct one, and invoke the driver dependent open routine.

```
⟨ Request driver to init or goto end 61 ⟩ ≡
  if ((status ← _OPEN( ))) goto end;
```

This code is used in section 47.

62. Closing is fairly obvious. Some supplementary checks shall be done and will be added in the future.

```
⟨ close.c 62 ⟩ ≡
#include <unistd.h>
#define HIC
#include "_corr.h"
  int C_close(struct C_Corr *Corr)
  {
    int status ← G_SUCCESS;      /* be able to be safely called when not opened (interrupted user program) */
    if (Corr→handler ≡ −1) return G_SUCCESS;
    _CHECK_HANDLER;

    if (_HANDLER→hf ≠ Λ) {
      if (fclose(_HANDLER→hf) < 0) status ← errno ≪ G_ERROR_SHIFT;
    }      /* we may be called before driver pointers to routine are set */
    if (_HANDLER→close ≠ Λ) {
      status ← _CLOSE( );
    }      /* check status */
    if (_DFILE.flags & C_ICONV_INVALID) G_msg(C_WICONVINVALID, C_Warnings, G_WARNING);
    if (_DFILE.flags & C_DATE_INVALID) G_msg(C_WDATEINVALID, C_Warnings, G_WARNING);
    ⟨ Close iconv converters 68 ⟩
    ⟨ Close partition mapping 63 ⟩
    free(_C_handlers[Corr→handler]);
    _C_handlers[Corr→handler] ← Λ;
    Corr→handler ← −1;
    −− _C_nb_handlers;
    return status;
  }
```

63. If the partition mapping was open, we need to close it first. We'd better not be too smart and try to play tricks with *Corr→Mapping.handler* so we use the same condition as for opening.

```
⟨ Close partition mapping 63 ⟩ ≡
  if (Corr→Head.datum_group_name ≠ Λ) status ← P_close(&Corr→Mapping);
```

This code is used in section 62.

64. Image manipulations.

A **C_Image** struct must have been allocated first.

When requesting an image, the group id must be set. Then we simply query for the multiplicity and the att identifiers. The attributes will be retrieved by chunks of *Head.chunk_size*, that is some window of the image is retrieved. *offset* is set, as well as *nb_atts_here*. If $multiplicity \leq Head.chunk\_size \wedge offset \equiv 0$ then $nb\_atts\_here \equiv multiplicity$. If there is more, you need to call *C_get_image*, updating *offset* to retrieve the remaining attributes. You are done if $offset + nb\_atts\_here \equiv multiplicity$.

So *offset* shall always be set to some meaningful value

```
⟨ image.c 64 ⟩ ≡
#include "_corr.h"
  int C_get_image(const struct C_Corr *Corr, struct C_Image *Image)
  {
    int i;
    int status ← G_SUCCESS;
    struct C_Att *Att;
    _CHECK_HANDLER;
    _CHECK_RESTORING;
    if (Image→group ≡ P_GROUP_ALL ∧ ¬(_DFILE.flags & C_DUMPING)) {
      _DFILE.flags |= C_DUMPING;
      _HANDLER→dump_retrieved ← 0;
    }
    if ((status ← C_clean_atts(Corr→Data.nb_datums, Image))) goto end;
    if ((status ← _GET_IMAGE(Image))) goto end;
    for (i ← 0; i < Image→nb_atts_here; i++) {
      Att ← &Image→Atts[i];
      ⟨ Convert fields for in 73 ⟩
    }
    if (_DFILE.flags & C_DUMPING) {
      _HANDLER→dump_retrieved += Image→nb_atts_here;
      if (_HANDLER→dump_retrieved ≡ Corr→Data.nb_atts) {      /* done */
        _HANDLER→dump_retrieved ← 0;
        _DFILE.flags &= ∼C_DUMPING;
      }
    }
  end: return status;
  }
  ⟨ Other image functions 65 ⟩
```

65. Setting an image is quite simple: we just pass attribute by attribute.

```
⟨ Other image functions 65 ⟩ ≡
  int C_set_image(const struct C_Corr *Corr, struct C_Image *Image)
  {
    unsigned long i;
    int status ← G_SUCCESS;
    struct C_Att *Att;
    _CHECK_HANDLER;
    _CHECK_DUMPING;
    if (Image→group ≡ P_GROUP_ALL ∧ ¬(_DFILE.flags & C_RESTORING)) {
      if (Corr→Data.nb_atts) {
        status ← C_ENOTEMPTY;
        goto end;
      }
      if (Image→offset) {      /* we must begin at beginning */
        status ← C_ERESTORE;
        goto end;
      }
      _DFILE.flags |= C_RESTORING;
      _HANDLER→to_be_restored ← Image→multiplicity;
    }
    for (i ← 0; i < Image→nb_atts_here; i++) {
      Att ← &Image→Atts[i];
      ⟨ Convert fields for out 72 ⟩
    }
    if ((status ← _SET_IMAGE(Image))) goto end;
    if (_DFILE.flags & C_RESTORING) {
      if (_HANDLER→to_be_restored ≡ Corr→Data.nb_atts) {
        _DFILE.flags &= ∼C_RESTORING;
        _HANDLER→to_be_restored ← 0;
      }
    }
  end: return status;
  }
```

This code is used in section 64.

66. Fields conversions: `iconv` and other related transformations.

Conversions between two codesets is done with *iconv* functions (a POSIX extension, incorporated directly in the libc or that needs to be provided externally).

All KerGIS values are passed as `UTF-8` strings. Depending on the *encoding* specified, there are two descriptors to open.

This operation is done after the back-end dependent part, since the back-end may set some defaults (this is the case of the `dbf` one).

We could simply pass every field in a **C_Att** to the `iconv` function, but since this is costy we will try to improve a little.

There is only one kind of *C_Datum* that always need a conversion (perhaps a null one): `C_TEXT_L`. In all other cases, since values shall be all ASCII (numbers) or raw no conversion shall be attempted...except when a character is encoded in more than one byte!

The handling of wyde characters is not done in this current implementation so the test is unlikely to be of some real use. But for the record and the logics we will put it here nevertheless.

So we will create a *fields_convs* that is an array of flags telling whether we need to convert it or not. If we must go from byte based to wyde based, every field has to be converted. At the moment no attempt is done to try to convert locale representation of numerical values to standard ones (for us C). This will be handled some day (perhaps) via this array. At the moment, one can use a byte encoding for text, and absolutely refrain from trying to use comma for number separation or local date...

67. To start with, we open the converters for text (i.e. `iconv` functionnality).

```
⟨ Init fields conversions 67 ⟩ ≡
  if ((_CD_IN ← iconv_open(KT_ICONV_UTF_8, Corr→Head.encoding)) ≡ (iconv_t) − 1) {
    status ← errno ≪ G_ERROR_SHIFT;
    goto end;
  }
  if ((_CD_OUT ← iconv_open(Corr→Head.encoding, KT_ICONV_UTF_8)) ≡ (iconv_t) − 1) {
    status ← errno ≪ G_ERROR_SHIFT;
    goto end;
  }
  ⟨ Set fields_convs 70 ⟩
```

This code is used in section 47.

68. On closing, the reverse will be strictly true so put it here and pass to more important information.

```
⟨ Close iconv converters 68 ⟩ ≡
  if (_CD_IN ≠ (iconv_t) − 1) (void) iconv_close(_CD_IN);
  if (_CD_OUT ≠ (iconv_t) − 1) (void) iconv_close(_CD_OUT);
```

This code is used in section 62.

69. So at initialization, after allocating an array of ints matching the number of *C_Datum*s, we make one test to see if an ASCII one byte value is converted to more than one byte. If this is still one byte, the iconv is concerned only by text. If not, everything has to be converted (there could be some optimizations depending on the encoding: `US − ASCII` will remain itself).

```
⟨ Private structures 49 ⟩ +≡
#define _C_XF_ICONV °1
```

70.

```
⟨ Set fields_convs 70 ⟩ ≡
  {
    int i, need_iconv;
    if ((_FIELDS_CONVS ← (int *) malloc(Corr→Data.nb_datums * sizeof(int))) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    ⟨ Set need_iconv to −1, 0 or 1 depending on problem, byte or wyde promotion 71 ⟩
    for (i ← 0; i < Corr→Data.nb_datums; i++)
      _FIELDS_CONVS[i] ← (need_iconv ≡ 1 ∨ Corr→Data.Datums[i].Logical.type ≡ C_TEXT_L) ? _C_XF_ICONV :
          0;
  }
```

This code is used in section 67.

71. The test is fairly straightforward: we will pass the character `'A'` and read how much is left in the destination.

There are three cases: whether there was a problem so we will do nothing at all (*need_iconv* ← −1). Or conversion succeeded, but only one byte was needed, do nothing (*need_iconv* ← 0) or more were needed (*need_iconv* ← 1).

```
⟨ Set need_iconv to −1, 0 or 1 depending on problem, byte or wyde promotion 71 ⟩ ≡
  {
    char *dst;
    const char *src ← "A";
    size_t dstleft ← 4;
    size_t srcleft ← 1;
    if ((dst ← (char *) malloc(3)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    need_iconv ← 0;
    if (iconv(_CD_OUT, &src, &srcleft, &dst, &dstleft)) {
      G_msg(C_WBADICONV, C_Warnings, G_WARNING);
      need_iconv ← −1;
    }
    else if (dstleft ≠ 3) {
      G_msg(C_INEEDICONV, C_Infos, G_INFO);
      need_iconv ← 1;
    }
  }
```

This code is used in section 70.

72. OK, now we will be ready to convert input to our version (*UTF-8* for text to begin with), and to do the converse for output. We will work on images and will simply proceed all the fields needing this in all the atts.

```
⟨ Convert fields for out 72 ⟩ ≡
  {
    int j;
    char *s;
    for (j ← 0; j < Corr→Data.nb_datums; j++) {
      if (_FIELDS_CONVS[j] & _C_XF_ICONV) {
        if ((status ← _C_iconv(_CD_OUT, Att→fields[j], &s, &_DFILE.flags))) goto end;
        free(Att→fields[j]);
        Att→fields[j] ← s;
      }
    }
  }
```

This code is used in section 65.

73.

```
⟨ Convert fields for in 73 ⟩ ≡
  {
    int j;
    char *s;
    for (j ← 0; j < Corr→Data.nb_datums; j++) {
      if (_FIELDS_CONVS[j] & _C_XF_ICONV) {
        if ((status ← _C_iconv(_CD_IN, Att→fields[j], &s, &_DFILE.flags))) goto end;
        free(Att→fields[j]);
        Att→fields[j] ← s;
      }
    }
  }
```

This code is used in section 64.

74. The *_C_iconv* function, takes a converter descriptor, a pointer to char (a string) to convert, and a pointer to a pointer to char, i.e. a place where to store the value of the pointer to the translated string.

The function does the memory allocation for **dest*.

The *flags* pointer allows to set C_ICONV_INVALID is one or more chars were invalid for the target codeset. We do not pass a **C_DFile** structure to allow the function to be called in a not correspondence environment.

We simply try to reallow twice the place, and we start again from the offset in the destination buffer where converted chars have already been put.

Since it will be a nuisance to have a warning issued every time there is an invalid character for the target codeset, we set a flag in the _DFILE.*flags* to indicate that there was one or more problem. The warning will be issued once at closing.

```
⟨ Private prototypes 55 ⟩ +≡
  int _C_iconv(iconv_t cd, const char *src, char **dest, unsigned long *flags);
```

75.

```
⟨ locale.c 75 ⟩ ≡
#include <stdlib.h>
#include <string.h>
#include "_corr.h"
  int _C_iconv(iconv_t cd, const char *src, char **dest, unsigned long *flags)
  {
    int status ← G_SUCCESS;
    const char *s;      /* follow the source */
    char *p;      /* temporary */
    char *dst;
    size_t srcleft, dstleft;
    size_t size_dest;
    size_t invalid;
    size_t delta;      /* offset in the converted string */
    ⟨ Handle the Λ and '\0' cases 77 ⟩
    s ← src;
    ⟨ Set srcleft to src length 78 ⟩
    size_dest ← srcleft + 1;      /* handle terminating null byte */
    dstleft ← size_dest;
    delta ← 0;
    invalid ← 0;
    *dest ← Λ;      /* will realloc */
    for ( ; srcleft; delta ← dst − *dest) {
      ⟨ Realloc room in destination and reset the offset and size 76 ⟩
      invalid ← iconv(cd, &s, &srcleft, &dst, &dstleft);
      if (srcleft ∧ errno ≠ E2BIG) {
        status ← errno ≪ G_ERROR_SHIFT;
        free(*dest);
        goto end;
      }
    }
    if (invalid ≡ (size_t) −1) {
      status ← errno ≪ G_ERROR_SHIFT;
      free(*dest);
      goto end;
    }
    else if (invalid) {
      G_msg(C_IICONV, C_Infos, G_INFO, src);
      *flags |= C_ICONV_INVALID;
    }

    *dst ← '\0';      /* null terminate string */
    delta ← dst − *dest;      /* the string length indeed */
    dst ← *dest;      /* back to beginning */      /* there may be unused space, so allocate only the used one */
    if ((*dest ← (char *) malloc(delta + 1)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      free(dst);
      goto end;
    }
    (void) strcpy(*dest, dst);      /* release unused space */
    free(dst);
```

```
end: return status;
}
```

76. We will realloc twice the previous space.

dst is pointing to the char after the last translated one. So in the buffer pointed to by $*dest$, the offset is $dst - *dest$. After reallocating—via a temporary pointer since, if it fails, we will have trash the value of $*dest$ without freeing it—, we reset *dst* at the correct offset, update the *size_dest* and *dstleft*, the sources values being already correct.

The room created must be the previous value of *size_dest* before doubling, since if there was an error, and there was room left, we shall not be here.

```
⟨ Realloc room in destination and reset the offset and size 76 ⟩ ≡
  dstleft += size_dest;
  size_dest ← 2 * size_dest;
  if ((p ← (char *) realloc(*dest, size_dest)) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    free(*dest);
    goto end;
  }
  *dest ← p;
  dst ← *dest + delta;
```

This code is used in section 75.

77. If the *src* passed is Λ, just allocate one byte for consistency, allowing callers to free *dest* without handling special cases.

```
⟨ Handle the Λ and '\0' cases 77 ⟩ ≡
  if (src ≡ Λ) {
    *dest ← Λ;
    goto end;
  }
  else if (*src ≡ '\0') {
    if ((*dest ← (char *) malloc(1)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    **dest ← '\0';
    goto end;
  }
```

This code is used in section 75.

78. Calling *strlen* costs, while doing its work is easy without the overhead of the function call.

```
⟨ Set srcleft to src length 78 ⟩ ≡
  {
    const char *p;
    for (p ← src, srcleft ← 0; *p; p++, ++srcleft) ;
  }
```

This code is used in section 75.

79. Memory allocation.
The core `libcorr` functions free memory allocated in *Image* when done. The back-end allocate memory for *Att*.

```
⟨ Private prototypes 55 ⟩ +≡
  int clean_image(struct C_Image *Image);
```

80.

```
⟨ memory.c 80 ⟩ ≡
#include <stdlib.h>
#include "_corr.h"
  int C_clean_atts(const int nb_datums, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    int i, j;
    for (i ← 0; i < Image→nb_atts_here; i++) {
      for (j ← 0; j < nb_datums; j++) free(Image→Atts[i].fields[j]);
      free(Image→Atts[i].fields);
    }
    free(Image→Atts);
    Image→Atts ← Λ;
    Image→nb_atts_here ← 0;
    return status;
  }
  ⟨ Other memory functions 81 ⟩
```

81. Reallocating the *Image→Atts* is done by freeing the individual *fields* in each att, freeing the *Atts* array, and allocating the vector *Image→Atts* and the new *fields*.

```
⟨ Other memory functions 81 ⟩ ≡
  int C_alloc_atts(const int nb_datums, struct C_Image *Image, int nb_atts)
  {
    int status ← G_SUCCESS;
    int i;
    if ((status ← C_clean_atts(nb_datums, Image))) goto end;
    if ((Image→Atts ← (struct C_Att *) calloc(nb_atts, sizeof(struct C_Att))) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }     /* this keep track of the size of the array */
    Image→nb_atts_here ← nb_atts;
    for (i ← 0; i < Image→nb_atts_here; i++) {
      if ((Image→Atts[i].fields ← (char **) calloc(nb_datums, sizeof(char *))) ≡ Λ) {
        (void) C_clean_atts(nb_datums, Image);
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
    }
  end: return status;
  }
```

See also section 82.

This code is used in section 80.

82. We allocate the requested structure and ensure that everything is zeroed.

```
⟨ Other memory functions 81 ⟩ +≡
  int C_alloc_image(struct C_Image **Image)
  {
    if ((*Image ← (struct C_Image *) calloc(1, sizeof(struct C_Image))) ≡ Λ)
      return errno ≪ G_ERROR_SHIFT;
    else return G_SUCCESS;
  }
  int C_free_image(const int nb_datums, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    if ((status ← C_clean_atts(nb_datums, Image))) goto end;
    free(Image);
    Image ← Λ;
  end: return status;
  }
```

83. The `dbf` back-end.

The `dbf` back-end has a dedicated sub-library. Initialization is fairly obvious.

⟨ Init the dbf back-end 83 ⟩ ≡

```
{
  _HANDLER→open ← C_dbf_open;
  _HANDLER→close ← C_dbf_close;
  _HANDLER→get_image ← C_dbf_get_image;
  _HANDLER→set_image ← C_dbf_set_image;
  _HANDLER→ioctl ← (int(*)(struct C_DFile *, unsigned long, void *argp))Λ;
}
```

This code is used in section 59.

84. The `psql` back-end.

The `psql` back-end has a dedicated sub-library. At the moment, the invocation of the user interface program `psql` is rough...

⟨ Init the psql back-end 84 ⟩ ≡

```
{
  _HANDLER→open ← C_psql_open;
  _HANDLER→close ← C_psql_close;
  _HANDLER→get_image ← C_psql_get_image;
  _HANDLER→set_image ← C_psql_set_image;
  _HANDLER→ioctl ← (int(*)(struct C_DFile *, unsigned long, void *argp))Λ;
}
```

This code is used in section 59.

85. The `stdin` back-end.

The `stdin` back-end has a dedicated sub-library. See it for the actual level of support...

⟨ Init the stdin back-end 85 ⟩ ≡

```
{
  _HANDLER→open ← C_stdin_open;
  _HANDLER→close ← C_stdin_close;
  _HANDLER→get_image ← C_stdin_get_image;
  _HANDLER→set_image ← C_stdin_set_image;
  _HANDLER→ioctl ← (int(*)(struct C_DFile *, unsigned long, void *argp))Λ;
}
```

This code is used in section 59.

86. The `stdout` back-end.

The `stdout` back-end has a dedicated sub-library. See it for the actual level of support...

⟨ Init the stdout back-end 86 ⟩ ≡

```
{
  _HANDLER→open ← C_stdout_open;
  _HANDLER→close ← C_stdout_close;
  _HANDLER→get_image ← C_stdout_get_image;
  _HANDLER→set_image ← C_stdout_set_image;
  _HANDLER→ioctl ← (int(*)(struct C_DFile *, unsigned long, void *argp))Λ;
}
```

This code is used in section 59.

87. Informations.
All the messages are put in an array and in a special section so that internationalisation can be achieved easily.

```
⟨ Infos macros 87 ⟩ ≡
#define C_IICONV 64
#define C_INEEDICONV 65
#define C_dbf_IHEADER 66
#define C_IMAX 66
```
This code is used in section 9.

88.
```
⟨ infos.c 88 ⟩ ≡
#include <stdlib.h>  /* at least NULL */
#include "_corr.h"
  ⟨ C lang infos formats 89 ⟩
  static char *_C_istrerror(int msgnum);
  static struct G_Msgs _C_Infos ← {"libcorr", C_IMAX, _C_istrerror};
  const struct G_Msgs *const C_Infos ← &_C_Infos;
  static char *_C_istrerror(int msgnum)
  {
    return infos[msgnum − G_ERROR_MAX − 1];
  }
```

89. Informations formats for the required C locale.

```
⟨ C lang infos formats 89 ⟩ ≡  /* must be kept in sync with the infos macros! */
  static char *infos[] ← {"the␣string␣'%s'␣has␣incorrect␣character␣for␣codeset\n",
      "byte␣to␣wyde␣promotion␣needed:␣every␣field␣will␣be␣converted\n",
      "dbf:␣the␣header␣version␣is:␣%#x␣'(%s)'\n", };
```
This code is used in section 88.

90. Warnings.
All the messages are put in an array and in a special section so that internationalisation can be achieved easily.

```
⟨ Warnings macros 90 ⟩ ≡
#define C_WICONVINVALID 64
#define C_WDATEINVALID 65
#define C_WVALUECUT 66
#define C_WPARSE 67
#define C_WBADICONV 68
#define C_WCLIENTEXIT 69
#define C_WBADNBDATUMS 70
#define C_WPARSER 71
#define C_psql_WSETTYPE 72
#define C_dbf_WTYPE 73
#define C_dbf_WSETTYPE 74
#define C_dbf_WTYPELENGTH 75
#define C_dbf_WDATE 76
#define C_dbf_WCODEPAGE 77
#define C_WMAX 77
```

This code is used in section 9.

91.

```
⟨ warnings.c 91 ⟩ ≡
#include <stdlib.h>      /* at least NULL */
#include "_corr.h"
  ⟨ C lang warnings formats 92 ⟩
  static char *_C_wstrerror(int msgnum);
  static struct G_Msgs _C_Warnings ← {"libcorr", C_WMAX, _C_wstrerror};
  const struct G_Msgs *const C_Warnings ← &_C_Warnings;
  static char *_C_wstrerror(int msgnum)
  {
    return warnings[msgnum − G_ERROR_MAX − 1];
  }
```

92. Warnings formats for the required C locale.

```
⟨ C lang warnings formats 92 ⟩ ≡      /* must be kept in sync with the warnings macros! */
  static char *warnings[] ← {"the strings converted had invalid chars for codeset\n",
      "some dates were not in ISO 8601 format\n",
      "the string exceeds the space allocated. Cut\n",
      "scan error: more than %d fields passed in description\n",
      "iconv test failed: no conversion will be attempted\n",
      "back-end client has exited\n",
      "the nb of datums was incorrectly specified in data file\n","line %d: %s at %s\n",
    /* psql */
  "psql: the KerGIS type: %.3o will be set as raw 'Text' in PG\n",
      /* dbf */
  "dbf: the type: %c will be treated as raw data\n",
      "dbf: the KerGIS type: %x will be set as raw 'C' in dbf file\n",
      "dbf: the size of the word is too big: significand: %d, exp_max: %d\n",
      "dbf: the date '%s' is not in ISO 8601 format. Left alone...\n",
      "dbf: the code page specified (%#x) is incorrect(should be CP437)\n", };
```

This code is used in section 91.

93. Error handling.

⟨ Errors macros 93 ⟩ ≡
#define C_ENOTIMPLEMENTED 64
#define C_ENONSENSE 65
#define C_EBADHEADFORMAT 66
#define C_EBADDRIVER 67
#define C_EDRIVER 68
#define C_EPART 69
#define C_EBADHANDLER 70
#define C_EMODE 71
#define C_EOPENMAX 72
#define C_EEMPTY 73
#define C_EBUILK 74
#define C_ENOTEMPTY 75
#define C_ERESTORE 76
#define C_EOPEN 77
#define C_EPFINDGROUPNAME 78
#define C_EBADDATAFORMAT 79
#define C_EPARTNAME 80
#define *C_psql_ECMD* 81
#define *C_psql_EALLOCATT* 82
#define *C_psql_EBADFORMAT* 83
#define *C_psql_EREAD* 84
#define *C_psql_EMISSINGVALUES* 85
#define *C_psql_EMISSINGSELECTION* 86
#define *C_psql_EOPENCACHE* 87
#define *C_dbf_EREAD* 88
#define *C_dbf_EFORMAT* 89
#define *C_dbf_ETYPE* 90
#define *C_dbf_EEMPTY* 91
#define *C_dbf_EWRITE* 92
#define *C_dbf_EWRONGHEADER* 93
#define *C_dbf_ELOST* 94
#define *C_dbf_EREADRANGE* 95
#define *C_dbf_EMISSINGVALUES* 96
#define *C_stdin_EBADDATA* 97
#define *C_stdin_EBADGROUP* 98
#define C_EMAX 98

This code is used in section 9.

94.

```
⟨ errors.c 94 ⟩ ≡
#include <stdlib.h>      /* at least NULL */
#include "_corr.h"
  ⟨ C lang errors formats 95 ⟩
  static char *_C_estrerror(int msgnum);
  static struct G_Msgs _C_Errors ← {"libcorr", C_EMAX, _C_estrerror};
  const struct G_Msgs *const C_Errors ← &_C_Errors;
  static char *_C_estrerror(int msgnum)
  {
    if (msgnum ≤ G_ERROR_MAX) G_msg(msgnum, G_Errors, G_FATAL);
    return errors[msgnum − G_ERROR_MAX − 1];
  }
```

95. Errors formats for the required C locale.

```
⟨ C lang errors formats 95 ⟩ ≡
  static char *errors[] ← {"not␣implemented␣yet\n", "this␣should␣not␣happen!\n",
      "format␣of␣correspondence␣header␣file␣is␣incorrect\n",
      "no␣driver␣matching␣driver␣name\n", "error␣in␣calling␣driver␣open␣routine\n",
      "error␣in␣setting␣mapping␣between␣group␣ids␣and␣names\n",
      "the␣handler␣id␣passed␣is␣incorrect\n",
      "file␣not␣open␣in␣correct␣mode␣for␣requested␣operation\n",
      "too␣many␣files␣already␣handled\n", "the␣correspondence␣head␣file␣is␣empty\n",
      "the␣table␣is␣currently␣dumping/restoring.\n""No␣read␣nor␣write␣until␣end\n",
      "restore␣can␣not␣be␣done,␣dataset␣not␣empty\n",
      "restoration␣must␣start␣at␣the␣beginning!\n", "unable␣to␣open␣head␣file\n",
      "unable␣to␣find␣group␣name␣associated\n",
      "format␣of␣correspondence␣data␣file␣is␣incorrect\n", "mismatch␣between␣pa\
      rtition␣name␣in␣Corr->Head.partition""␣and␣Corr->Mapping.name\n",
    /* psql */
  "psql:␣the␣command␣passed␣to␣the␣system␣failed!\n",
      "psql:␣failed␣to␣allocate␣memory␣for␣att\n",
      "psql:␣the␣format␣of␣the␣psql␣output␣doesn't␣match␣my␣expectations\n",
   "psql:␣failed␣to␣read␣the␣requested␣att\n", "psql:␣dbase␣and␣table␣must␣be␣set!\n",
      "psql:␣one␣of␣datum_group␣or␣datum_group_name␣(even␣dummy)␣shall␣be␣set!\n",
      "psql:␣can␣not␣open␣temporary␣cache␣file!\n",     /* dbf */
  "dbf:␣failed␣to␣read␣the␣requested␣bytes!\n",
      "dbf:␣this␣dbf␣version␣is␣unknown␣to␣me!␣Will␣try...\n",
      "dbf:␣unknown␣field␣type!\n", "dbf:␣the␣file␣requested␣for␣read␣is␣empty!\n",
      "dbf:␣failed␣to␣write␣the␣requested␣bytes!\n", "dbf:␣the␣header␣is␣incorrect!\n",
      "dbf:␣I'm␣lost,␣the␣offset␣does␣not␣make␣sense!\n",
      "dbf:␣the␣record␣id␣requested␣for␣read␣is␣out␣of␣range\n",
      "dbf:␣at␣least␣Head->table␣shall␣be␣set!\n",     /* stdin */
  "stdin:␣whether␣Data␣structure␣is␣not␣set␣or␣incorrectly␣filled!\n",
      "stdin:␣only␣group␣P_GROUP_ALL␣makes␣sense\n", };
```

This code is used in section 94.

96. Index. Here is a list of the identifiers. Overstriked entries indicate the place of definition.

_C_Corr: 48, 49, 52.
_C_Errors: 94.
_C_estrerror: 94.
_C_handlers: 49, 51, 52, 62.
_C_iconv: 72, 73, 74, 75.
_C_Infos: 88.
_C_istrerror: 88.
_C_lineno: 56, 57, 58.
_C_nb_handlers: 49, 52, 62.
_C_Warnings: 91.
_C_wstrerror: 91.
_C_XF_ICONV: 69, 70, 72, 73.
_C_yyin: 56, 57, 58.
_C_yyparse: 55, 56, 58.
_CD_IN: 49, 53, 67, 68, 73.
_CD_OUT: 49, 53, 67, 68, 71, 72.
_CHECK_DUMPING: 51, 65.
_CHECK_HANDLER: 51, 62, 64, 65.
_CHECK_RESTORING: 51, 64.
_CLOSE: 49, 62.
_Data: 58.
_DFILE: 49, 51, 53, 54, 62, 64, 65, 72, 73, 74.
_FIELDS_CONVS: 49, 70, 72, 73.
_GET_IMAGE: 49, 64.
_HANDLER: 49, 53, 54, 62, 64, 65, 83, 84, 85, 86.
_Head: 56.
_IOCTL: 49.
_KERGIS_CORR_H: 45.
_MAX_HANDLERS: 49, 51, 52.
_nb_datums: 58.
_OPEN: 49, 61.
_SET_IMAGE: 49, 65.
accuracy: 5.
Admin: 35, 58.
anonymous: 2, 11.
argp: 49, 83, 84, 85, 86.
ASCII: 69.
att: 17.
Att: 39, 64, 65, 72, 73, 79.
attribute: 4.
Attributes: 2.
Atts: 40, 64, 65, 80, 81.
buf: 54.
C_Admin: 34, 35.
C_ALL_DISTINCT_A: 31.
C_alloc_atts: 19, 81.
C_alloc_image: 19, 82.
C_ASCII_W: 21.
C_Att: 3, 4, 19, 38, 40, 64, 65, 66, 81.
C_ATT_DELETED: 39.
C_ATT_MODIFIED: 39.
C_AUTO_INCREMENT_A: 31, 33.
C_BINARY_W: 21.
C_clean_atts: 19, 64, 80, 81, 82.
C_close: 15, 47, 62.
C_Corr: 13, 14, 15, 16, 17, 37, 47, 56, 58, 62, 64, 65.
C_Data: 4, 14, 36, 37, 48, 58.
C_DATE_INVALID: 48, 62.
C_DATE_L: 26, 28.
C_Datum: 4, 35, 36.
C_dbf_close: 83.
C_dbf_EEMPTY: 93.
C_dbf_EFORMAT: 93.
C_dbf_ELOST: 93.
C_dbf_EMISSINGVALUES: 93.
C_dbf_EREAD: 93.
C_dbf_EREADRANGE: 93.
C_dbf_ETYPE: 93.
C_dbf_EWRITE: 93.
C_dbf_EWRONGHEADER: 93.
C_dbf_get_image: 83.
C_dbf_IHEADER: 87.
C_dbf_open: 83.
C_dbf_set_image: 83.
C_dbf_WCODEPAGE: 90.
C_dbf_WDATE: 90.
C_dbf_WSETTYPE: 90.
C_dbf_WTYPE: 90.
C_dbf_WTYPELENGTH: 90.
C_DECIMAL_L: 29.
C_DFile: 48, 49, 74, 83, 84, 85, 86.
C_DIGITS_TO_BITS: 23.
C_DUMPING: 16, 48, 51, 64.
C_EBADDATAFORMAT: 58, 93.
C_EBADDRIVER: 59, 93.
C_EBADHANDLER: 51, 93.
C_EBADHEADFORMAT: 56, 93.
C_EBUILK: 51, 93.
C_EDRIVER: 93.
C_EEMPTY: 54, 93.
C_EMAX: 93, 94.
C_EMODE: 93.
C_ENONSENSE: 93.
C_ENOTEMPTY: 65, 93.
C_ENOTIMPLEMENTED: 93.
C_EOPEN: 53, 93.
C_EOPENMAX: 52, 93.
C_EPART: 60, 93.
C_EPARTNAME: 60, 93.
C_EPFINDGROUPNAME: 93.
C_ERESTORE: 65, 93.
C_Errors: 9, 94.

C_FLAGS_SHIFT: 48.
C_FLOAT_L: 29.
C_FLOAT_W: 21.
C_free_corr: 15.
C_free_image: 19, 82.
C_get_image: 3, 16, 64.
C_HAS_DEFAULT_A: 31.
C_Head: 37, 47, 48, 56.
C_ICONV_INVALID: 48, 62, 74, 75.
C_IEEE_754_DOUBLE_W: 22.
C_IEEE_754_SINGLE_W: 22.
C_IICONV: 75, 87.
C_Image: 4, 10, 16, 17, 19, 40, 49, 64, 65, 79, 80, 81, 82.
C_IMAX: 87, 88.
C_INEEDICONV: 71, 87.
C_Infos: 9, 71, 75, 88.
C_INTEGER_L: 29.
C_INTEGER_W: 21.
C_INT16_W: 22.
C_INT32_W: 22.
C_INT64_W: 22.
C_INT8_W: 22.
C_INVALID_TRANSACTION: 48.
C_IS_DELETED: 39.
C_IS_MODIFIED: 17, 39.
C_IS_NUMERIC: 29.
C_IS_STRING: 26.
C_Logical: 24, 30, 35.
C_LOGICAL_L: 29.
C_MAX_NB_ATTS: 3, 56.
C_MODIFIED: 48.
C_NB_HEAD_VALUES: 37.
C_NULL_ALLOWED_A: 31.
C_open: 13, 37, 47.
C_psql_close: 84.
C_psql_EALLOCATT: 93.
C_psql_EBADFORMAT: 93.
C_psql_ECMD: 93.
C_psql_EMISSINGSELECTION: 93.
C_psql_EMISSINGVALUES: 93.
C_psql_EOPENCACHE: 93.
C_psql_EREAD: 93.
C_psql_get_image: 84.
C_psql_open: 84.
C_psql_set_image: 84.
C_psql_WSETTYPE: 90.
C_RAW_L: 3, 25.
C_read_data: 14, 58.
C_read_head: 14, 54, 56.
C_RESTORING: 17, 48, 51, 65.
C_set_image: 17, 65.
C_SET_W: 22.
C_SIGNED_W: 21.
C_stdin_close: 85.
C_stdin_EBADDATA: 93.
C_stdin_EBADGROUP: 93.
C_stdin_get_image: 85.
C_stdin_open: 85.
C_stdin_set_image: 85.
C_stdout_close: 86.
C_stdout_get_image: 86.
C_stdout_open: 86.
C_stdout_set_image: 86.
C_TEXT_L: 26, 66, 70.
C_TIME_L: 29.
C_UINT16_W: 22.
C_UINT32_W: 22.
C_UINT64_W: 22.
C_UINT8_W: 22.
C_UNSIGNED_W: 21.
C_URI_L: 26, 27.
C_Warnings: 9, 58, 62, 71, 91.
C_WBADICONV: 71, 90.
C_WBADNBDATUMS: 58, 90.
C_WCLIENTEXIT: 90.
C_WDATEINVALID: 62, 90.
C_WICONVINVALID: 62, 90.
C_WMAX: 90, 91.
C_Word: 21, 29, 35.
C_WORD_TO_DIGITS: 23.
C_WPARSE: 90.
C_WPARSER: 90.
C_write_data: 14, 58.
C_write_head: 14, 54, 56.
C_WVALUECUT: 90.
calloc: 52, 81, 82.
cd: 74, 75.
cd_in: 49.
cd_out: 49.
ceil: 23.
chunk_size: 3, 37, 56, 64.
clean_image: 79.
close: 49, 62, 83, 84, 85, 86.
cluster: 37, 56.
Corr: 11, 13, 14, 15, 17, 46, 47, 48, 49, 51, 52, 53, 54, 56, 58, 59, 60, 62, 63, 64, 65, 67, 70, 72, 73.
corr: 16.
CORR_HEAD_SUBELT: 37, 46, 53.
CORR_MAJOR: 9.
CORR_MINOR: 9.
CORR_REVISION: 9.
Correspondence: 4.
creation: 46, 47.

Data: 11, 37, 48, 53, 58, 64, 65, 70, 72, 73.
dataset: 2, 4, 11, 13.
Datum: 23, 26, 29.
datum_group: 37, 56.
datum_group_name: 37, 56, 60, 63.
DATUM_GROUP_NAME: 60.
Datums: 36, 58, 70.
dbase: 37, 56.
dbf: 12.
Default_data: 41, 55.
Default_head: 41, 55, 56.
default_value: 31, 32, 34, 58.
delta: 75, 76.
dest: 74, 75, 76, 77.
df: 49.
DFile: 49.
Driver: 61.
driver: 37, 56, 59.
dst: 71, 75, 76.
dstleft: 71, 75, 76.
dump_retrieved: 49, 64.
efficiency: 7.
encoding: 3, 37, 56, 66, 67.
end: 47, 53, 54, 59, 60, 61, 64, 65, 67, 70, 71, 72, 73, 75, 76, 77, 81, 82.
errno: 52, 53, 54, 62, 67, 70, 71, 75, 76, 77, 81, 82.
errors: 94, 95.
exp_max: 21, 22, 23, 58.
EXTERN: 49.
E2BIG: 75.
fclose: 62.
fd: 47, 53, 54.
fdopen: 53.
fgets: 53.
fields: 38, 72, 73, 80, 81.
fields_convs: 49, 66.
flags: 21, 22, 31, 34, 38, 39, 48, 51, 58, 62, 64, 65, 72, 73, 74, 75.
floor: 23.
fp: 14, 56, 57, 58.
fprintf: 56, 58.
free: 62, 72, 73, 75, 76, 80, 82.
fstat: 54.
G__open: 11, 46, 53.
G_ENOENT: 46.
G_EOPEN: 53.
G_ERROR_MAX: 88, 91, 94.
G_ERROR_SHIFT: 52, 53, 54, 62, 67, 70, 71, 75, 76, 77, 81, 82.
G_Errors: 53, 94.
G_FATAL: 94.
G_fd_to_file_mode: 53.
G_find_file2: 46.
G_INFO: 60, 71, 75.
G_mapset: 46.
G_msg: 53, 58, 60, 62, 71, 75, 94.
G_Msgs: 9, 88, 91, 94.
G_O_RDONLY: 46, 54.
G_O_RDWR: 47.
G_O_WRONLY: 46.
G_SUCCESS: 9, 47, 56, 58, 60, 62, 64, 65, 75, 80, 81, 82.
G_WARNING: 53, 58, 62, 71.
get_image: 49, 83, 84, 85, 86.
GISDATABASE: 37, 46.
group: 2, 10, 40, 64, 65.
group_all_as: 37, 56.
group_empty_as: 37, 56.
handler: 37, 47, 48, 49, 51, 52, 62, 63.
Head: 3, 11, 37, 48, 53, 54, 56, 59, 60, 63, 64, 67.
head: 2, 11.
hf: 49, 53, 54, 62.
HIC: 49, 50, 62.
i: 58, 64, 65, 70, 80, 81.
iconv: 66, 71, 75.
iconv_close: 68.
iconv_open: 67.
iconv_t: 49, 53, 67, 68, 74, 75.
id: 2, 17, 38.
id_kt: 9, 40.
image: 2.
Image: 16, 17, 19, 49, 64, 65, 79, 80, 81, 82.
infos: 88, 89.
INT: 43.
int16_t: 6.
int32_t: 6, 23.
invalid: 75.
ioctl: 49, 83, 84, 85, 86.
j: 72, 73, 80.
KerGIS: 2.
KERGIS_CORR_H: 9.
KT_ICONV_UTF_8: 56, 67.
length: 38.
libpart: 4.
Logical: 26, 29, 35, 58, 70.
M_LN10: 23.
M_LN2: 23.
malloc: 70, 71, 75, 77.
Mapping: 37, 48, 53, 60, 63.
mapset: 46, 47, 53.
mode: 46, 47, 48, 53, 54, 60.
msgnum: 88, 91, 94.
multiplicity: 3, 4, 40, 64, 65.
name: 11, 20, 30, 37, 46, 47, 53, 54, 58, 60.

NAME: 43.
nb_att: 36.
nb_atts: 19, 36, 58, 64, 65, 81.
nb_atts_here: 3, 4, 40, 64, 65, 80, 81.
nb_bits: 23.
nb_datums: 19, 36, 58, 64, 70, 72, 73, 80, 81, 82.
nb_of_digits: 23.
need_iconv: 70, 71.
null_as: 37, 56.
O_WRONLY: 16.
offset: 3, 16, 40, 64, 65.
oflag: 13, 16.
open: 49, 83, 84, 85, 86.
OPEN_MAX: 49.
p: 75, 78.
P_close: 63.
P_Errors: 60.
P_GROUP_ALL: 16, 17, 18, 64, 65.
P_Map: 37, 48.
P_open: 60.
partition: 2, 11, 13, 37, 56, 60.
portability: 6.
psql: 12.
realloc: 76.
request: 49.
s: 72, 73, 75.
scale: 33, 34, 58.
SEARCH_PATH: 46.
separator: 37, 56.
set_image: 49, 83, 84, 85, 86.
shift: 33, 34, 58.
sign: 23.
significand: 21, 22, 23, 58.
size: 21, 31, 34, 58.
size_dest: 75, 76.
src: 71, 74, 75, 77, 78.
srcleft: 71, 75, 78.
st_size: 54.
stat: 54.
status: 47, 53, 54, 56, 58, 59, 60, 61, 62, 63, 64, 65, 67, 70, 71, 72, 73, 75, 76, 77, 80, 81, 82.
strcmp: 59, 60.
strcpy: 75.
STRING: 43.
strlen: 78.
table: 37, 56.
Text: 2.
to_be_restored: 49, 65.
transaction: 12.
type: 21, 24, 26, 29, 30, 58, 70.
uint8_t: 24.
unit: 33, 34, 58.
US: 69.
warnings: 91, 92.
Word: 22, 23, 24, 35, 58.

⟨ Allocate and init new **_C_Corr**; **goto** *end* if no handler left or problem 52 ⟩ Used in section 47.
⟨ C lang errors formats 95 ⟩ Used in section 94.
⟨ C lang infos formats 89 ⟩ Used in section 88.
⟨ C lang warnings formats 92 ⟩ Used in section 91.
⟨ Check or init correspondence file 54 ⟩ Used in section 53.
⟨ Close partition mapping 63 ⟩ Used in section 62.
⟨ Close *iconv* converters 68 ⟩ Used in section 62.
⟨ Convert fields for in 73 ⟩ Used in section 64.
⟨ Convert fields for out 72 ⟩ Used in section 65.
⟨ Errors macros 93 ⟩ Used in section 9.
⟨ Handle the Λ and `'\0'` cases 77 ⟩ Used in section 75.
⟨ Infos macros 87 ⟩ Used in section 9.
⟨ Init fields conversions 67 ⟩ Used in section 47.
⟨ Init pointers to driver functions 59 ⟩ Used in section 53.
⟨ Init the dbf back-end 83 ⟩ Used in section 59.
⟨ Init the psql back-end 84 ⟩ Used in section 59.
⟨ Init the stdin back-end 85 ⟩ Used in section 59.
⟨ Init the stdout back-end 86 ⟩ Used in section 59.
⟨ Open head if not anonymous and init *DFile* 53 ⟩ Used in section 47.
⟨ Other image functions 65 ⟩ Used in section 64.
⟨ Other memory functions 81, 82 ⟩ Used in section 80.
⟨ Private macros 51 ⟩ Used in section 45.
⟨ Private prototypes 55, 74, 79 ⟩ Used in section 45.
⟨ Private structures 49, 69 ⟩ Used in section 45.
⟨ Public macros 3, 22, 23, 25, 26, 29, 31, 39 ⟩ Used in section 9.
⟨ Public prototypes 13, 14, 15, 16, 17, 19 ⟩ Used in section 9.
⟨ Public structures 21, 30, 34, 35, 36, 37, 38, 40, 48 ⟩ Used in section 9.
⟨ Realloc room in destination and reset the offset and size 76 ⟩ Used in section 75.
⟨ Request driver to init or **goto** *end* 61 ⟩ Used in section 47.
⟨ Restart the lexer 57 ⟩ Used in sections 56 and 58.
⟨ Set mapset and creation flag 46 ⟩ Used in section 47.
⟨ Set partition conversion 60 ⟩ Used in section 47.
⟨ Set *fields_convs* 70 ⟩ Used in section 67.
⟨ Set *need_iconv* to −1, 0 or 1 depending on problem, byte or wyde promotion 71 ⟩ Used in section 70.
⟨ Set *srcleft* to *src* length 78 ⟩ Used in section 75.
⟨ Warnings macros 90 ⟩ Used in section 9.
⟨ `_corr.h` 45 ⟩
⟨ `close.c` 62 ⟩
⟨ `corr.h` 9 ⟩
⟨ `data.c` 58 ⟩
⟨ `errors.c` 94 ⟩
⟨ `head.c` 56 ⟩
⟨ `image.c` 64 ⟩
⟨ `infos.c` 88 ⟩
⟨ `locale.c` 75 ⟩
⟨ `memory.c` 80 ⟩
⟨ `open.c` 47 ⟩
⟨ `warnings.c` 91 ⟩