The dBASE Correspondence back-end

(Work in progress—handle with care)
$Id: libdbf.w,v 1.18 2005/06/17 14:39:39 tlaronde Exp

Thierry LARONDE

Abstract

`dbaseIII` is a common format used to hold records. It is, for example, used as the file format for attributes in ESRI `shape` specification.

The purpose of the library is to be able to translate between the dbase format and the KerGIS Attributes one. Hence, this library is not a general purpose library allowing to directly modify a dbase database, but a library interfacing to the KerGIS way of handling attributes.

`dbaseIII` est un format très répandu pour distribuer des informations sous forme d'enregistrements. Il est, par exemple, utilisé comme format pour les attributs dans la spécification `shape` de ESRI.

La finalité de cette bibliothèque est de permettre les transcriptions entre le format dbase et le format de KerGIS pour la gestion des attributs. Cette bibliothèque n'est donc pas une bibliothèque générique de manipulation d'enregistrements dbase, mais une collection de routines d'interfaçage avec la gestion des attributs dans KerGIS.

November 26, 2006 at 11:44

1. Licence.

Copyright 2004-2006 Thierry LARONDE
All rights reserved.
In what follows, AUTHORS stands for Thierry LARONDE.

THIS SOFTWARE IS PROVIDED BY THE AUTHORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR ITS USE OR DEALING, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

YOU USE THIS SOFTWARE AT YOUR OWN RISK AND UNDER YOUR OWN RESPONSABILITY AND USING IT IMPLIES ACCEPTATION OF THE TERMS OF THIS LICENCE.

THIS AGREEMENT IS GOVERNED BY THE LAWS OF FRANCE.

Copyright 2004-2006 Thierry LARONDE
Tous droits réservés.
Dans ce qui suit, le terme AUTEURS est mis pour : Thierry LARONDE.

CE PROGICIEL EST FOURNI PAR LES AUTEURS "EN L'ÉTAT" ET NOUS DÉNIONS TOUTE GARANTIE DE QUELQUE SORTE QUE CE SOIT, TANT EXPLICITE QU'IMPLICITE CONCERNANT ENTRE AUTRES MAIS PAS UNIQUEMENT TOUTE GARANTIE DE COMMERCIALISATION OU D'ADÉQUATION À UN USAGE PARTICULIER. EN AUCUN CAS LES AUTEURS NE POURRONT ÊTRE TENUS POUR RESPONSABLES OU REDEVABLES DE TOUT DOMMAGE DIRECT, INDIRECT, FORTUIT, PARTICULIER, EXEMPLAIRE OU CONSÉCUTIF (Y COMPRIS, MAIS NE SE LIMITANT PAS À : L'ACQUISITION DE MARCHANDISES OU DE SERVICES DE REMPLACEMENT ; LES PERTES D'USAGE, DE TEMPS, DE DONNÉES OU DE REVENUS ; OU L'INTERRUPTION D'ACTIVITÉ) QUI POURRAIT RÉSULTER DE L'USAGE DU PRÉSENT PROGICIEL, ET NOUS RÉFUTONS TOUTE PRÉSOMPTION DE RESPONSABILITÉ QUEL QUE SOIT LE MOTIF INVOQUÉ, QUE CE SOIT DANS LE CADRE D'UN CONTRAT, POUR DES RESPONSABILITÉS STRICTES OU DES PRÉJUDICES (Y COMPRIS DÛS À UNE NÉGLIGENCE OU AUTRE) SE PRODUISANT DE QUELQUE MANIÈRE QUE CE SOIT DIRECTEMENT, INDIRECTEMENT OU EN DEHORS DU LOGICIEL, DE SON USAGE OU DE SES UTILISATIONS, MÊME EN CAS D'AVERTISSEMENT DE LA POSSIBILITÉ DE TELS DOMMAGES.

VOUS UTILISEZ CE PROGICIEL ENTIÈREMENT À VOS RISQUES ET PÉRILS ET SOUS VOTRE ENTIÈRE RESPONSABILITÉ, ET CETTE UTILISATION VAUT ACCEPTATION DE CETTE LICENCE.

CET ACCORD EST RÉGI PAR LES LOIS FRANÇAISES.

2. Overview.

The *dBASE* format is quite simple: a header describing the fields, followed by the records. Its simplicity has, at least, one drawback: the precision and length of the values in the fields are limited. This means that some precision may be lost when converting from KerGIS Attributs to *dBASE*, and that

$$kat \rightarrow dbf \rightarrow kat \neq kat$$

So we are going to describe the mapping between the two.

3. If some values are not specified, we set defaults. You'd better know what they are!

```
⟨ Set defaults 3 ⟩ ≡
  if (Dbf→Head→table ≡ Λ) {
    status ← C_dbf_EMISSINGVALUES;
    goto end;
  }
  if (Dbf→Head→dbase ≡ Λ) Dbf→Head→dbase ← X_DBF_SUBELT;
  if (Dbf→Head→cluster ≡ Λ) Dbf→Head→cluster ← G_mapset( );
  Dbf→Head→null_as ← "";
  if (Dbf→Head→encoding ≡ Λ) Dbf→Head→encoding ← KT_ICONV_CP850;
```

This code is used in section 13.

4. The current implementation has some limits.

We will do nothing with BLOB since I don't have the relevant information to handle them.

There is no query language, just raw access and administration of the data. The library only provide access to whole records: it does not retrieve a value from a single field. It is the responsibility of the caller to extract exactly what he needs.

This attributes back end works at *C_Att* level : this is the responsability of the caller (for example `libatt`) to organize, query and return *C_Image*, since we work in KerGIS on the image level (see the `libatt` description).

Modification of an attribute, called in DBF a *record*, is made by deleting a record and inserting a new one.

Record locking and appending guaranteed to be an atomic operation are not handled for now and will be implemented partly in the `libgis` at *G__open* level, and partly by opening several distinct fds on the same file with different flags.

5. *dBASE* is mainly—that is one of its strength—a byte format (ASCII but probably used with eight bit too for texts). The integers (16 and 32 bits) are in little endian. We will be using macros provided by `include/ksys/endian.h` to ensure portability.

6. 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 conventions are the KerGIS ones. The API conventions are the KerGIS ones too: useful values are returned by putting the result in a memory place, the returning value being whether **void**, if no error is significant, or an **int** specifying the status: `G_SUCCESS` $\equiv 0$ for OK, or the error number if there was one.

```
⟨ corr/dbf.h 6 ⟩ ≡
#ifndef DBF_H
#define DBF_H
#include "gis.h"
#include "corr.h"
  ⟨ Public prototypes 8 ⟩
#endif /* DBF_H */
```

7. Opening/closing a *dBASE* file.

When dealing with a *dBASE* file, except the obvious information about the pathname of the file to deal with, the more important information is the one about the *dBASE* fields we are going to deal with, whether when creating a file, or when reading a file.

The internals must be kept. . . internal, so we need to exchange two kind of informations.

On opening, the filename will be combined with the *cluster*, *dbase* and location to localize the file in the gisdatabase.

Since the `libdbf` can be used inside other libraries, being combined for example inside the `libshape`, it does not have a hardcoded dedicated sub-element.

Once opened, since informations shall be initialized, just a mean to identify precisely the opened file to deal with.

Actually, on opening a pointer to a *C_DFile* structure will be passed to the *C_dbf_open* along with the *mode* that must correspond to one of the *G__open* ones. Hence, opening is done with *G__open* policy.

8. The *C_dbf_open* function does what was advertised: take a pointer to a *C_DFile*, try to open the requested file (including creation), filling the *handler* member of the structure on success.

When opening an existing *dBASE* file, the *Data* array is filled to give the information to the caller.

When creating a new file, *C_dbf_open* will always fail if the file already exists (this is handled by *G__open*).

⟨ Public prototypes 8 ⟩ ≡

int *C_dbf_open*(**struct** *C_DFile* ∗*Dbf*);

See also sections 9, 10, and 11.

This code is used in section 6.

9. Closing, from the caller's standpoint is quite simple.

⟨ Public prototypes 8 ⟩ +≡

int *C_dbf_close*(**struct** *C_DFile* ∗*Dbf*);

10. Dealing with records.

There is at the moment no query implementation, nor an index file to give a correspondence between a group id and record number (att id).

Therefore the selection is simply *att_id* ← *group_id*, with the special cases `P_GROUP_EMPTY` and `P_GROUP_ALL` taken into account.

The function for reading a record, reading next if *Att→id* ≡ 0 or reading the specified attribute.

⟨ Public prototypes 8 ⟩ +≡

int *C_dbf_get_image*(**struct** *C_DFile* ∗*Dbf*, **struct** *C_Image* ∗*Image*);

11. This library does only provide access to a whole record in the *dBASE* file, whether when inserting or reading.

When modifying a record, the caller must call the function *C_dbf_set_image*. Mode is set by flags in the *Att* structure.

If modifying an att, it will write it at the specified address. If the record is new, it is appended.

⟨ Public prototypes 8 ⟩ +≡

int *C_dbf_set_image*(**struct** *C_DFile* ∗*Dbf*, **struct** *C_Image* ∗*Image*);

12. Implementation.

The details of the manipulations are hidden from the public. Furthermore, critical data structures are kept strictly internal to avoid the mess.

The naming conventions are the ones adopted for the whole KerGIS sources.

```
⟨ dbf/_dbf.h 12 ⟩ ≡
#ifndef _DBF_H
#define _DBF_H
#include <assert.h>
#include <errno.h>
#include <string.h>      /* memcpy */
#include <sys/types.h>      /* off_t */
#include "ksys/endian.h"      /* portability stuff */
#include "gis.h"
#include "corr.h"
#include "corr/dbf.h"
  ⟨ Private macros 16 ⟩
  ⟨ Field structure 24 ⟩
  ⟨ Header structure 21 ⟩
  ⟨ Private structures 14 ⟩
  ⟨ Private prototypes 29 ⟩
#endif      /* _DBF_H */
```

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

If a new file is created, the *Header* is written. If the file already exists, the *Header* is filled with the information.

The pathname of the file is always set according to the gisdatabase, location, mapset (explicitely set or implicitely retrieved from gis environment), the *dbase* if set, or `X_DBF_SUBELT` and the filename.

```
⟨ dbf/open.c 13 ⟩ ≡
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "dbf/_dbf.h"
  int C_dbf_open(struct C_DFile *Dbf)
  {
    int status ← G_SUCCESS;      /* return value */
    int i;
    Dbf→handler ← −1;
    ⟨ Set defaults 3 ⟩
    ⟨ Allocate and init new _C_dbf_File; goto end if no handler left or problem 17 ⟩
    ⟨ Open the file or goto end 18 ⟩
    ⟨ Check or init dbf file 19 ⟩
  end: return status;
  }
```

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

Some limits (one may open several distinct *dBASE* files) have to be set. Since there is only, on the `libdbf` side, a single file to handle, we can set the number of handlers to the minimum value of `OPEN_MAX`.

```
⟨ Private structures 14 ⟩ ≡
  struct _C_dbf_File {
    struct Header Header;
    unsigned long atts_written;      /* this number were written since header update */
    int fd;
  };
#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 nb_dbf_handlers;      /* nb of handlers actually in use */
      EXTERN
          struct _C_dbf_File *dbf_Handlers[_MAX_HANDLERS];
#define _ADMIN  (dbf_Handlers[Dbf→handler])
#define _DBF(dbf_Handlers[Dbf→handler])→fd
#define _HEADER  ((dbf_Handlers[Dbf→handler])→Header)
#define _DATUMS  ((Dbf→Data)→Datums)
#define _NB_DATUMS  Dbf→Data→nb_datums
#define _FIELD  _HEADER.Fields
```

This code is used in section 12.

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

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

```
⟨ Private macros 16 ⟩ ≡
#define _CHECK_HANDLER if (Dbf→handler < 0∨Dbf→handler > _MAX_HANDLERS∨dbf_Handlers[Dbf→handler] ≡ Λ)
      return C_EBADHANDLER
```

See also sections 22 and 57.

This code is used in section 12.

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

⟨ Allocate and init new **_C_dbf_File**; **goto** *end* if no handler left or problem 17 ⟩ ≡

```
if (nb_dbf_handlers ≥ _MAX_HANDLERS) {
  status ← C_EOPENMAX;
  goto end;
}
for (Dbf→handler ← 0; Dbf→handler < _MAX_HANDLERS ∧ dbf_Handlers[Dbf→handler] ≠ Λ; ++Dbf→handler) ;
if ((dbf_Handlers[Dbf→handler] ← (struct _C_dbf_File *) malloc(sizeof(struct _C_dbf_File))) ≡ Λ) {
  Dbf→handler ← −1;
  status ← errno ≪ G_ERROR_SHIFT;
  goto end;
}
++nb_dbf_handlers;
_ADMIN→atts_written ← 0;
```

This code is used in section 13.

18. Opening is done via *G__open* enforcing the policy (advisory locking will be one day implemented in `libgis` and enforced via this very function).

⟨ Open the file or **goto** *end* 18 ⟩ ≡

```
errno ← 0;
if ((_DBF ← G__open(Dbf→Head→dbase, Dbf→Head→table, Dbf→Head→cluster, Dbf→mode)) < 0) {
  status ← errno ? errno ≪ G_ERROR_SHIFT : G_EOPEN;
  C_dbf_close(Dbf);
  goto end;
}
```

This code is used in section 13.

19. If the file is empty and a write mode was requested, we need to initialize the header. If not empty, we read the information that shall be there.

```
⟨ Check or init dbf file 19 ⟩ ≡
  {
    struct stat buf;
    if (fstat(_DBF, &buf) < 0) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
    if (buf.st_size) {
      if (¬(status ← _C_dbf_read_header(Dbf))) {
        ⟨ Translate Header in Data 26 ⟩
      }
    }
    else if (Dbf→mode ≡ G_O_RDONLY) status ← C_dbf_EEMPTY;
    else {      /* creating */
      Dbf→Data→nb_atts ← 0;      /* will be feed after */
      Dbf→flags |= C_MODIFIED;      /* header will be written */
      ⟨ Translate Data in Header 23 ⟩
      _C_dbf_write_header(Dbf);
    }
    if (status) {
      C_dbf_close(Dbf);
      goto end;      /* already error, bail out */
    }
  }
```

This code is used in section 13.

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

```
⟨ dbf/close.c 20 ⟩ ≡
#include <unistd.h>
#define HIC
#include "dbf/_dbf.h"
  int C_dbf_close(struct C_DFile *Dbf)
  {
    int status ← G_SUCCESS;      /* be able to be safely called when not opened (interrupted user program */
    if (Dbf→handler ≡ −1) return G_SUCCESS;
    _CHECK_HANDLER;
        /* if header changed, flush it to disk */
    _C_dbf_write_header(Dbf);
    if (_DBF ≠ −1) {
      if (close(_DBF) < 0) status ← errno ≪ G_ERROR_SHIFT;
    }
    free(dbf_Handlers[Dbf→handler]);
    dbf_Handlers[Dbf→handler] ← Λ;
    Dbf→handler ← −1;
    −−nb_dbf_handlers;
    return status;
  }
```

21. *Header* **manipulation.**

The *dBASE* format starts by a 32 bytes file *Header* followed by a vector of *Field* structures, ending with the HEADER_TERMINATOR and followed by the actual *Record*s.

The format is mainly a vector of bytes, the rare numbers being little endian.

We will in turn describe these three elements.

Indeed, on a little endian 32 bits word computer with no alignment constraints, we could cast a pointer in a buffer to the matching structure to access the members.

But, as said, this is only true on such very type of machine.

Since we aim portability, we will translate between this virtual 32 bits word little endian no alignment constraints *dBASE* machine, to the actual machine we are working on.

The structure members make the signification of the code more clear so filling the structure is not totally a waste of time.

We take C types that are guaranteed to be at least as big as the actual sizes used in the *dBASE* file :

- long is at least *int32_t*;
- short is at least *int16_t*;
- since the copy uses *memcpy*, a string routine handling **char**s, we do not cast explicitely to *uint8_t*.

⟨ Header structure 21 ⟩ ≡

```
#define HEADER_SIZE 32
  struct Header {
    unsigned char version;
    struct {
      unsigned char yy;      /* binary */
      unsigned char mm;      /* binary */
      unsigned char dd;      /* binary */
    } last_update;      /* add 1900 to the year */
    unsigned long nb_records;
    unsigned short data_offset;      /* allows to compute number of fields */
    unsigned short record_length;      /* including delete flag */
    char reserved1[16];
    unsigned char table_flag;
      /* #01 has cdx, #02 has a Memo field, #04 is a database (.dbc) NOT USED BY THE LIB */
    unsigned char code_page;
      /* ArcView stores #57 to indicate ANSI coding instead of DOS437 one. By default, 0 shall be used */
    char reserved2[2];
    struct Field *Fields;
  };
#define HEADER_TERMINATOR #0D      /* separator between whole header and data */
```

This code is used in section 12.

22. When we are creating a new file, the header must be set. There are defaults, and an update that must come from the *Data* member of the *C_DFile* structure.

⟨ Private macros 16 ⟩ +≡

```
#define HEADER_VERSION #03      /* dBASE III with no memo for creation */
#define CODE_PAGE 0      /* Default to DOS850, compatible with DOS437 */
```

23. There are some default values, and the remaining is taken from the *Data* definition.

```
⟨ Translate Data in Header 23 ⟩ ≡
  _HEADER.version ← HEADER_VERSION;
  ⟨ Set header modification date 37 ⟩
  _HEADER.nb_records ← 0;
  _HEADER.data_offset ← (unsigned short)(32 + _NB_DATUMS * 32 + 1);      /* add separator */
  _HEADER.record_length ← 0;      /* filled later */
  for (i ← 0; i < 16; i++) _HEADER.reserved1[i] ← 0;
  _HEADER.table_flag ← 0;
  _HEADER.code_page ← CODE_PAGE;
  for (i ← 0; i < 2; i++) _HEADER.reserved2[i] ← 0;
  ⟨ Alloc Fields 51 ⟩
  ⟨ Translate Data in Fields 27 ⟩
```

This code is used in section 19.

24. The description of the *Field* structure varies depending on the version. We will write dBASE III format and try to read up to dBASE V.

```
⟨ Field structure 24 ⟩ ≡
#define FIELD_SIZE 32
  struct Field {
    char name[11];      /* if less than 10 padded with zeroes */
    char type;
    long offset;      /* set in memory, not in file */
    unsigned char length;      /* in bytes */
    unsigned char precision;      /* number of decimal places */
    char flags;      /* precisions about the type */
    long auto_next;      /* next auto_increment value */
    char auto_step;      /* size of step */
    char reserved[8];
  };
#define IS_SYSTEM(field) ((field).flags & #01)
#define CAN_NULL(field) ((field).flags & #02)
#define IS_BINARY(field) ((field).flags & #04)
#define IS_AUTOINC(field) ((field).flags & #0C)
```

This code is used in section 12.

25. There are too symetric operations that need to be performed: translating a *dBASE* field in a KerGIS *C_Datum*, and translate an *C_Datum* in a *dBASE* field.

The former is simpler since the type and precisions are fixed. So let be lazy and start by the—relatively—easy.

26. Translating a *dBASE* field to a KerGIS *C_Datum* should be easy... if we had all the information about the precision of the *dBASE* type. Unfortunately, the information I have at the moment, coming from an Internet search, is neither clear nor complete. Ajustement will perhaps be necessary when experience and real data show hiatus.

```
⟨ Translate Header in Data 26 ⟩ ≡
  Dbf→Data→nb_atts ← _HEADER.nb_records;
  if ((Dbf→Data→Datums ← (struct C_Datum *) calloc(_NB_DATUMS, sizeof(struct C_Datum))) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    C_dbf_close(Dbf);
    goto end;
  }
  for (i ← 0; i < _NB_DATUMS; i++) {
    char *datum_name;
    if ((datum_name ← (char *) malloc(11)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      C_dbf_close(Dbf);
      goto end;
    }
    (void) strcpy(datum_name, _FIELD[i].name);      /* these are defaults that may be overwritten */
    (_DATUMS[i]).Logical.name ← datum_name;
    (_DATUMS[i]).Admin.size ← 1;      /* number of C_Word */
    (_DATUMS[i]).Admin.unit ← 1;
    switch (_FIELD[i].type) {
    case 'C': C_UINT8_W((_DATUMS[i]).Word);
      (_DATUMS[i]).Logical.type ← C_TEXT_L;
      (_DATUMS[i]).Admin.size ← _FIELD[i].length;
      break;
    case 'I': C_INT32_W((_DATUMS[i]).Word);
      (_DATUMS[i]).Logical.type ← C_INTEGER_L;
      break;
    case 'N':
      if (_FIELD[i].precision ≡ 0) {
        C_INT32_W((_DATUMS[i]).Word);
        (_DATUMS[i]).Logical.type ← C_INTEGER_L;
      }
      else {
        C_IEEE_754_DOUBLE_W((_DATUMS[i]).Word);
        (_DATUMS[i]).Logical.type ← C_FLOAT_L;
      }
      break;
    case 'F': case 'B': C_IEEE_754_DOUBLE_W((_DATUMS[i]).Word);
      (_DATUMS[i]).Logical.type ← C_FLOAT_L;
      break;
    case 'L': C_SET_W((_DATUMS[i]).Word, 0, 2, 0);      /* Λ also with true and false */
      (_DATUMS[i]).Logical.type ← C_LOGICAL_L;
      break;
    case 'D': C_UINT8_W((_DATUMS[i]).Word);
      (_DATUMS[i]).Logical.type ← C_DATE_L;
      (_DATUMS[i]).Admin.size ← _FIELD[i].length;
      break;
    case 'T': C_UINT32_W((_DATUMS[i]).Word);
      (_DATUMS[i]).Logical.type ← C_TIME_L;
      break;
```

```
    case 'Y':      /* in the range ±922337203685477.5807 ! */
      C_SET_W((_DATUMS[i]).Word, C_INTEGER_W | C_SIGNED_W, C_DIGITS_TO_BITS(20, 1), 0);
      (_DATUMS[i]).Logical.type ← C_DECIMAL_L;
      (_DATUMS[i]).Admin.scale ← −4;
      break;
    default: G_msg(C_dbf_WTYPE, C_Warnings, G_WARNING, _HEADER.Fields[i].type);
      C_UINT8_W((_DATUMS[i]).Word);
      (_DATUMS[i]).Logical.type ← C_RAW_L;
      (_DATUMS[i]).Admin.size ← _FIELD[i].length;
      break;
    }
  }
```

This code is used in section 19.

27. The converse operation consists of trying to translate data format specified in *Data* in a *dBASE* header. Some things will not be possible, and for the moment with take into account the type, and emit a warning if the size of the type exceeds *dBASE* capabilities.

```
⟨ Translate Data in Fields 27 ⟩ ≡
  {
    unsigned short offset ← 1;      /* after flag */
    for (i ← 0; i < _NB_DATUMS; i++) {      /* calloc ensures padding with zeroes */
      _FIELD[i].offset ← offset;
      (void) memcpy(_FIELD[i].name, (_DATUMS[i]).Logical.name, 10);
      if (C_IS_NUMERIC(_DATUMS[i])) {
        if (C_WORD_TO_DIGITS((_DATUMS[i])) > 254) {
          G_msg(C_dbf_WTYPELENGTH, C_Warnings, G_WARNING, _DATUMS[i].Word.significand,
              _DATUMS[i].Word.exp_max);
          _FIELD[i].length ← 254;
        }
        else _FIELD[i].length ← (unsigned char) C_WORD_TO_DIGITS((_DATUMS[i]));
      }
      else _FIELD[i].length ← (unsigned char) _DATUMS[i].Admin.size;
      switch ((_DATUMS[i]).Logical.type) {
      case C_TEXT_L: _FIELD[i].type ← 'C';
        break;
      case C_LOGICAL_L: _FIELD[i].type ← 'L';
        _FIELD[i].length ← 1;      /* we may have comput more */
        break;
      case C_INTEGER_L: case C_DECIMAL_L: case C_FLOAT_L: _FIELD[i].type ← 'N';
        break;
      case C_DATE_L: case C_TIME_L: _FIELD[i].type ← 'D';
        _FIELD[i].length ← 8;
        break;
      default: G_msg(C_dbf_WSETTYPE, C_Warnings, G_WARNING, (unsigned int)(_DATUMS[i]).Logical.type);
        _FIELD[i].type ← 'C';
        break;
      }
      offset ← (unsigned short)(offset + _FIELD[i].length);
    }
    _HEADER.record_length ← offset;      /* delete flag already added */
  }
```

This code is used in section 23.

28. Just in order to be user friendly, we add an array with litteral descriptions of known versions.

```
⟨ dbf/header.c 28 ⟩ ≡
#include <unistd.h>
#include "dbf/_dbf.h"
  static struct {
    unsigned char version;
    char *description;
  } versions[ ] ← {{#02, "FoxBASE"}, {#03, "FoxBASE+/dBASE␣III+,␣no␣memo"}, {#30, "Visual␣FoxPro"},
      {#31, "Visual␣FoxPro␣autoincrementing"}, {#43, "dBASE␣IV␣SQL␣table␣files,␣no␣memo"},
      {#63, "dBASE␣IV␣SQL␣system␣files,␣no␣memo"}, {#83, "FoxBASE+/dBASE␣III+,␣with␣memo"},
      {#8B, "dBASE␣IV␣with␣memo"}, {#CB, "dBASE␣IV␣SQL␣table␣files,␣with␣memo"}, {#F5,
      "FoxPro␣2.x␣with␣memo"}, {#FB, "FoxBASE"}, {0, Λ}};
  ⟨ Header functions 30 ⟩
```

29. We simply read a buffer of 32 bytes, for the header proper, and for every field, using macros for portability.

```
⟨ Private prototypes 29 ⟩ ≡
  int _C_dbf_read_header(struct C_DFile *Dbf);
```

See also sections 34, 41, and 46.

This code is used in section 12.

30.

```
⟨ Header functions 30 ⟩ ≡
  int _C_dbf_read_header(struct C_DFile *Dbf)
  {
    int status ← G_SUCCESS;
    int i, j;
    long field_offset;      /* the offset is not set on disk and must be computed */
    char buf[32];      /* will be used for header and fields */
    ⟨ Saved offset 52 ⟩      /* check mode */
    if (Dbf→mode ≡ G_O_WRONLY) return C_EMODE;
    ⟨ Save current offset and seek to beginning 53 ⟩
    if (read(_DBF, buf, HEADER_SIZE) ≠ HEADER_SIZE) {
      status ← C_dbf_EREAD;
      goto end;
    }
    _HEADER.version ← GET_BYTE(0);
    for (i ← 0; versions[i].version ≡ _HEADER.version ∨ versions[i].description ≡ Λ; i++) ;
    if (¬versions[i].version) {
      G_msg(C_dbf_IHEADER, C_Infos, G_INFO, _HEADER.version, "unknown");
      status ← C_dbf_EWRONGHEADER;
      goto end;
    }
    else G_msg(C_dbf_IHEADER, C_Infos, G_INFO, _HEADER.version, versions[i].description);
    _HEADER.last_update.yy ← GET_BYTE(1);
    _HEADER.last_update.mm ← GET_BYTE(2);
    _HEADER.last_update.dd ← GET_BYTE(3);
    _HEADER.nb_records ← GET_LET(4);
    _HEADER.data_offset ← GET_LEW(8);
    _HEADER.record_length ← GET_LEW(10);
    for (i ← 0; i < 16; i++) _HEADER.reserved1[i] ← GET_BYTE(12 + i);
    _HEADER.table_flag ← GET_BYTE(28);
    _HEADER.code_page ← GET_BYTE(29);
    ⟨ Check encoding specified in header 33 ⟩
    for (i ← 0; i < 2; i++) _HEADER.reserved2[i] ← GET_BYTE(30 + i);
    ⟨ Set _NB_DATUMS 32 ⟩
    ⟨ Alloc Fields 51 ⟩
    field_offset ← 1;      /* record start by flag */
    for (i ← 0; i < _NB_DATUMS; i++) {
      ⟨ Read field 31 ⟩
    }      /* check terminator */
    if (read(_DBF, buf, 1) ≠ 1) {
      status ← C_dbf_EREAD;
      goto end;
    }
    if (*buf ≠ HEADER_TERMINATOR) status ← C_dbf_EWRONGHEADER;
  end: ⟨ Restore offset 56 ⟩
    return status;
  }
```

See also section 35.

This code is used in section 28.

31. Reading each field is now quite a routine.

```
⟨ Read field 31 ⟩ ≡
  if (read(_DBF, buf, FIELD_SIZE) ≠ FIELD_SIZE) {
    status ← C_dbf_EREAD;
    goto end;
  }
  for (j ← 0; j < 10; j++) _FIELD[i].name[j] ← GET_BYTE(j);
  _FIELD[i].name[10] ← '\0';      /* safety: do it ourselves */
  _FIELD[i].type ← GET_BYTE(11);
  _FIELD[i].offset ← field_offset;      /* not on disk, set in memory */
  _FIELD[i].length ← GET_BYTE(16);
  field_offset += _FIELD[i].length;
  _FIELD[i].precision ← GET_BYTE(17);
  _FIELD[i].flags ← GET_BYTE(18);
  _FIELD[i].auto_next ← GET_LET(19);
  _FIELD[i].auto_step ← GET_BYTE(23);
  for (j ← 0; j < 8; j++) _FIELD[i].reserved[j] ← GET_BYTE(24 + j);
```

This code is used in section 30.

32. To compute the number of fields is quite easy since we have the formula: $data\_offset = 32 + nb\_fields * 32 + 1$.

```
⟨ Set _NB_DATUMS 32 ⟩ ≡
  _NB_DATUMS ← (unsigned short)((_HEADER.data_offset − 33)/32);
```

This code is used in section 30.

33. By default, a *dBASE* file is encoded in DOS CP437. Since CP850 is compatible (extends) CP437, it can be used as a default. ESRI set the *code_page* to #57 for an ANSI encoding (does this mean latin1 for eight bits?). So we check the *code_page* and update the *Dbf→Head→encoding* accordingly.

```
⟨ Check encoding specified in header 33 ⟩ ≡
  if (¬_HEADER.code_page) Dbf→Head→encoding ← KT_ICONV_CP850;
  else if (_HEADER.code_page ≡ #57) Dbf→Head→encoding ← KT_ICONV_ISO_8859_1;
  else G_msg(C_dbf_WCODEPAGE, C_Warnings, G_WARNING, _HEADER.code_page);
```

This code is used in section 30.

34. When we write the header, the offset is saved and restored, and this function, like *_C_dbf_read_header* shall not be "threaded"!

```
⟨ Private prototypes 29 ⟩ +≡
  int _C_dbf_write_header(struct C_DFile *Dbf);
```

35.

```
⟨ Header functions 30 ⟩ +≡
  int _C_dbf_write_header(struct C_DFile *Dbf)
  {
    int status ← G_SUCCESS;
    int i;
    uint8_t a_byte;
    uint16_t a_wyde;
    uint32_t a_tetra;
    char buf[32];
    long offset ← 0;
    ⟨ Saved offset 52 ⟩ /* check mode */
    if (Dbf→mode ≡ G_O_RDONLY) return C_EMODE; /* if no change, do nothing */
    if (¬(Dbf→flags & C_MODIFIED)) return G_SUCCESS;
    ⟨ Save current offset and seek to beginning 53 ⟩ /* we update when writing */
    ⟨ Set header modification date 37 ⟩ /* header first */
    PUT_BYTE(_HEADER.version);
    PUT_BYTE(_HEADER.last_update.yy);
    PUT_BYTE(_HEADER.last_update.mm);
    PUT_BYTE(_HEADER.last_update.dd);
    PUT_LET(Dbf→Data→nb_atts);
    PUT_LEW(_HEADER.data_offset);
    PUT_LEW(_HEADER.record_length);
    for (i ← 0; i < 16; i++) PUT_BYTE(_HEADER.reserved1[i]);
    PUT_BYTE(_HEADER.table_flag);
    PUT_BYTE(_HEADER.code_page);
    for (i ← 0; i < 2; i++) PUT_BYTE(_HEADER.reserved2[i]); /* write header */
    if (write(_DBF, buf, HEADER_SIZE) ≠ HEADER_SIZE) {
      status ← C_dbf_EWRITE;
      goto end;
    } /* the fields */
    for (i ← 0; i < _NB_DATUMS; i++) {
      ⟨ Write field 36 ⟩
    } /* terminator */
    *buf ← HEADER_TERMINATOR;
    if (write(_DBF, buf, 1) ≠ 1) {
      status ← C_dbf_EWRITE;
      goto end;
    } /* restore flags */
    Dbf→flags &= ∼C_MODIFIED;
  end: ⟨ Restore offset 56 ⟩
    return status;
  }
```

36. Writing a field is trivial.
The name will be put UPPERCASE.

```
⟨ Write field 36 ⟩ ≡
  {
    int j;
    char name[11];
    for (j ← 0; j < 11; j++) name[j] ← '\0';      /* need to be padded with Λs */
    (void) strcpy(name, _FIELD[i].name);
    G_toucase(name);
    offset ← 0;      /* starts from beginning */
    for (j ← 0; j < 11; j++) PUT_BYTE(name[j]);
    PUT_BYTE(_FIELD[i].type);
    PUT_LET(0);      /* offset is set in memory */
    PUT_BYTE(_FIELD[i].length);
    PUT_BYTE(_FIELD[i].precision);
    PUT_BYTE(_FIELD[i].flags);
    PUT_LET(_FIELD[i].auto_next);
    PUT_BYTE(_FIELD[i].auto_step);
    for (j ← 0; j < 8; j++) PUT_BYTE(_FIELD[i].reserved[j]);      /* write field */
    if (write(_DBF, buf, FIELD_SIZE) ≠ FIELD_SIZE) {
      status ← C_dbf_EWRITE;
      goto end;
    }
  }
```

This code is used in section 35.

37. The update date is stored as three bytes (binary value) in the order: year (from 1900), month and day.
Since we aim at some fixed reference, we will use UTC time.

```
⟨ Set header modification date 37 ⟩ ≡
  {
#include <time.h>
    time_t caltime;
    struct tm *tmptr;
    if (time(&caltime) < 0) G_msg(errno ≪ G_ERROR_SHIFT, C_Warnings, G_WARNING);
    else {
      tmptr ← gmtime(&caltime);
      _HEADER.last_update.yy ← (unsigned char) tmptr→tm_year;
      _HEADER.last_update.mm ← (unsigned char)(tmptr→tm_mon + 1);
      _HEADER.last_update.dd ← (unsigned char) tmptr→tm_mday;
    }
  }
```

This code is used in sections 23 and 35.

38. Dealing with records.

There is at the moment no query implementation, nor an index file to give a correspondence between a group id and record number (att id).

So we simply set the group id to be the att id.

There are, however, two special cases: `P_GROUP_EMPTY` and `P_GROUP_ALL`.

The first one—without a query support—does not make sense.

The second will mean: dump all.

```
⟨ dbf/get.c 38 ⟩ ≡
#include "dbf/_dbf.h"
  int C_dbf_get_image(struct C_DFile *Dbf, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    unsigned long nb_atts_here;
    struct C_Att *Att;
    _CHECK_HANDLER;      /* check mode */
    if (Dbf→mode ≡ G_O_WRONLY) return C_EMODE;
    if (Image→group ≡ P_GROUP_EMPTY) {
      Image→multiplicity ← 0;
      nb_atts_here ← 0;
    }
    else if (Image→group ≡ P_GROUP_ALL) {
      Image→multiplicity ← Dbf→Data→nb_atts;
      nb_atts_here ← (Image→multiplicity − Image→offset > Dbf→Head→chunk_size) ? Dbf→Head→chunk_size :
          Image→multiplicity − Image→offset;
    }
    else if (Image→group ≤ Dbf→Data→nb_atts ∧ ¬Image→offset) {
      Image→multiplicity ← 1;
      nb_atts_here ← 1;
    }
    else {      /* out of range */
      Image→multiplicity ← 0;
      nb_atts_here ← 0;
    }
    if ((status ← C_alloc_atts(Dbf→Data→nb_datums, Image, nb_atts_here))) goto end;
    for (i ← 0; i < Image→nb_atts_here; i++) {
      Att ← &Image→Atts[i];
      Att→id ← (Image→group ≠ P_GROUP_ALL ? Image→group : 1) + Image→offset + i;
      if ((status ← _C_dbf_read_att(Dbf, Att))) goto end;
    }
  end: return status;
  }
```

39. When setting an image, we seek to the specified place if *Att→id* $\neq 0$, or we append a new record.

```
⟨ dbf/set.c 39 ⟩ ≡
#include "dbf/_dbf.h"
  int C_dbf_set_image(struct C_DFile *Dbf, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    struct C_Att *Att;
    _CHECK_HANDLER;      /* check mode */
    if (Dbf→mode ≡ G_O_RDONLY) return C_EMODE;
    for (i ← 0; i < Image→nb_atts_here; i++) {
      Att ← &Image→Atts[i];
      if ((status ← _C_dbf_write_att(Dbf, Att))) goto end;
    }
  end: return status;
  }
```

40. All the data records for dBASE III are ASCII format (we will handle the char as unsigned chars, allowing more than strictly ASCII).

For *dBASE*, the *id* of an attribute is the record number.

A valid record starts by the char '#20', a deleted record starts by the char '#2A'.

```
⟨ Record related macros 40 ⟩ ≡
#define RECORD_ALIVE ' '     /* #20 */
#define RECORD_DELETED '*'     /* #2A */
```

This code is used in section 42.

41. Writing an att is always done at the end (appending record). The record number is returned as the id.

```
⟨ Private prototypes 29 ⟩ +≡
  int _C_dbf_write_att(struct C_DFile *Dbf, struct C_Att *Att);
```

42.

```
⟨ dbf/att.c 42 ⟩ ≡
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include "dbf/_dbf.h"
  ⟨ Record related macros 40 ⟩
  int _C_dbf_write_att(struct C_DFile *Dbf, struct C_Att *Att)
  {
    int status ← G_SUCCESS;
    char *buf;
    int i;
    ⟨ Saved offset 52 ⟩
      /* do something, inconditionnally when restoring, only if modified or new in normal case */
    if (C_IS_MODIFIED(Att) ∨ (Dbf→flags & C_RESTORING ∧ Att→id)) {
      ⟨ Save current offset and seek to specified record 54 ⟩
    }
    else if (¬Att→id) {      /* new, append */
      ⟨ Save current offset and seek to end 55 ⟩
    }
    else {      /* do nothing */
      return G_SUCCESS;
    }
    if ((buf ← (char *) malloc((size_t) _HEADER.record_length)) ≡ Λ) return errno ≪ G_ERROR_SHIFT;
        /* set record valid */
    *buf ← (C_IS_DELETED(Att)) ? RECORD_DELETED : RECORD_ALIVE;
    for (i ← 0; i < _NB_DATUMS; i++) {
      char *value;
      value ← Att→fields[i] ≡ Λ ? Λ : G_store(Att→fields[i]);
      if (_FIELD[i].type ≡ 'D' ∧ Att→fields[i] ≠ Λ) {
        ⟨ Convert the date to YYYYMMDD 43 ⟩
      }
      else if (_FIELD[i].type ≡ 'L' ∧ Att→fields[i] ≠ Λ) {
        ⟨ Convert the boolean to dbf value 45 ⟩
      }
      ⟨ Justify the field and fill with blanks handling Λ value 44 ⟩
      free(value);
    }      /* write it */
    if (write(_DBF, buf, _HEADER.record_length) ≠ _HEADER.record_length) {
      status ← C_dbf_EWRITE;
      goto end;
    }      /* set C_MODIFIED flag so that the header will be updated */
    Dbf→flags |= C_MODIFIED;      /* update number of records */
    if (C_IS_MODIFIED(Att)) Att→flags &= ∼C_ATT_MODIFIED;
    else {
      ++_ADMIN→atts_written;
      Att→id ← ++Dbf→Data→nb_atts;
    }
  end: ⟨ Restore offset 56 ⟩
```

```
    free(buf);
    return status;
  }
  ⟨ Other att functions 47 ⟩
```

43. We are supposed to receive data values in the ISO 8601 format.

```
⟨ Convert the date to YYYYMMDD 43 ⟩ ≡
  {
    char buf[8];
#define _ISO_8601_FORMAT "%c%c%c%c-%c%c-%c%c␣%*s"
    if (sscanf(value, _ISO_8601_FORMAT, buf, buf + 1, buf + 2, buf + 3, buf + 4, buf + 5, buf + 6, buf + 7) ≠ 8)
      Dbf→flags |= C_DATE_INVALID;
    else {
      int i;
      for (i ← 0; i < 8; i++) value[i] ← buf[i];
    }
  }
```

This code is used in section 42.

44. Right justifying the field seems to be mandatory for numeric types. For others, we left justify. And in every case, there is no null, only spaces for filling.

```
⟨ Justify the field and fill with blanks handling Λ value 44 ⟩ ≡
  {
    int j;
    char *es;      /* pointer to end of string for length computation */
    size_t alength;      /* handle null case */
    if (value ≡ Λ) {
      errno ← 0;
      if ((value ← (char *) malloc(1)) ≡ Λ) {
        status ← errno ≪ G_ERROR_SHIFT;
        goto end;
      }
      *value ← '\0';
    }
    for (es ← value; *es; es++) ;      /* skip to end for length computation */
    alength ← (size_t)(es − value);
    if (alength ≥ _FIELD[i].length) (void) strncpy(buf + _FIELD[i].offset, value, _FIELD[i].length);
    else if (C_IS_NUMERIC(_DATUMS[i])) {      /* right justify */
      int j;
      (void) strcpy(buf + _FIELD[i].offset + _FIELD[i].length − alength, value);
      for (j ← 0; j < _FIELD[i].length − alength; j++) buf[_FIELD[i].offset + j] ← '␣';
    }
    else {      /* left justify */
      (void) strcpy(buf + _FIELD[i].offset, value);
      for (j ← 0; j < _FIELD[i].length − alength; j++) buf[_FIELD[i].offset + alength + j] ← '␣';
    }
  }
```

This code is used in section 42.

45. Logical types are of several flavors. A switch will do (XXX more than 8 bits are not handled now).

```
⟨ Convert the boolean to dbf value 45 ⟩ ≡
  switch (*value) {
  case 0: case 'f': case 'F': case 'n': case 'N': *value ← 'F';
    break;
  case 1: case 't': case 'T': case 'y': case 'Y': *value ← 'T';
    break;
  default: *value ← '?';
    break;
  }
```

This code is used in section 42.

46. After writing an att, let's see reading one.

The *Att* **MUST** have been allocated by the caller, to ensure that pointers are whether Λ or allocated ones, since we will *realloc*. This means that the data specified in *Att* must be copied elsewhere between invocation since it is trashed.

```
⟨ Private prototypes 29 ⟩ +≡
  int _C_dbf_read_att(struct C_DFile *Dbf, struct C_Att *Att);
```

47.

```
⟨ Other att functions 47 ⟩ ≡
  int _C_dbf_read_att(struct C_DFile *Dbf, struct C_Att *Att)
  {
    int status ← G_SUCCESS;
    int i;
    char *buf;
    char value[256];      /* we will strip the values here */      /* check if request make sense */
    if (¬Att→id ∨ Att→id > _HEADER.nb_records) return C_dbf_EREADRANGE;
    ⟨ Seek to record specified by Att→id 49 ⟩
    if ((buf ← (char *) malloc(_HEADER.record_length)) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }     /* read record */
    if (read(_DBF, buf, _HEADER.record_length) ≠ _HEADER.record_length) {
      status ← C_dbf_EREAD;
      goto end;
    }     /* flags */
    Att→flags ← 0;
    if (*buf ≡ RECORD_DELETED) Att→flags |= C_ATT_DELETED;
    for (i ← 0; i < _NB_DATUMS; i++) {
      char *utf;
      (void) strncpy(value, buf + _FIELD[i].offset, _FIELD[i].length);
      value[_FIELD[i].length] ← '\0';      /* be sure to terminate */
      G_strip(value);
      if (*value) ⟨ Convert dbf field values 48 ⟩
      utf ← value ≡ Λ ? Λ : G_store(value);
      if (Dbf→Head→null_as ≠ Λ ∧ (¬strcmp(utf, Dbf→Head→null_as))) Att→fields[i] ← Λ;      /* redondant */
      else {
        char *eutf;      /* pointer to end (length) */
        for (eutf ← utf; *eutf; eutf++) ;      /* skip to end for length */
        errno ← 0;
        if ((Att→fields[i] ← (char *) malloc((size_t)(eutf − utf) + 1)) ≡ Λ) {
          status ← errno ≪ G_ERROR_SHIFT;
          free(utf);
          goto end;
        }
        (void) strcpy(Att→fields[i], utf);
      }
      free(utf);
    }
  end: free(buf);
    return status;
  }
```

This code is used in section 42.

48. We need to convert some dbf field values that are not in standard format for us.

```
⟨ Convert dbf field values 48 ⟩ ≡
  if (_FIELD[i].type ≡ 'D') {
    char mm[2];
    char dd[2];
    if (sscanf(value, "%*4c%2c%2c", mm, dd) ≠ 2) Dbf→flags |= C_DATE_INVALID;
    value[4] ← '-';
    value[5] ← mm[0];
    value[6] ← mm[1];
    value[7] ← '-';
    value[8] ← dd[0];
    value[9] ← dd[1];
    value[10] ← '\0';
  }
```

This code is used in section 47.

49. Seeking to a specified *Att→id* (for *dBASE* this is the record number) is simple enough given the size of a record and the data offset;

```
⟨ Seek to record specified by Att→id 49 ⟩ ≡
  errno ← 0;
  if (lseek(_DBF, (off_t)(_HEADER.data_offset + (Att→id − 1) ∗ _HEADER.record_length), SEEK_SET) < 0)
    return errno ≪ G_ERROR_SHIFT;
```

This code is used in sections 47 and 54.

50. Miscellanei routines.

51. Allocating memory ressources for Fields is common to *C_dbf_open* and *_C_dbf_read_header*.

```
⟨ Alloc Fields 51 ⟩ ≡
  if ((_HEADER.Fields ← (struct Field *) calloc(_NB_DATUMS, sizeof(struct Field))) ≡ Λ) {
    status ← errno ≪ G_ERROR_SHIFT;
    C_dbf_close(Dbf);
    goto end;
  }
```

This code is used in sections 23 and 30.

52. Saving and restoring offsets are routinely used, it is clearer to let them be sections.

```
⟨ Saved offset 52 ⟩ ≡
  off_t currpos_dbf ← 0;      /* current saved */
```

This code is used in sections 30, 35, and 42.

53.

```
⟨ Save current offset and seek to beginning 53 ⟩ ≡
  currpos_dbf ← lseek(_DBF, 0, SEEK_CUR);
  lseek(_DBF, (off_t)0, SEEK_SET);
```

This code is used in sections 30 and 35.

54.

```
⟨ Save current offset and seek to specified record 54 ⟩ ≡
  currpos_dbf ← lseek(_DBF, 0, SEEK_CUR);
  ⟨ Seek to record specified by Att→id 49 ⟩
```

This code is used in section 42.

55.

```
⟨ Save current offset and seek to end 55 ⟩ ≡
  currpos_dbf ← lseek(_DBF, 0, SEEK_CUR);
  lseek(_DBF, 0, SEEK_END);
```

This code is used in section 42.

56.

```
⟨ Restore offset 56 ⟩ ≡
  lseek(_DBF, currpos_dbf, SEEK_SET);
```

This code is used in sections 30, 35, and 42.

57. Conversions and precisions.

Unfortunately, the databases handle different types of numerical values with different accuracy. These types generally overflow largely minimal ranges defined by standard C.

But I have tried to design the KerGIS Attributes structures so that, at least at importation time, the exact precision of the original database is kept (it can be lost when using the collection of informations in an implementation of a database manager that can not handle the same precisions).

We will start by defining some macros that progress in a *buf*, incrementing *offset*, and casting the values. Consider *a_wyde*, *a_tetra*, *buf* and *offset* as reserved names and be careful!

```
⟨ Private macros 16 ⟩ +≡
#define PUT_LEW(some_short)
  {
    a_wyde ← wtolew((uint16_t)some_short);
    (void) memcpy(buf + offset, &a_wyde, 2);
    offset += 2;
  }
#define PUT_LET(some_long)
  {
    a_tetra ← ttolet((uint32_t)some_long);
    (void) memcpy(buf + offset, &a_tetra, 4);
    offset += 4;
  }
#define PUT_BYTE(some_char)
  {
    a_byte ← (uint8_t)some_char;
    (void) memcpy(buf + offset, &a_byte, 1);
    ++offset;
  }
#define GET_BYTE(offset) *(buf + offset)
#define GET_LEW(offset) (unsigned short) lewtow(*((uint16_t*)(buf + offset)))
#define GET_LET(offset) (unsigned long) lettot(*((uint32_t*)(buf + offset)))
```

58. In this first implementation, the flags will not be used: I limit the handling to `dBASE III`.

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