

The KerGIS Part Library

(Work in progress—handle with care)
\$Id: libpart.w,v 1.8 2006/11/17 14:32:12 tlaronde Exp

Thierry LARONDE (tlaronde@polynum.com)

Abstract

KerGIS handles geometrical/geographical elements. Some way to identify atomic elements and to select them by grouping has to be provided, both for having a mean to link geographical sets of elements with external properties and for being able to keep track of the choices made following some criterium.

This is the purpose of the KerGIS Part Library, that will allow to create multiple selections of elements belonging to a same set and provide ways to perform typical set operations: union, intersection and so on.

Another aim of the PARTLIB is to provide a mapping between group numbers— easily managed by a computer —and group names, more easily handled by human beings (*à la* DNS).

KerGIS manipule des éléments géométriques. Un moyen de les identifier et de les grouper doit donc être disponible, autant pour des manipulations de la géométrie que pour l'association d'attributs externes (non géométriques) suivant une sélection faite d'après certains critères.

C'est tout l'objet de la bibliothèque de partitionnement PARTLIB, qui autorisera des sélections multiples d'éléments appartenant à un même ensemble géométrique (carte) tout en fournissant la structure permettant de réaliser les manipulations ensemblistes : union, intersection et différence.

Pour l'instant, au-delà de la définition des groupes, la PARTLIB assure le mappage entre les numéros de groupe — entiers qui sont efficacement utilisés par un ordinateur — et les noms de groupe — qui sont plus faciles à utiliser par les êtres humains (à la DNS).

Ce programme documenté a été produit avec la version du logiciel de programmation lettrée pour le C de Donald E. Knuth (la version C développée avec Silvio Levy), j'ai nommé : CWEB

Voir <http://www-cs-faculty.stanford.edu/~knuth/cweb.html> pour de plus amples informations.

	Section	Page
Licence	1	2
Vue d'ensemble	2	3
Définitions	3	4
Limites sur le nombre des éléments	6	6
Les groupes réservés	7	7
Les structures de données	13	8
Format du fichier de mappage	52	15
Gestion des données	53	16
Informations de support : les tables de hachage	58	18
L'affectation des nouveaux numéros de groupe	64	19
Les noms de groupe	67	20
Modifier une paire/insérer une nouvelle paire	76	23
Insertion/suppression dans les tables de hachage	78	24
Ouvrir et fermer un fichier de mappage de paires	82	26
Gestion interne des fichiers	92	31
Les messages d'erreur	99	33
Programme d'exemple et de test	104	34
Index	105	37

1. Licence.

Copyright 2004-2006 Thierry LARONDE (tlaronde@polynum.com)

All rights reserved.

In what follows, AUTHORS stands for Thierry LARONDE (tlaronde@polynum.com).

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 (tlaronde@polynum.com)

Tous droits réservés.

Dans ce qui suit, le terme AUTEURS est mis pour : Thierry LARONDE (tlaronde@polynum.com).

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. Vue d'ensemble.

Dans la version originelle de GRASS, un moyen simple d'associer une ligne de texte avec des éléments géométriques (le résultat de la reconnaissance topologique) était fourni. Les routines étaient directement adaptées de celles utilisées dans le cadre des manipulations des données de quadrillage (*raster*).

Le flou dans l'utilisation des termes, `cat` et `label` étant utilisés de manière interchangeable, tantôt pour désigner l'entier, d'autres fois pour désigner le texte associé, puis la dérive naturelle consistant à utiliser ce procédé rudimentaire pour en fait associer aux géométries des attributs externes avaient rendu la définition et l'objectif de cette fonctionnalité flous.

Qui plus est, les catégories du quadrillage avaient, et ont toujours pour objectif de permettre d'associer des valeurs de type flottant aux cellules, alors que dans le cadre du vectoriel, soit les attributs sont multiples, soit ils sont uniques mais de type texte.

La nature bijective de la relation donne cependant déjà un indice : le texte est la transcription de l'entier, une version plus facilement manipulable par les humains. C'est-à-dire que la logique veut que l'entier soit un numéro de groupe (*group*), et que le texte soit son pendant textuel, le nom du groupe (*gname*), à l'instar de la correspondance existant entre les adresses IP et les noms de domaine. La nécessité de disposer d'un procédé pour identifier et grouper les éléments géométriques *par topologie*, i.e. en désignant l'objet par son emplacement (un point *sur* un objet) est fondamentale. L'association non limitée d'autres attributs (zéro ou une multitude d'attributs, de nature indéfinie) doit être fournie en utilisant ce moyen de regroupement, mais doit être externe aux manipulations géométriques (orthogonalisation).

Les liens entre éléments géométriques et éléments non géométriques (attributs textuels par exemple) sont effectués par l'intermédiaire des groupes, mais la gestion des attributs est extérieure à la manipulation géométrique. La VECTORLIB sait manipuler les géométries et identifier les groupes de géométries, mais ne connaît rien des attributs. La CORRLIB gère les correspondances entre groupes (resp. noms de groupe) et des attributs non géométriques mais ne sait rien de la nature des éléments géométriques qu'elle ne manipule que par leur "petit nom", que ce soit le *group* ou le *gname*. Dans ce dernier cas, la PARTLIB doit intervenir car c'est elle qui gère les associations (*group*, *gname*). L'interface entre VECTORLIB et CORRLIB, la "colle", ce sont les groupes et rien d'autre.

Le fichier de mappage (groupe, nom de groupe) peut être partagé par plusieurs cartes vectorielles. Un groupe/nom de groupe peut être utilisé ou inutilisé sans que cela pose de problème. Par contre il serait contre-productif, puisqu'un même fichier peut être manipulé dans diverses circonstances, qu'un processus retire des noms qu'il n'utilise pas, en en privant ainsi des cartes qui, elles, les utilisent.

En conséquence, on peut ajouter des paires ou modifier des paires, mais pas en supprimer.

Enfin, l'objectif essentiel est de fournir un moyen de consultation facile, en particulier une recherche par numéro de groupe ou nom de groupe efficace. L'insertion de nouveaux groupes sera moins fréquente que la recherche, elle pourrait donc être défavorisée par rapport à celle-ci. En fait, dans notre implémentation, l'insertion est aussi efficace.

3. Définitions.

Les éléments géométriques déduits par topologie appartiennent à un ensemble pleinement identifié par la GISDATABASE, la LOCATION, le MAPSET, le VECTOR_GEOM_SUBELT et le nom de la carte. Généralement, le nom de la carte (nom du fichier) suffit, les autres éléments étant implicitement définis par l'environnement. Cette hiérarchisation des données est réalisée dans KerGIS par une organisation en répertoires.

Sur cet ensemble, on peut définir un nombre quelconque de *partitions* distinctes, c'est-à-dire répartir les éléments géométriques en différents groupes.

Une *partition* est un ensemble de parties non vides de l'ensemble vectoriel considéré, tel que la réunion de toutes ces parties est l'ensemble, et tel qu'aucun élément n'appartient à deux parties distinctes. Dans le vectoriel, cette partition se base sur des propriétés topologiques (un point qui doit "tomber" le plus proche possible d'un et d'un seul élément de nature conforme au type attendu). L'association des éléments vectoriels aux groupes dépend de l'unité vectorielle. La partie topologique dépend donc de la VECTORLIB et est uniquement identifiée par la GISDATABASE, la LOCATION, le MAPSET, le VECTOR_PART_SUBELT et un nom. Lorsque le nom est celui du vecteur, cette partition est la partition par défaut (la première traitée par *v.support*(1)).

La traduction d'un numéro de groupe en nom de groupe, et vice-versa est faite par la présente PARTLIB qui est la partie des manipulations des partitions indépendante du module géométrique (elle pourra être utilisée également pour les données de quadrillage — RASTERLIB etc.).

Elle est pleinement identifiée par la GISDATABASE, la LOCATION, le MAPSET, le PART_ASCII_SUBELT et un nom (on ne compile pas aujourd'hui le dictionnaire, il n'existe qu'en version texte).

4. L'en-tête public donne les informations nécessaires aux programmes appelant. Une partie de la gestion est opaque et réservée à la bibliothèque elle-même. Cela pour deux motifs. Le premier : la sûreté. Un jour, certaines données seront partagées et il est nécessaire qu'un seul processus y accède : celui géré par la bibliothèque. Le deuxième : la maintenance. L'API une fois fixée ne devrait évoluer que lentement, tandis que l'implémentation peut varier drastiquement.

```

format id_kt int /* group type */
⟨part.h 4⟩ ≡
#ifndef KERGIS_PART_H
#define KERGIS_PART_H
#define PART_MAJOR 1
#define PART_MINOR 0
#define PART_REVISION 0
#include <sys/types.h>
#include "ksys/cdefs.h"
#include "ksys/types.h" /* id_kt */
#include "gis.h"
⟨Errors macros 99⟩
extern const struct G_Msgs *const P_Errors;
⟨Public macros 7⟩
⟨Public structures 13⟩
⟨Public prototypes 19⟩
#endif /* KERGIS_PART_H */

```

5. “Keep your secret secret”. Les variables partagées par plusieurs fichiers de la bibliothèque ne sont peut-être pas publiées, elles n’en restent pas moins publiques. Comme la bibliothèque est malgré tout relativement courte, on va tout placer dans un seul et même fichier.

```
#include <assert.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
  <Include files 25>
#include "gis.h"
#include "part.h"
  <Private macros 24>
  <Private structures 60>
  <Private prototypes 78>
  <Global variables 84>
```

6. Limites sur le nombre des éléments.

L'identifiant d'un élément est de type **id.kt** qui est un type *signé*. Le nombre maximal d'arcs, puisque le vectoriel de KerGIS est topologique et que KerGIS construit tous les éléments géométriques à partir d'arcs, le nombre maximal d'arcs est donc égal à la valeur positive maximale de l'entier signé **id.kt**.

Mais comme les éléments géométriques qui sont associés, regroupés, sont des *constructions topologiques*, le nombre d'éléments géométriques susceptibles d'être affectés à un groupe ne se déduit pas aussi facilement du nombre d'arcs.

Les géométries `V_GTYPE_POINT` sont des arcs d'un type particulier (`V_TTYPE_DOT`) et il y a bijection entre un arc et le point construit. Idem pour un arc de type `V_TTYPE_PATH` qui donne une géométrie `V_GTYPE_LINE`. Ces arcs n'entrent pas en compte pour la construction du dernier type de géométrie : les aires `V_GTYPE_AREA`. Une aire peut être définie par un seul arc (le noeud de fin étant identique au noeud de début). L'objectif est donc de déterminer une borne maximale sur le nombre d'aires susceptibles d'être créées afin d'évaluer les besoins en dimension de l'entier codant le groupe.

Nous considérons un vectoriel topologiquement correct (il n'y a pas d'aires se chevauchant ; les arcs sont connectés par les noeuds et forment un maillage). En effet, sans la contrainte de consistance topologique, on pourrait considérer toutes les aires définissables par un parcours fermé d'arcs, ce qui reviendrait à considérer un sous-ensemble de l'ensemble des parties de l'ensemble des arcs, c'est-à-dire un ensemble de puissance supérieure à l'ensemble initial. Ce qui ne fait pas précisément nos affaires...

Soit l'énoncé suivant, que nous démontrons par récurrence :

Théorème 1. *Pour tout ensemble dénombrable d'arcs de type `V_TTYPE_EDGE` topologiquement correct formant n aires, $n \neq 0$, il existe une injection de l'ensemble des aires dans l'ensemble des arcs, associant de manière unique à toute aire l'un des arcs la composant.*

Démonstration : l'énoncé est trivialement vérifié pour $n \equiv 1$ puisqu'il suffit d'associer l'un des arcs (il en existe au moins un, éventuellement fermé sur lui-même). Il l'est également pour $n \equiv 2$ puisque si deux aires sont distinctes, c'est qu'il existe deux parcours fermés différents donc au minimum deux arcs (le cas minimal étant celui de deux îles ou de deux aires partageant non un arc mais un noeud).

Supposons l'énoncé vrai pour $n \geq 2$. Soit un ensemble produisant $n + 1$ aires. Il suffit de retirer l'un des arcs composant l'une des aires pour que le nombre d'aires soit de n ou de $n - 1$, ce dernier cas se produisant si la frontière entre deux aires est composée de plus d'un arc (on aurait pu simplifier en imposant la non existence de noeuds de multiplicité 2 connectant deux arcs distincts — un arc isolé fermé sur lui-même possède un noeud de multiplicité 2 puisqu'il est à la fois début et fin ; deux arcs fermés sur eux-mêmes partageant le noeud sont donc accrochés à un noeud de multiplicité 4, etc.).

L'énoncé étant vrai pour n ou $n - 1$, il existe donc une injection recherchée pour laquelle l'arc retiré (dans le cas n aires) ou l'arc retiré et l'un des arcs subsistant non connecté (cas $n - 1$) ne font pas partie de l'ensemble image puisqu'ils ne participent pas à la définition d'une aire. Il suffit donc d'associer l'arc retiré à la nouvelle parcelle, ou l'arc retiré à l'une des nouvelles parcelles, et l'arc non connecté à l'autre pour étendre l'injection à l'ensemble considéré. QED.

Nous pouvons donc également utiliser **id.kt** pour les groupes, et nous pouvons disposer des valeurs négatives pour coder autre chose que les groupes en eux-mêmes.

7. Les groupes réservés.

Nous allons définir, par facilité, certains numéros de groupes pour désigner des groupes habituels. Dans certains cas, il ne s'agit pas de groupes effectifs, mais d'un moyen rudimentaire d'opérer une sélection. En effet, puisque la plage négative est disponible, on peut se permettre de coder les *compléments* d'un groupe, ou le groupe de l'ensemble de tous les éléments du vecteur etc. Cela ne met pas en cause notre définition de la partition : un élément appartiendra à un et un seul groupe défini dans le fichier de partition. Mais des conventions de langage nous permettent des désignations supplémentaires en jouant sur les valeurs qui ne peuvent pas être affectées à un groupe valide.

On codera simplement le complément d'un groupe en prenant le complément binaire du numéro de groupe (le bit de poids le plus fort sera donc toujours positionné pour un complément de groupe).

```
<Public macros 7> ≡
#define P_GCOMPLEMENT(gid) ~(gid)
```

See also sections 8, 9, 10, 11, 16, 20, and 34.

This code is used in section 4.

8. Comme les groupes sont assignés par l'utilisateur, autant faire en sorte que les groupes normaux soient les plus faciles à identifier. De plus, il existe une pratique historique, celle d'assigner la valeur 0 au groupe de tous les éléments qui n'appartiennent pas à un groupe déterminé. Ce groupe sera identifié par la macro P_GROUP_OTHER à laquelle nous donnons la valeur établie par l'usage (qui apparaît pertinente qui plus est).

Cependant la macro a pour seule destination d'expliciter dans le code le groupe manipulé. La valeur de 0 est cruciale et constitue l'un des postulats sur lesquels est bâti le code.

```
<Public macros 7> +≡
#define P_GROUP_OTHER 0 /* 0 and nothing else! don't change ! */
```

9. P_GROUP_DEFINED sera l'ensemble des éléments appartenant à un groupe défini (autre que P_GROUP_OTHER). C'est donc simplement le complément de celui-ci.

```
<Public macros 7> +≡
#define P_GROUP_DEFINED P_GCOMPLEMENT(P_GROUP_OTHER)
```

10. L'ensemble vide peut être utile également, par exemple comme valeur pour coder dans une base de données les attributs qui ne sont associés à aucune géométrie (pour des géométries disparues, pour la gestion de stocks, etc.).

Afin de laisser à l'utilisateur la plage naturelle (démarrant à 1 et croissant), nous allons coder P_GROUP_EMPTY comme la limite maximale de notre demi-intervalle (intervalle positif).

```
<Public macros 7> +≡
#define P_GROUP_EMPTY(id_kt) ((unsigned) ~0 >> 1)
```

11. Le complément de l'ensemble vide, c'est l'ensemble de tous les éléments.

```
<Public macros 7> +≡
#define P_GROUP_ALL P_GCOMPLEMENT(P_GROUP_EMPTY)
```

12. À noter qu'à première vue, nous pourrions, dans un cas extrême manquer non pas de deux (P_GROUP_OTHER et P_GROUP_EMPTY) mais d'un seul (P_GROUP_EMPTY) numéro de groupe si le nombre d'éléments dans le vectoriel était maximal.

En effet, aucun indice d'élément n'est nul, donc P_GROUP_OTHER n'empiète pas sur la plage possible. Mais, à tout prendre, l'élément non affecté parce qu'il n'y aurait plus de numéro de groupe disponible, tomberait dans P_GROUP_OTHER qui est, pour finir, un groupe aussi bon qu'un autre...

13. Les structures de données.

La structure de base est la paire constituée d'un entier et d'un nom.

```

<Public structures 13> ≡
struct P_Pair {
    id_kt group;
    char *gname;
};

```

See also sections 14 and 15.

This code is used in section 4.

14. Les données consistent en une liste de paires.

Des informations supplémentaires, constituant l'en-tête et qui seront placées dans le fichier texte situé dans PART_ASCII_SUBELT (car il n'y a pas de version compilée, seulement une version texte lue à l'ouverture et conservée intégralement en mémoire), sont de trois ordres : des méta-informations (la description) ; des informations permettant de vérifier l'intégrité des données ; et des informations modifiant le comportement des générations.

```

<Public structures 13> +≡
struct P_Header {
    <Header members 17>
};

```

15. À l'ouverture d'un fichier de mappage, une structure *P_Map* est passée par l'appelant. Dans cette structure on doit trouver bien évidemment le nom de la partition, une structure **P_Header** qui n'est utilisée qu'en création (sinon elle est complétée par les informations trouvées dans le fichier existant) et enfin le *handler* qui est renseigné (entier positif si l'ouverture a réussi, -1 sinon). Dans toutes les invocations suivantes, c'est ce handler qui permettra à la bibliothèque d'identifier les données à manipuler.

Les ouvertures se font via la LIBGIS et suivant le réglément défini par celle-ci. Les *modes* sont donc ceux de la LIBGIS : G_O_RDONLY, G_O_WRONLY ou G_O_RDWR. Suivant notre politique, *P_open* renvoie un entier comme code de retour, les données étant renvoyées via les pointeurs sur les structures.

```

<Public structures 13> +≡
struct P_Map {
    char *name;
    int handler;
    struct P_Header Header;
};

```

16. Comme le format peut changer, l'information sur la version doit être présente. Mais il est inutile de la positionner en création : elle sera déterminée par la version de la bibliothèque créant le fichier.

```

<Public macros 7> +≡
#define P_VERSION "1.0"

```

17.

```

<Header members 17> ≡
char *version;

```

See also sections 22, 26, 32, 38, 45, and 49.

This code is used in section 14.

18.

⟨ Set default Header values 18 ⟩ ≡
Mapping-Header.version ← P_VERSION;

See also sections 23, 27, 39, and 46.

This code is used in section 85.

19. Les paires seront gérées via les routines publiques. La modification de la gestion des données (ce qui est dans l'en-tête), plus rare et spécifique, sera opérée via *P_ioctl*.

P_ioctl ne reprend pas la structure classique de `ioctl(2)`, car *de facto* toutes les informations susceptibles d'être modifiées sont présentes dans *Mapping-Header*. La requête consiste donc à demander de prendre en compte les modifications placées dans cet en-tête, ou à renvoyer des renseignements dans l'en-tête.

⟨ Public prototypes 19 ⟩ ≡
int *P_ioctl*(**struct** P_Map **Mapping*, **unsigned long** *request*);

See also sections 70, 72, 76, 83, and 89.

This code is used in section 4.

20.

⟨ Public macros 7 ⟩ +≡
 ⟨ `ioctl` macros 29 ⟩

21.

```
int P_ioctl(struct P_Map *Mapping, unsigned long request)
{
    _CHECK_HANDLER;

    switch (request) {
        ⟨ ioctl requests 30 ⟩
        default: return P_EIOCTLUNKNOWNREQUEST;
    }
}
```

22. L'en-tête peut comporter un titre (une description). Comme nous n'en faisons rien de spécifique durant le processus, il suffit de le mettre à jour dans la structure *Mapping* pour qu'il soit écrit à la fermeture.

⟨ Header members 17 ⟩ +≡
char **description*;

23.

⟨ Set default Header values 18 ⟩ +≡
if (*Mapping-Header.description* ≡ Λ) *Mapping-Header.description* ← *G_store*("Generated_by_PARTLIB");

24. Pour des motifs d'efficacité et d'économie de mémoire, les noms de groupe devront avoir une longueur maximale. Cette longueur maximale sera définie à la création (et pourra être changée par édition du fichier si elle est trop large, ou augmentée si l'on prévoit des noms plus longs). Par contre, une fois l'initialisation terminée et le fichier ouvert en mémoire, comme toutes nos structures de gestion en dépendent, il n'est pas question de la changer en cours de route.

Cette longueur comporte le caractère nul final. Par défaut, la longueur est le maximum entre 64 et la longueur maximale de la représentation décimale du numéro de groupe.

Comme en représentation décimale pour un entier nous avons :

$$a_0 * 10^0 + a_1 * 10^1 + \dots + a_n * 10^n$$

la longueur de la représentation est donc la partie entière de la puissance + 1 (il y a $n + 1$ chiffres).

⟨ Private macros 24 ⟩ ≡

```
#define _P_MAX_DEC_LENGTH(size_t) floor(sizeof(id_kt) * CHAR_BIT * M_LN2 / M_LN10) + 1
```

See also sections 54, 57, 94, and 96.

This code is used in section 5.

25. Par sécurité, on recopie la valeur placée dans l'en-tête dans la structure privée (si la valeur a été modifiée dans la structure pointée par *Mapping* et que nous l'utilisons aveuglément, la catastrophe est en vue).

⟨ Include files 25 ⟩ ≡

```
#include <sys/types.h>
```

```
#include <limits.h>
```

```
#include <math.h>
```

See also section 97.

This code is used in section 5.

26.

⟨ Header members 17 ⟩ +≡

```
size_t gname_max_len; /* including trailing nil */
```

27.

⟨ Set default Header values 18 ⟩ +≡

```
if (Mapping-Header.gname_max_len == 0)
```

```
Mapping-Header.gname_max_len ← (_P_MAX_DEC_LENGTH + 1 > 64) ? _P_MAX_DEC_LENGTH + 1 : 64;
```

28.

⟨ Pairs management 28 ⟩ ≡

```
size_t gname_max_len; /* this copy is used */
```

See also sections 31, 33, 41, 53, 55, and 61.

This code is used in section 92.

29.

⟨ ioctl macros 29 ⟩ ≡

```
#define P_IOCTL_SET_GNAME_MAX_LEN °I
```

See also sections 35, 42, 47, and 50.

This code is used in section 20.

30.

```

< ioctl requests 30 > ≡
case P_IOCTL_SET_GNAME_MAX_LEN:
    if (_HANDLER-initialized) return P_EIOCTLBADREQUEST;
    if (Mapping-Header.gname_max_len < strlen(Mapping-Header.prefix) + _P_MAX_DEC_LENGTH + 1)
        return P_EIOCTLLENTOOSHORT;
    _HANDLER-gname_max_len ← Mapping-Header.gname_max_len;
    return G_SUCCESS;

```

See also sections 36, 43, 48, and 51.

This code is used in section 21.

31. Si tous les noms doivent être des noms automatiques, il est inutile de les conserver. En conséquence, des drapeaux sont nécessaires pour indiquer si la génération est totalement automatique, ou pas. Mais c'est tout l'un ou tout l'autre : soit tous les noms sont automatiques et rien n'est stocké, soit certains ne sont pas automatiques et tout est stocké.

C'est-à-dire que le mécanisme pour générer un nom automatique fonctionne même si la génération totalement automatique n'est pas demandée, mais dans ce cas le nom généré est entré dans la liste de telle façon qu'un changement de préfixe n'ait aucune influence sur l'intégrité des données. Par ailleurs, si le nom automatiquement généré entre en conflit avec un nom existant, il sera rejeté.

La requête pour modifier le comportement n'a de sens qu'avant l'injection des données. Nous ajoutons donc un sémaphore dans la structure interne qui bloquera les modifications lorsqu'elles ne sont plus possibles (dès lors que des données ont été entrées).

```

< Pairs management 28 > +≡
int initialized;

```

32. La requête ioctl est utilisée donc à la lecture de l'en-tête.

Comme pour les autres données qui sont susceptibles d'être utilisées en interne mais qui sont accessibles en modification par le programme appelant (sans qu'on ait l'assurance qu'une requête ioctl sera lancée en cas de modification) les drapeaux sont recopiés dans la structure interne.

```

< Header members 17 > +≡
int flags;

```

33.

```

< Pairs management 28 > +≡
int pairs_flags;

```

34.

```

< Public macros 7 > +≡
#define P_FLAGS_AUTOMATIC 0

```

35.

```

< ioctl macros 29 > +≡
#define P_IOCTL_SET_FLAGS 02

```

36.

```

< ioctl requests 30 > +≡
case P_IOCTL_SET_FLAGS:
    if (_HANDLER-initialized) return P_EIOCTLBADREQUEST;
    _HANDLER-pairs_flags ← Mapping-Header.flags;
    return G_SUCCESS;

```

37. À noter que notre initialisation respecte un ordre essentiel. En effet, la génération automatique aura des conséquences sur la gestion (stockage) du nom de `P_GROUP_OTHER`.

38. Notre définition de la partition impose l'unicité des noms. Il n'y aura pas de groupe anonyme.

Notre volonté de faciliter la génération automatique doit prendre en compte cette contrainte d'unicité. Le plus simple est, non pas d'autoriser un format (comme les catégories de quadrillage) mais de toujours composer le nom auto-généré à partir de la transcription décimale du numéro de groupe : on est sûr d'avoir l'unicité.

Par conséquent nous introduisons comme donnée d'en-tête un *prefix*, le nom auto-généré étant la concaténation du *prefix* et de la représentation décimale du numéro de groupe. Par défaut le préfixe est la chaîne vide (ce qui associe au numéro de groupe sa représentation décimale).

```
<Header members 17> +≡
char *prefix;
```

39.

```
<Set default Header values 18> +≡
if (Mapping-Header.prefix ≡ Λ) Mapping-Header.prefix ← G_store("");
```

40. Puisque le préfixe est donné et que la longueur maximale de la représentation décimale du numéro de groupe se calcule, il est inutile de conserver, à part, le préfixe : quand on le change, on alloue un tampon *buffer* suffisamment grand, et on recopie le préfixe au début. Il suffit alors de conserver l'information sur l'index de fin de préfixe *prefix_end* pour concaténer à la demande le numéro de groupe.

41.

```
<Pairs management 28> +≡
char *buffer; /* to create an automatic name; compute once */
char *prefix_end; /* pointer to the end of the prefix in buffer */
```

42.

```
<ioctl macros 29> +≡
#define P_IOCTL_SET_PREFIX °4
```

43.

```
<ioctl requests 30> +≡
case P_IOCTL_SET_PREFIX:
if (Mapping-Header.prefix ≡ Λ) return P_EIOCTLBADREQUEST;
{
size_t len;
len ← strlen(Mapping-Header.prefix);
if (len + _P_MAX_DEC_LENGTH + 1 > _HANDLER-gname_max_len) return P_EPREFIXTOOLONG;
free(_HANDLER-buffer);
_HANDLER-prefix_end ← Λ;
if ((_HANDLER-buffer ← (char *) malloc(len + _P_MAX_DEC_LENGTH + 1)) ≡ Λ)
return (errno ≪ G_ERROR_SHIFT);
(void) strcpy(_HANDLER-buffer, Mapping-Header.prefix);
_HANDLER-prefix_end ← _HANDLER-buffer + len;
}
return G_SUCCESS;
```

44. La génération d'un nom automatique s'opère donc simplement en recopiant la représentation décimale du groupe après le préfixe, ce qui revient à concaténer les deux chaînes.

La recopie du buffer est à la charge de l'appelant. Le morceau de code suivant ne fait que créer le nom.

```
< Set default gname in _HANDLER-buffer 44 > ≡
  (void) sprintf(_HANDLER-prefix_end, "%1d", (long) Pair-group);
```

This code is used in sections 48, 68, and 69.

45. Il existe un groupe toujours défini : P_GROUP_OTHER. En conséquence, ce groupe réservé possède des informations placées dans l'en-tête. Par défaut, son nom est NO_DATA. Cela peut être changé, à la création en plaçant le nom correct dans l'en-tête, en cours de manipulation via une requête ioctl sauf dans le cas P_FLAGS_AUTOMATIC : le nom est le nom automatique car nous ne stockons rien.

```
< Header members 17 > +≡
  char *group_other;
```

46.

```
< Set default Header values 18 > +≡
  if (Mapping-Header.group_other ≡ Λ) Mapping-Header.group_other ← G_store("NO_DATA");
```

47.

```
< ioctl macros 29 > +≡
#define P_IOCTL_SET_GROUP_OTHER °10
```

48.

```
< ioctl requests 30 > +≡
case P_IOCTL_SET_GROUP_OTHER:
  if (_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC) return P_EIOCTLBADREQUEST;
  {
    struct P_Pair _Pair;
    struct P_Pair *Pair;
    Pair ← &_Pair;
    Pair-group ← 0;
    if (Mapping-Header.group_other ≡ Λ) {
      < Set default gname in _HANDLER-buffer 44 >
      Mapping-Header.group_other ← G_store(_HANDLER-buffer);
    }
    if (*Mapping-Header.group_other ≡ '\0') return P_EBADGNAME;
    if (strlen(Mapping-Header.group_other) + 1 > _HANDLER-gname_max_len) return P_EBADGNAME;
    if (!_HANDLER-initialized) {
      _HANDLER-blocks ← (char **) realloc(_HANDLER-blocks, 1 * sizeof(char *));
      _HANDLER-initialized ← 1;
      if ((_HANDLER-blocks[0] ← (char *) calloc(1, _P_BLOCK_SIZE * _HANDLER-gname_max_len)) ≡ Λ)
        return errno << G_ERROR_SHIFT;
    }
    Pair-gname ← Mapping-Header.group_other;
    if (P_find_by_gname(Mapping, &_Pair) ≡ G_SUCCESS) return P_EEXIST;
    Pair-gname ← _P_GNAME(0);
    if (*Pair-gname ≠ '\0') _P_nremove(Mapping, 0);
    _P_COPY(0, Mapping-Header.group_other);
    _P_ninsert(Mapping, 0);
    return G_SUCCESS;
  }
```

49. Une dernière information est utile à l'appelant (par exemple lorsqu'il s'agit de 'dumper' le fichier de mappage en un autre format) : le plus grand numéro de groupe attribué. À l'issue de l'initialisation, cette information sera recopiée dans *Header.group_max*.

Par contre il nous faut une convention dans le cas de la génération totalement automatique (puisque la seule limite c'est le potentiel de **id.kt** pour sa partie positive). Nous renverrons simplement `P_GROUP_EMPTY` et il suffira à l'appelant de tester cette valeur pour savoir que, probablement, il peut laisser tomber : les numéros de groupes se suffisent à eux-mêmes.

Par contre, comme il s'agit d'une variable (dans le cas de l'écriture), une requête `ioctl` est nécessaire pour mettre à jour la donnée.

```
< Header members 17 > +≡
    id.kt group_max;
```

50.

```
< ioctl macros 29 > +≡
#define P_IOCTL_GET_GROUP_MAX °20
```

51.

```
< ioctl requests 30 > +≡
case P_IOCTL_GET_GROUP_MAX:
    if (_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC) Mapping-Header.group_max ← P_GROUP_EMPTY;
    else Mapping-Header.group_max ← _HANDLER-group_max;
    return G_SUCCESS;
```

52. Format du fichier de mappage.

Le format a été introduit ci-dessus. Voici la syntaxe.

$\langle statements \rangle \rightarrow \langle header \rangle \langle pairs \rangle$

$\langle header \rangle \rightarrow \langle version \rangle \langle opts_header \rangle$

$\langle version \rangle \rightarrow \text{VERSION ':' VERSION_NB}$

$\langle opts_header \rangle \rightarrow /* \text{ empty } */ | \langle opts_header \rangle \langle opt_header \rangle$

$\langle opt_header \rangle \rightarrow \langle description \rangle | \langle group_other \rangle | \langle prefix \rangle | \langle gname_max_len \rangle$

$\langle description \rangle \rightarrow \text{DESCRIPTION ':' STRING}$

$\langle prefix \rangle \rightarrow \text{PREFIX ':' STRING}$

$\langle group_other \rangle \rightarrow \text{GROUP_OTHER ':' STRING}$

$\langle gname_max_len \rangle \rightarrow \text{GNAME_MAX_LEN ':' INT}$

$\langle pairs \rangle \rightarrow /* \text{ empty } */ | \langle pairs \rangle \langle pair \rangle$

$\langle pair \rangle \rightarrow \text{INT ',' STRING}$

53. Gestion des données.

Nous désirons donc disposer d'une gestion efficace de la mémoire, et bénéficier d'une recherche rapide des paires, que ce soit par numéro ou par nom.

Les numéros de groupe étant des entiers, le plus efficace est de s'en servir comme index, c'est-à-dire de déduire par un calcul simple l'adresse du nom associé en fonction de la valeur du numéro de groupe.

Mais afin d'économiser la mémoire, de la gérer plus efficacement (c'est-à-dire de ne pas étendre des vecteurs au coup par coup) ainsi que d'autoriser des définitions clair-semées (non contiguës), nous allons gérer les paires par blocs.

Puisque le groupe sert d'index, la seule information à stocker est celle du nom associé. Comme indiqué plus haut, toujours pour permettre une gestion plus efficace de la mémoire, les noms ont une longueur maximale définie par `_HANDLER-gname_max_len` (en fait un caractère de moins puisque le dernier est le caractère nul afin de pouvoir utiliser les fonctions normales de manipulation des chaînes). On alloue donc un bloc de `_P_BLOCK_SIZE * _HANDLER-gname_max_len` de caractères (et l'initialisation s'assure que tout est à zéro).

Il nous faut par contre une convention pour indiquer qu'un groupe n'est pas utilisé (puisque toutes les entrées d'un bloc sont présentes). Puisque le nom n'est pas stocké sous forme de pointeur vers un vecteur de caractères, mais directement sous forme de vecteur de caractères, la chaîne vide s'impose (puisque'il n'y a pas de groupe anonyme et que dans le cas de la génération automatique la consultation n'aura pas lieu).

⟨ Pairs management 28 ⟩ +≡

```
char **blocks; /* blocks of _P_BLOCK_SIZE * _HANDLER-gname_max_len */
```

54. Le calcul s'effectue alors facilement. `group/_P_BLOCK_SIZE` donnera le numéro du bloc, tandis que `(group % _P_BLOCK_SIZE) * _HANDLER-gname_max_len` donne l'index de l'entrée dans ce bloc.

⟨ Private macros 24 ⟩ +≡

```
#define _P_BLOCK_SIZE 128
```

```
#define _P_BLOCK_NB(group) ((size_t) group/_P_BLOCK_SIZE)
```

```
#define _P_GNAME(group)
```

```
(_HANDLER-blocks[_P_BLOCK_NB(group)] + ((size_t) group % _P_BLOCK_SIZE) * _HANDLER-gname_max_len)
```

55. Pour effectuer correctement notre gestion nous avons besoin d'une information : celle de la longueur de la liste des blocs.

Même si la longueur de la liste possède une entrée pour le bloc, il n'est en rien obligatoire que le bloc soit alloué : les définitions peuvent être clair-semées, le groupe 304 étant alloué alors qu'aucun dans la plage [128, 255] ne l'a été.

Comment conserver cette information ? Tout simplement en conservant l'information sur le groupe maximal affecté, information qui nous servira également pour l'insertion automatique : si une nouvelle paire doit être inséré mais que l'appelant nous laisse le choix du numéro du groupe, on affecte simplement `group_max + 1`.

⟨ Pairs management 28 ⟩ +≡

```
id_kt group_max; /* maximum group number in use */
```

56. En conservant la valeur maximale du numéro de groupe utilisé, nous avons les informations nécessaires pour décider de l'extension de la liste. Même s'il n'est pas nécessaire d'étendre la liste (la liste peut être assez longue pour comporter le pointeur sur le bloc, mais le bloc n'est pas présent) encore faut-il l'allouer s'il n'est pas présent.

Compte tenu de nos macros (et du fait que nous manipulons des entiers non signés via `size_t`), le problème de l'initialisation se pose pour le groupe `P_GROUP_OTHER` qui existe toujours. Mais ce problème est traité par l'initialisation via `P_ioctl` de `P_GROUP_OTHER`. C'est à ce moment-là que, dans le cas de la non génération automatique, le premier bloc est alloué pour y insérer le nom de `P_GROUP_OTHER` afin de conserver l'unicité des noms : ce nom-là, comme les autres, doit être unique et il doit donc être présent dans nos structures.

Ce cas particulier étant traité à part, le fragment de code qui suit s'occupe du cas général (pour lequel le bloc 0 existe déjà, ce qui nous permet de fonctionner même avec `group_max` \equiv 0).

```

⟨ Extend blocks if needed and allocate new block if not present 56 ⟩ ≡
{
    char **p;
    size_t i;
    if (_P_BLOCK_NB(Pair→group) > _P_BLOCK_NB(_HANDLER→group_max)) {
        if ((p ← (char **) realloc(_HANDLER→blocks, (_P_BLOCK_NB(Pair→group) + 1) * sizeof(char *))) ≡ Λ)
            return errno ≪ G_ERROR_SHIFT;
        _HANDLER→blocks ← p;
        for (i ← _P_BLOCK_NB(_HANDLER→group_max) + 1; i < _P_BLOCK_NB(Pair→group) + 1; ++i)
            _HANDLER→blocks[i] ← Λ;
        _HANDLER→group_max ← Pair→group;
    }
    if (_HANDLER→blocks[_P_BLOCK_NB(Pair→group)] ≡ Λ) {
        if ((_HANDLER→blocks[_P_BLOCK_NB(Pair→group)] ← (char *) calloc(1,
            _P_BLOCK_SIZE * _HANDLER→gname_max_len)) ≡ Λ) return errno ≪ G_ERROR_SHIFT;
    }
}

```

This code is used in section 77.

57. La copie d'un nouveau nom de groupe à son emplacement (à la charge pour l'appelant de vérifier préalablement que ce nouveau nom n'excède pas la longueur maximale) peut s'effectuer facilement.

```

⟨ Private macros 24 ⟩ +≡
#define _P_COPY(group, gname) strncpy (_P_GNAME(group), gname, _HANDLER→gname_max_len - 1)

```

58. Informations de support : les tables de hachage.

Ce que nous avons vu précédemment, ce sont les données brutes. À l'instar de la gestion du vectoriel dans KerGIS, des informations supplémentaires sont construites afin de faciliter les recherches. En l'occurrence ici, puisqu'il s'agit de vérifier des textes, ces informations supplémentaires sont des tables de hachage pour accélérer les comparaisons.

59. Puisque ces informations sont supplémentaires et construites, nos données cruciales sont et restent les paires, c'est-à-dire les listes de blocs. En conséquence, quand on ajoute, on commence par mettre à jour les paires, puis on construit le support.

Quand on modifie, on retire les informations de support, avant de modifier la paire puis de reconstruire le support.

60. Nous n'allons pas dupliquer les noms dans la table de hachage, puisque ces noms sont déjà stockés dans nos blocs. Il suffit d'enregistrer les pointeurs. Or, par construction, les numéros de groupe peuvent donner directement l'adresse. On stocke donc le numéro de groupe.

Pour la recherche, nous utiliserons le hachage avec une simple liste doublement liée.

La fonction de hachage provient du source de CWEB de Donald Knuth.

```
<Private structures 60> ≡
struct _P_Name {
    id_kt group;
    struct _P_Name *Next;
    struct _P_Name *Prev;
};
```

See also section 92.

This code is used in section 5.

61.

```
<Pairs management 28> +≡
#define HASH_SIZE 353 /* prime */
struct _P_Name *hashes[HASH_SIZE];
```

62. Le calcul de hachage est effectué par la formule suivante :

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \dots + c_n) \bmod \text{HASH_SIZE}.$$

Dans tous les morceaux de code appelant ce fragment devra être utilisé en déclarant un *unsigned int h*.

```
<Compute hash value of gname 62> ≡
{
    unsigned char *i;
    i ← (unsigned char *) gname;
    h ← *i;
    while (*++i) h ← (2 * h + *i) % HASH_SIZE;
}
```

This code is used in sections 74, 79, and 81.

63. L'initialisation des tables de hachage consiste simplement à affecter des *NULLs* pour chaque entrée du tableau de taille fixe. En fait, on utilisera `calloc(3)` pour s'assurer que toutes les valeurs sont à zéro.

64. L'affectation des nouveaux numéros de groupe.

Pour ce qui concerne les routines de la bibliothèque en elle-même, elle ne connaîtra que les valeurs entières. Les compléments des groupes sont en fait destinés à la CORRLIB pour permettre de coder des attributs partagés. Un langage propre sera développé pour permettre les manipulations ensemblistes, langage qui ne s'appuiera pas sur les groupes négatifs pour la négation (ces groupes sont simplement du sucre syntaxique : c'est possible, la possibilité est implémentée, elle a ses usages ; mais ses capacités sont limitées).

```
< group is valid or return an error 64 > ≡
  if (Pair→group < 0) return P_EBADGROUP;
```

This code is used in sections 71 and 77.

65. Toujours afin de simplifier l'affectation, si lors d'une demande de modification d'une paire on a $Pair→group ≡ 0$, on traduit la demande de modification en demande d'insertion, le numéro de groupe étant alors automatiquement $group_max + 1$ si $group_max < P_GROUP_EMPTY - 1$, c'est-à-dire si l'on n'a pas atteint le maximum de groupes que l'on est capable de coder.

```
< Set new group number if Pair→group ≡ 0 ∧ ¬(_HANDLER→pairs_flags & P_FLAGS_AUTOMATIC) 65 > ≡
  if (Pair→group ≡ 0 ∧ ¬(_HANDLER→pairs_flags & P_FLAGS_AUTOMATIC)) {
    if (_HANDLER→group_max < P_GROUP_EMPTY - 1) Pair→group ← _HANDLER→group_max + 1;
    else return P_EMAXNBGROUPS;
  }
```

This code is used in section 77.

66. Comme indiqué plus haut, cela signifie que le traitement du groupe 0 (P_GROUP_OTHER) ne pourra pas être fait via P_modify qui utilise le fragment précédent mais par la requête ioctl définie plus haut.

67. Les noms de groupe.

Il faut prendre soin des allocations de mémoire, puisque le nom de groupe est un *char* *.

La PARTLIB ne renvoie *jamais* un pointeur sur ses données internes, mais un pointeur sur une *copie* de ses données internes. La modification de cette copie n'affecte donc en rien la bibliothèque. Mais il est de la responsabilité de l'appelant de libérer cet emplacement mémoire s'il est inutilisé par la suite.

Par construction, nous recopions le nom de groupe dans nos structures internes. L'appelant peut donc faire ce qu'il veut de la chaîne qu'il a transmise.

68. De par la définition d'une partition, le *gname* est unique : il ne peut exister deux groupes différents avec le même *gname*. La tentative d'introduction d'une paire dans laquelle le nom de groupe serait identique à un nom de groupe existant échouera.

Et il n'existe pas de groupe anonyme.

```

⟨gname is valid or return an error 68⟩ ≡
{
  struct P_Pair _Pair;
  struct P_Pair *Pair_p;
  _Pair ← *Pair;
  Pair_p ← &_Pair;
  if (Pair→gname ≡ Λ) {
    ⟨Set default gname in _HANDLER→buffer 44⟩
    Pair→gname ← G_store(_HANDLER→buffer);
  }
  if (*Pair→gname ≡ '\0') return P_EBADGNAME;
  if (P_find_by_gname(Mapping, Pair_p) ≡ G_SUCCESS) return P_EEXIST;
}

```

This code is used in section 77.

69. Récupérer une paire dans la liste à partir du groupe consiste donc à s'assurer dans un premier temps que le groupe tombe dans un bloc affecté, et deuxièmement à vérifier si la paire existe ce qui signifie, d'après nos conventions, que **gname* ≠ '\0'.

```

⟨Get gname associated with Pair→group 69⟩ ≡
{
  if (_HANDLER→pairs_flags & P_FLAGS_AUTOMATIC) {
    ⟨Set default gname in _HANDLER→buffer 44⟩
    gname ← _HANDLER→buffer;
  }
  else if (_P_BLOCK_NB(Pair→group) > _P_BLOCK_NB(_HANDLER→group_max) ∨
    _HANDLER→blocks[_P_BLOCK_NB(Pair→group)] ≡ Λ) gname ← Λ; /* not here
    */
  else {
    gname ← _P_GNAME(Pair→group);
    if (*gname ≡ '\0') gname ← Λ;
  }
}

```

This code is used in section 71.

70. Bien entendu, nous créons une fonction effectuant précisément la recherche par numéro de groupe.

P_find_by_group renvoie dans *Pair-gname* une copie du nom trouvé, si la paire existe. Sinon, une erreur est renvoyée, et *Pair-gname* $\equiv \Lambda$.

Il est de la responsabilité de l'appelant de libérer la mémoire éventuellement utilisée par *Pair-gname*. La fonction se contente d'y stocker le pointeur vers une copie de la chaîne (pas un pointeur vers ses structures internes) et se contente d'écraser la valeur précédente.

⟨ Public prototypes 19 ⟩ +≡

```
int P_find_by_group(struct P_Map *Mapping, struct P_Pair *Pair);
```

71.

```
int P_find_by_group(struct P_Map *Mapping, struct P_Pair *Pair)
{
    char *gname;
    _CHECK_HANDLER;

    Pair-gname  $\leftarrow \Lambda$ ;
    ⟨ group is valid or return an error 64 ⟩
    ⟨ Get gname associated with Pair-group 69 ⟩
    if (gname  $\equiv \Lambda$ ) return P_EGROUPNOTFOUND;
    else Pair-gname  $\leftarrow$  G_store(gname);
    return G_SUCCESS;
}
```

72. Symétriquement, on doit pouvoir trouver le groupe associé à un certain nom.

Une erreur est renvoyée si le groupe n'est pas trouvé et *Pair-group* \leftarrow P_GROUP_EMPTY. Sinon *Pair-group* est évidemment renseigné avec le bon numéro.

Si *_HANDLER-hashes[h]* $\equiv \Lambda$, c'est-à-dire que la liste des chaînes correspondant au hachage est vide, le problème est réglé.

Si, par contre, il existe des entrées enregistrées, on recherche une identité.

⟨ Public prototypes 19 ⟩ +≡

```
int P_find_by_gname(struct P_Map *Mapping, struct P_Pair *Pair);
```

73.

```
int P_find_by_gname(struct P_Map *Mapping, struct P_Pair *Pair)
{
    _CHECK_HANDLER;

    Pair-group  $\leftarrow$  P_GROUP_EMPTY;
    if (Pair-gname  $\equiv \Lambda$ ) return P_EBADGNAME;
    if (_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC) ⟨ Compare gname with automatic name and return 75 ⟩
    else {
        unsigned int h; /* hash value */
        struct _P_Name *p;
        char *gname;

        if (strlen(Pair-gname)  $\geq$  _HANDLER-gname_max_len) return P_EGNAMENOTFOUND;
        ⟨ Compare gname against hashes and return if found 74 ⟩
    }
    return P_EGNAMENOTFOUND;
}
```

74. La comparaison par rapport aux tables de hachage consiste simplement à parcourir le vecteur correspondant au hachage et à vérifier via *strcmp(3)*.

```

⟨ Compare gname against hashes and return if found 74 ⟩ ≡
  gname ← Pair-gname;
  ⟨ Compute hash value of gname 62 ⟩
  if ((p ← _HANDLER-hashes[h]) ≠ Λ) {
    for (; p ≠ Λ; p ← p-Next) {
      gname ← _P_GNAME(p-group);
      if (strcmp(gname, Pair-gname) ≡ 0) {
        Pair-group ← p-group;
        return G_SUCCESS;
        break;
      }
    }
  }
}

```

This code is used in section 73.

75. Dans le cas de la génération automatique, il s'agit dans un premier temps de vérifier que le préfixe est là puis de s'assurer que la conversion du reste donne un entier valide.

```

⟨ Compare gname with automatic name and return 75 ⟩ ≡
{
  size_t i;
  char c;
  id_kt group;
  i ← _HANDLER-prefix_end - _HANDLER-buffer;
  if (strncmp(_HANDLER-buffer, Pair-gname, i - 1) ≡ 0) {
    if (sscanf(Pair-gname + i, "%1d%c", &group, &c) ≡ 1) {
      Pair-group ← group;
      return G_SUCCESS;
    }
  }
  else return P_EGNAMENOTFOUND;
}

```

This code is used in section 73.

76. Modifier une paire/insérer une nouvelle paire.

Si *Pair-group* $\equiv 0$, il s'agit d'une requête pour insérer une nouvelle paire. Sinon, d'une requête pour modifier le nom associé au groupe.

Parce que le fichier de mappage peut être partagé par différents fichiers vectoriels (en attendant d'autres types), il ne s'agit pas qu'un programme supprime des entrées parce que le fichier qu'il considère, sur le moment, n'en a pas besoin.

Les noms ont entre autres pour but d'enlever toute signification intrinsèque aux nombres (l'implémentation des nombres clair-semés autorise en fait une utilisation basée sur les numéros ; mais c'est une possibilité, pas une recommandation).

Les manipulations concernent donc essentiellement les listes de hachage, la mise à jour de l'information sur la paire dans les blocs étant simple.

⟨ Public prototypes 19 ⟩ +≡

```
int P_modify(struct P_Map *Mapping, struct P_Pair *Pair);
```

77.

```
int P_modify(struct P_Map *Mapping, struct P_Pair *Pair)
```

```
{
```

```
    char *gname;
```

```
    _CHECK_HANDLER;
```

```
    if (_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC) return P_EMODE;
```

```
    ⟨ group is valid or return an error 64 ⟩
```

```
    ⟨ Set new group number if Pair-group  $\equiv 0 \wedge \neg(\_HANDLER-pairs\_flags \& P\_FLAGS\_AUTOMATIC)$  65 ⟩
```

```
    ⟨ gname is valid or return an error 68 ⟩
```

```
    ⟨ Extend blocks if needed and allocate new block if not present 56 ⟩
```

```
    gname ← _P_GNAME(Pair-group);
```

```
    if (*gname ≠ '\0') /* already existing */
```

```
        _P_remove(Mapping, Pair-group);
```

```
    _P_COPY(Pair-group, Pair-gname);
```

```
    _P_insert(Mapping, Pair-group);
```

```
    return G_SUCCESS;
```

```
}
```

78. Insertion/suppression dans les tables de hachage.

Les routines qui suivent sont privées. Elles sont invoquées après les vérifications et ne peuvent pas échouer (sauf bourde de programmation).

Le préfixe ‘n’ vient du fait qu’elles manipulent essentiellement les tables de hachage, donc les noms. Mais, conformément à ce que nous avons précisé plus haut, *_P_nremove*, qui est invoquée temporairement pour retirer un nom (avant qu’il ne soit remplacé par un autre), met à jour à l’issue la non validité du nom dans les données propres (les blocs de paires), en remettant la chaîne à zéro.

_P_ninsert quant à elle met à jour *group_max* puisque le passage par les tables de support est un passage obligé. Nous mettons également à jour *group_max* quand nous étendons les listes, par sécurité, mais c’est insuffisant puisque cette portion de code n’est appelée que si le bloc auquel appartient le groupe n’est pas présent (c’est un numéro appartenant à ce bloc, ce n’est pas le plus grand numéro appartenant à ce bloc).

⟨ Private prototypes 78 ⟩ ≡

```
static void _P_ninsert(struct P_Map *Mapping, id_kt group);
```

See also sections 80 and 100.

This code is used in section 5.

79.

```
static void _P_ninsert(struct P_Map *Mapping, id_kt group)
{
    int h;
    struct _P_Name *p;
    char *gname;
    gname ← _P_GNAME(group);
    ⟨ Compute hash value of gname 62 ⟩
    p ← (struct _P_Name *) calloc(1, sizeof(struct _P_Name));
    p→group ← group;
    p→Next ← _HANDLER→hashes[h];
    if (_HANDLER→hashes[h] ≠ Λ) (_HANDLER→hashes[h]→Prev ← p;
    _HANDLER→hashes[h] ← p; /* update group_max if needed */
    if (group > _HANDLER→group_max) _HANDLER→group_max ← group;
}
```

80. Le retrait d’un nom des tables de hachage est temporaire (avant l’insertion de la nouvelle version).

⟨ Private prototypes 78 ⟩ +≡

```
static void _P_nremove(struct P_Map *Mapping, id_kt group);
```

81.

```

static void _P_remove(struct P_Map *Mapping, id_kt group)
{
    int h;
    char *gname;
    struct _P_Name *p;
    gname ← _P_GNAME(group);
    ⟨ Compute hash value of gname 62 ⟩
    if ((p ← _HANDLER→hashes[h]) ≠ Λ) {
        for (; p ≠ Λ; p ← p→Next) {
            if (p→group ≡ group) {
                if (p→Prev ≡ Λ) { /* head */
                    if ((p→Next) ≠ Λ) (p→Next)→Prev ← p;
                }
                else {
                    if ((p→Next) ≠ Λ) p→Next→Prev ← p;
                }
                *gname ← '\0';
                free(p);
                break;
            }
        }
    }
}

```

82. Ouvrir et fermer un fichier de mappage de paires.

Avant de pouvoir interroger la bibliothèque pour obtenir le nom associé à un groupe, ou avant de pouvoir insérer ou modifier une paire, il faut ouvrir un fichier de mappage des noms.

La bibliothèque autorise les mappages anonymes (éphémères), c'est-à-dire des mappages qui seront gérés en mémoire mais qui ne seront pas inscrits dans la GISDATABASE. Un mappage anonyme se crée en passant un pointeur *NULL* en guise de nom.

Si le mappage n'est pas anonyme, le comportement dépend du *mode* d'ouverture. En création, le fichier de mappage est écrit dans le mapset courant. En lecture ou lecture/écriture pour un fichier existant, la recherche est faite sur les mapsets accessibles définis dans *SEARCH_PATH*.

Le cas *G_O_RDWR* est particulier, car il peut s'agir, soit d'une création, soit d'une mise à jour. Or, dans le cas d'une création il n'y a évidemment rien à lire. Nous utilisons donc une variable pour indiquer s'il s'agit de créer ou pas.

```

⟨ Set mapset and creation flag 82 ⟩ ≡
    mapset ← Λ;
    if (mode ≡ G_O_WRONLY) creation ← 1;
    else creation ← 0;
    if (Mapping-name ≠ Λ ∧ mode ≠ G_O_WRONLY) {
        mapset ← G_find_file2(PART_ASCII_SUBELT, Mapping-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 85.

83. À l'invocation de *P_open*, une erreur différente de *G_SUCCESS* est renvoyée si problème et dans ce cas *handler* ≡ -1. Sinon *handler* est renseigné avec l'entier qui servira à identifier en interne le mappage.

Il est possible d'ouvrir un mappage anonyme (ou éphémère) en indiquant *Λ* comme valeur de *name*.

Dans ce cas, aucun fichier n'est ouvert, mais le mappage est créé en mémoire.

Après l'ouverture, *Mapping-Header.group_max* est renseignée avec la valeur effective, qui est *P_GROUP_EMPTY* dans le cas de la génération automatique.

```

⟨ Public prototypes 19 ⟩ +=
    int P_open(struct P_Map *Mapping, int oflag);

```

84. Puisque nous utilisons *yacc(1)*, nous avons besoin de partager au moins une information : celle sur le mappage actuellement utilisé. En fait, *yacc(1)* va renseigner *Mapping-Header* et invoquer les routines publiques (à commencer par *P_ioctl*) pour mettre à jour les données.

Afin d'éviter que cette variable, qui va être visible, serve à autre chose que ce pour quoi elle est conçue, il s'agit d'un pointeur sur le mappage courant qui est remis à *NULL* à l'issue.

```

⟨ Global variables 84 ⟩ ≡
    extern int _P_lineno;
    extern int _P_yyparse(void);
    extern FILE *_P_yyin;
    struct P_Map *_P_Cur_mapping;

```

See also sections 93 and 101.

This code is used in section 5.

85. Lorsque nous ouvrons un nouveau mappage, nous prenons soin, si nous sommes en écriture, de conserver le fichier ouvert (dans l'optique qu'un jour la GISLIB prennent en compte les verrous sur les fichiers partagés afin, si nous devons écrire, d'éviter qu'un autre processus tente de le faire également). Par contre, si nous sommes en lecture seule nous pourrions libérer les descripteurs. Nous les maintenons cependant, là aussi dans l'optique future de gérer les mises à jour dynamiques (nous avons une copie en mémoire qui pourrait être invalide).

Mais attention : si la gestion est prévue, elle n'est pas en place !

```

int P_open(struct P_Map *Mapping, int mode){ int status ← G_SUCCESS;    /* return value */
  char *mapset;    /* pointer to internal structure, don't free! */
  int creation;    /* G_O_RDWR case: if we create, nothing to parse */
  Mapping-handler ← -1;
  ⟨ Set mapset and creation flag 82 ⟩
  ⟨ Allocate and init new _P_Map; goto end if no handler left or problem 95 ⟩
  ⟨ Set default Header values 18 ⟩
  ⟨ Open mapping file if not anonymous 86 ⟩
  /* From now on, goto end on success or error */
  if (Mapping-name ≡ Λ ∨ creation) {    /* anonymous or new */
    ⟨ Init pairs management 87 ⟩
  }
  else {
    ⟨ Restart lexer 88 ⟩
    _P_Cur_mapping ← Mapping;
    if (_P_yyparse()) {
      status ← P_EBADFORMAT;
      _HANDLER_flags |= _P_FLAGS_DIRTY;
    }
    _P_Cur_mapping ← Λ;
    _P_yyin ← Λ;
  }
  end:
  if (status) (void) P_close(Mapping);
  else (void) P_ioctl(Mapping, P_IOCTL_GET_GROUP_MAX);
  return status; }

```

86. L'ouverture du fichier n'a rien d'exceptionnel. Tout ce qui concerne le `_HANDLER` sera évoqué dans la section consacrée à la gestion interne des fichiers.

On positionne `errno` à 0, car la GISLIB peut renvoyer des erreurs qui ne sont pas des erreurs système (et `errno` peut être distinct de zéro au moment de l'appel).

```

⟨ Open mapping file if not anonymous 86 ⟩ ≡
  if (Mapping-name ≠ Λ) {
    errno ← 0;
    if ((_HANDLER-fd ← G_open(PART_ASCII_SUBELT, Mapping-name, mapset, mode)) < 0) {
      G_msg(G_EOPEN, G_Errors, G_WARNING);
      status ← errno ? errno ≪ G_ERROR_SHIFT : P_EOPEN;
      _HANDLER-flags |= _P_FLAGS_DIRTY;
      goto end;
    }
    if ((_HANDLER-fp ← fdopen(_HANDLER-fd, G_fd_to_file_mode(mode))) ≡ Λ) {
      status ← errno ≪ G_ERROR_SHIFT;
      goto end;
    }
  }

```

This code is used in section 85.

87. Dans le cas anonyme ou pour une création, on doit effectuer l'initialisation ici (sinon elle est effectuée dans le parser).

À noter que l'ordre des initialisations est crucial. `P_GROUP_OTHER` doit être initialisé en dernier, car le comportement dépend des drapeaux.

```

⟨ Init pairs management 87 ⟩ ≡
  if ((status ← P_ioctl(Mapping, P_IOCTL_SET_FLAGS)) ≠ G_SUCCESS ∨ (status ← P_ioctl(Mapping,
    P_IOCTL_SET_GNAME_MAX_LEN)) ≠ G_SUCCESS ∨ (status ← P_ioctl(Mapping,
    P_IOCTL_SET_PREFIX)) ≠ G_SUCCESS) {
    _HANDLER-flags |= _P_FLAGS_DIRTY;
    goto end;
  }
  if (¬(_HANDLER-pairs-flags & P_FLAGS_AUTOMATIC) ∧ (status ← P_ioctl(Mapping,
    P_IOCTL_SET_GROUP_OTHER)) ≠ G_SUCCESS) {
    _HANDLER-flags |= _P_FLAGS_DIRTY;
    goto end;
  }

```

This code is used in section 85.

88. Par défaut, `lex(1)` reprend là où il s'est arrêté. Comme il retourne des valeurs, une partie des données sont statiques. Comme nous voulons pouvoir ouvrir plusieurs fichiers, il nous faut rétablir l'état initial du scanneur.

De facto, le code ici ne réalise pas cette ouverture multiple. C'est dans le code du scanneur lui-même (dans la fonction `_P_yyerror`) qu'une version est donnée pour `flex(1)`. Pour les autres, cela reste à faire (*a priori* le plus simple serait d'absorber le restant du texte dans le buffer après avoir fait pointer `_P_yyin` sur `/dev/null`).

```

⟨ Restart lexer 88 ⟩ ≡
  _P_lineno ← 1;
  _P_yyin ← _HANDLER-fp;

```

This code is used in section 85.

89. La fermeture du mappage via *P_close* entraînera, suivant le mode d'ouverture qui a été conservé, l'écriture ou pas des informations qui sont en mémoire.

```
<Public prototypes 19> +=  
  int P_close(struct P_Map *Mapping);
```

90.

```
int P_close(struct P_Map *Mapping)  
{  
  int status ← G_SUCCESS;    /* be able to be safely called when not opened (interrupted user program) */  
  if (Mapping-handler ≡ -1) return G_SUCCESS;  
  _CHECK_HANDLER;  
  
  <Write mapping information if requested 91>  
end: <Free _HANDLER entry and set Mapping-handler ← -1 98>  
  return status;  
}
```

91. L'écriture des informations est triviale et respecte (espérons-le !) le format défini.

```

< Write mapping information if requested 91 > ≡
#ifdef DEBUG
    if (_HANDLER-fp ≡ Λ) _HANDLER-fp ← stderr;
#endif
if (¬(_HANDLER-flags & _P_FLAGS_DIRTY) ∧ _HANDLER-fp ≠ Λ ∧ _HANDLER-mode ≠ G_O_RDONLY) {
    id_kt group;
    char *gname;
    if (_HANDLER-fp ≠ stderr ∧ _HANDLER-fp ≠ stdout ∧ _HANDLER-mode ≠ G_O_WRONLY) {
        if (fruncate(_HANDLER-fd, (off_t)0) ≡ -1) {
            G_msg(errno << G_ERROR_SHIFT, G_Errors, G_WARNING);
            goto end;
        }
        if (fseek(_HANDLER-fp, 0_L, SEEK_SET)) {
            G_msg(errno << G_ERROR_SHIFT, G_Errors, G_WARNING);
            goto end;
        }
    }
    fprintf(_HANDLER-fp, "VERSION: _%s\n", P_VERSION);
    if (Mapping-Header.description ≠ Λ)
        fprintf(_HANDLER-fp, "DESCRIPTION: _\"%s\"\n", Mapping-Header.description);
    *_HANDLER-prefix_end ← '\0';
    fprintf(_HANDLER-fp, "GNAME_MAX_LEN: _%lu\n", (unsigned long) _HANDLER-gname_max_len);
    fprintf(_HANDLER-fp, "PREFIX: _\"%s\"\n", _HANDLER-buffer);
    if (¬(_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC))
        fprintf(_HANDLER-fp, "GROUP_OTHER: _\"%s\"\n", _P_GNAME(0));
    fprintf(_HANDLER-fp, "FLAGS: _");
    if (_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC) fprintf(_HANDLER-fp, "P_FLAGS_AUTOMATIC\n");
    else fprintf(_HANDLER-fp, "%lu\n", _HANDLER-pairs_flags);
    (void) P_ioctl(Mapping, P_IOCTL_GET_GROUP_MAX);
    fprintf(_HANDLER-fp, "GROUP_MAX: _%lu\n", Mapping-Header.group_max);
    fprintf(_HANDLER-fp, "PAIRS: \n");
    if (¬(_HANDLER-pairs_flags & P_FLAGS_AUTOMATIC)) {
        for (group ← 1; group ≤ _HANDLER-group_max; group++) {
            gname ← _P_GNAME(group);
            if (*gname ≠ '\0') fprintf(_HANDLER-fp, "%lu, \"%s\"\n", (unsigned long) group, gname);
        }
    }
    fflush(_HANDLER-fp);
}

```

This code is used in section 90.

92. Gestion interne des fichiers.

À côté des données, il y a la gestion des fichiers. On doit au minimum conserver le mode d'ouverture (pour la fermeture...).

```

< Private structures 60 > +≡
struct _P_Map {
    char *name;
    FILE *fp;
    int fd;
    unsigned long flags;
    int mode;
    < Pairs management 28 >
};

```

93. Et comme l'on veut pouvoir manipuler plusieurs fichiers de mappage, la bibliothèque gère une liste de structures, le *handler* étant simplement un petit entier représentant l'index dans cette liste.

```

< Global variables 84 > +≡
#define _MAX_HANDLERS 16 /* minimum value of OPEN_MAX */
    static int _P_nb_handlers; /* nb of handlers actually in use */
    static struct _P_Map *_P_handlers[_MAX_HANDLERS];
#define _HANDLER (_P_handlers[Mapping-handler])

```

94. La manipulation des structures de gestion requiert tout le temps la vérification que les données passées ont un sens. On vérifie déjà, au minimum, le *handler*.

```

< Private macros 24 > +≡
#define _CHECK_HANDLER if (Mapping-handler < 0 ∨ Mapping-handler >
    _MAX_HANDLERS ∨ _P_handlers[Mapping-handler] ≡ Λ) return P_EBADHANDLER

```

95. S'il n'y a pas de handler disponible, on peut arrêter.

```

< Allocate and init new _P_Map; goto end if no handler left or problem 95 > ≡
if (_P_nb_handlers ≥ _MAX_HANDLERS) return P_EOPENMAX;
for (Mapping-handler ← 0; Mapping-handler < _MAX_HANDLERS ∧ _P_handlers[Mapping-handler] ≠ Λ;
    ++Mapping-handler) ; /* calloc ensure default values */
if ((_P_handlers[Mapping-handler] ← (struct _P_Map *) calloc(1, sizeof(struct _P_Map))) ≡ Λ) {
    Mapping-handler ← -1;
    return errno << G_ERROR_SHIFT;
}
++_P_nb_handlers;
_HANDLER-mode ← mode;
_HANDLER-fd ← -1;

```

This code is used in section 85.

96. Si un incident se produit dans la gestion en mémoire du mappage, il ne s'agit pas, même si cela était demandé, d'écraser une version correcte avec une version qui ne le serait pas. En conséquence, un drapeau est levé si les données sont sales **_P_FLAGS_DIRTY** et aucune écriture ne doit avoir lieu.

```

< Private macros 24 > +≡
#define _P_FLAGS_DIRTY °I

```

97. À la fermeture, il nous faut libérer les tables de hachages, les blocs, les descripteurs, avant de libérer la structure en elle-même.

```

< Include files 25 > +≡
#include <unistd.h>

```

98.

```

⟨Free _HANDLER entry and set Mapping-handler ← -1 98⟩ ≡
{
  id_kt i;
  for (i ← 0; i < HASH_SIZE; i++) {
    struct _P_Name *p, *q;
    for (p ← _HANDLER-hashes[i]; p ≠ Λ; p ← q) {
      q ← p-Next;
      free(p);
    }
  }
  if (_HANDLER-blocks ≠ Λ) {
    for (i ← 0; i ≤ _P_BLOCK_NB(_HANDLER-group-max); i++) free(_HANDLER-blocks[i]);
    free(_HANDLER-blocks);
  }
  if (_HANDLER-fd ≠ -1) (void) close(_HANDLER-fd);
  if (_HANDLER-fp ≠ Λ ∧ _HANDLER-fp ≠ stderr ∧ _HANDLER-fp ≠ stdout) (void) fclose(_HANDLER-fp);
  free(_P_handlers[Mapping-handler]);
  _P_handlers[Mapping-handler] ← Λ;
  _P_nb_handlers--;
  Mapping-handler ← -1;
}

```

This code is used in section 90.

99. Les messages d'erreur.

```

< Errors macros 99 > ≡
#define P_EGROUPNOTFOUND 64
#define P_EGNAMENOTFOUND 65
#define P_EEXIST 66
#define P_EBADGROUP 67
#define P_EBADGNAME 68
#define P_EBADFORMAT 69
#define P_EBADHANDLER 70
#define P_EOPENMAX 71
#define P_EOPEN 72
#define P_EIOCTLBADREQUEST 73
#define P_EIOCTLUNKNOWNREQUEST 74
#define P_EIOCTLLENTOOSHORT 75
#define P_EMAXNBGROUPS 76
#define P_EPREFIXTOOLONG 77
#define P_EPARSER 78
#define P_EMODE 79
#define P_EMAX 79

```

This code is used in section 4.

100. La structure `_P_Errors` est placée à part de manière à permettre l'utilisation de la fonctionnalité du *'change file'* de CWEB pour produire une version localisée. Un schéma plus général, au niveau de KerGIS, sera fourni plus tard.

```

< Private prototypes 78 > +≡
static char *_P_estrerror(int msgnum);

```

101.

```

< Global variables 84 > +≡
static struct G_Msgs _P_Errors ← {"libpart", P_EMAX, /* last error so max */
_P_estrerror};
const struct G_Msgs *const P_Errors ← &_P_Errors;
< C lang errors formats 103 >

```

102.

```

static char *_P_estrerror(int msgnum)
{
return errors[msgnum - G_ERROR_MAX - 1];
}

```

103. Les messages d'erreurs sont mis à part.

```

< C lang errors formats 103 > ≡ /* must be kept in sync with the errors macros! */
static char *errors[] ← {"group_not_found\n", "gname_not_found\n",
"group_name_already_in_use\n", "incorrect_group_number\n",
"incorrect_gname_(whether_NULL_or_empty)\n",
"incorrect_version_(supported_is_P_VERSION)_or_bad_format\n",
"the_handler_id_passed_is_incorrect\n", "too_many_files_already_handled\n",
"unable_to_open_mapping_file_in_requested_mode\n", "bad_ioctl_request\n",
"unknown_ioctl_request\n", "gname_max_len_is_too_short\n",
"maximum_number_of_groups_reached\n", "prefix_too_long_for_max_gname_length\n",
"parse_error,_line_%d\n%s\n%s\n",
"operation_not_allowed_in_present_mode_(flags)\n", };

```

This code is used in section 101.

104. Programme d'exemple et de test.

Ce programme donne une vue d'ensemble des routines et permet d'explorer ou de créer un mappage. Il peut donc servir également de programme de test.

```

<test_part.c 104> ≡
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include "gis.h"
#include "part.h"
static void get_a_num(void);
static void get_a_string(void);
static void get_a_pair(void);
struct P_Map Mapping; /* lazy to get all zeroed */
struct P_Pair Pair; /* idem */
/*_ARGSUSED_*/ int main(int argc, char **argv){ int status;
    char name[80];
    int quit; /* to flag quit choice */
    int choice;

    G_gisinit(argv[0]);
    name[79] ← '\0';
    quit ← 0;
    while (¬quit) {
        (void) fprintf(stderr,
            "Enter_choice:\n" "1 existing_partition\n" "2 new_anonymous_partition\n");
        if (scanf("%d", &choice) < 1) {
            fprintf(stderr, "ERROR: you must give a number!\n");
            continue;
        }
        switch (choice) {
        case 1: (void) printf("Give the name of a partition file: ");
            (void) scanf("%79s", name);
            Mapping.name ← name;
            if ((status ← P_open(&Mapping, G_O_RDONLY)) ≠ G_SUCCESS)
                G_msg(status, P_Errors, G_INFO);
            else quit ← 1;
            break;
        case 2: Mapping.name ← Λ;
            fprintf(stderr, "Enter_gname_max_len: ");
            get_a_num();
            Mapping.Header.gname_max_len ← (size_t) Pair.group;
            fprintf(stderr, "Enter_a_description: ");
            get_a_string();
            Mapping.Header.description ← G_store(Pair.gname);
            fprintf(stderr, "Enter_a_prefix: ");
            get_a_string();
            Mapping.Header.prefix ← G_store(Pair.gname);
            fprintf(stderr, "Enter_group_other_name: ");
            get_a_string();
            Mapping.Header.group_other ← G_store(Pair.gname);
            fprintf(stderr, "Enter_flags_(0_or_1_for_auto): ");
            get_a_num();
            Mapping.Header.flags ← Pair.group;

```

```

    if ((status ← P_open(&Mapping, G_O_WRONLY)) ≠ G_SUCCESS)
        G_msg(status, P_Errors, G_INFO);
    else quit ← 1;
    break;
default: break;
}
}
quit ← 0;
while (1) {
    (void) fprintf(stderr, "Enter choice:\n"
        "1 search_by_group\n"
        "2 search_by_gname\n"
        "3 insert/modify_group/gname\n"
        "4 automatic_group_a\
        nd_gname\n"
        "5 change_group_other\n"
        "6 change_prefix\n"
        "7 quit\n");
    if (scanf("%d", &choice) < 1) {
        fprintf(stderr, "ERROR: you must give a number!\n");
        continue;
    }
    switch (choice) {
case 1: get_a_num();
        if ((status ← P_find_by_group(&Mapping, &Pair))) G_msg(status, P_Errors, G_INFO);
        else (void) fprintf(stderr, "Group %ld, %s\n", (long) Pair.group, Pair.gname);
        break;
case 2: get_a_string();
        Pair.group ← 0;
        if ((status ← P_find_by_gname(&Mapping, &Pair))) G_msg(status, P_Errors, G_INFO);
        else (void) fprintf(stderr, "Group %ld, %s\n", (long) Pair.group, Pair.gname);
        break;
case 3: get_a_pair();
        if ((status ← P_modify(&Mapping, &Pair)) ≠ G_SUCCESS) G_msg(status, P_Errors, G_INFO);
        else {
            (void) fprintf(stderr, "Modification succeeded:");
            (void) fprintf(stderr, "Group %ld, %s\n", (long) Pair.group, Pair.gname);
        }
        break;
case 4: free(Pair.gname);
        Pair.gname ← Λ;
        Pair.group ← 0;
        if ((status ← P_modify(&Mapping, &Pair)) ≠ G_SUCCESS) G_msg(status, P_Errors, G_INFO);
        else (void) fprintf(stderr, "New: %ld, %s\n", (long) Pair.group, Pair.gname);
        break;
case 5: get_a_string();
        free(Mapping.Header.group_other);
        Mapping.Header.group_other ← Pair.gname;
        if ((status ← P_ioctl(&Mapping, P_IOCTL_SET_GROUP_OTHER)) ≠ G_SUCCESS)
            G_msg(status, P_Errors, G_INFO);
        else (void) fprintf(stderr, "Modification succeeded\n");
        break;
case 6: get_a_string();
        free(Mapping.Header.prefix);
        Mapping.Header.prefix ← Pair.gname;
        if ((status ← P_ioctl(&Mapping, P_IOCTL_SET_PREFIX)) ≠ G_SUCCESS)
            G_msg(status, P_Errors, G_INFO);
        else (void) fprintf(stderr, "Modification succeeded\n");
    }
}

```

```

        break;
    case 7: quit ← 1;
        break;
    default: break;
    }
    if (quit) break;
}
if ((status ← P_close(&Mapping))) goto error ;
(void) fprintf(stderr, "Successfully_run\n");
exit(EXIT_SUCCESS); error : G_msg(status, P_Errors, G_FATAL);
/*_NOTREACHED_*/ exit(EXIT_FAILURE); } static void get_a_string(void)
{
    int quit;
    if ((Pair.gname ← (char *) realloc(Pair.gname, 1024)) ≡ Λ)
        G_msg(errno ≪ G_ERROR_SHIFT, G_Errors, G_FATAL);
    Pair.gname[1023] ← '\0';
    quit ← 0;
    while (¬quit) {
        (void) fprintf(stderr, "Group_name/string_(in_\\"");
        if (scanf("\\""%1023[^\n]\\"", Pair.gname) ≡ 1) quit ← 1;
    }
}
static void get_a_pair(void)
{
    long num;
    int quit;
    if ((Pair.gname ← (char *) realloc(Pair.gname, 1024)) ≡ Λ)
        G_msg(errno ≪ G_ERROR_SHIFT, G_Errors, G_FATAL);
    Pair.gname[1023] ← '\0';
    quit ← 0;
    while (¬quit) {
        (void) fprintf(stderr,
            "Group/gname_to_insert/modify_\\"(space_separated, _gname_in_\\"");
        if (scanf("%ld, \\""%1023[^\n]\\"", &num, Pair.gname) ≡ 2) quit ← 1;
    }
    Pair.group ← (id_kt) num;
}
static void get_a_num(void)
{
    long num;
    int quit;
    quit ← 0;
    while (¬quit) {
        (void) fprintf(stderr, "Group/number: \");
        if (scanf("%ld", &num) ≡ 1) quit ← 1;
    }
    Pair.group ← (id_kt) num;
    free(Pair.gname); /* gname shall be whether Λ or valid */
    Pair.gname ← Λ;
}

```

105. Index.

Les entrées soulignées indiquent la place de la définition.

- _Pair*: 68.
- _CHECK_HANDLER*: 21, 71, 73, 77, 90, 94.
- _HANDLER*: 30, 36, 43, 44, 48, 51, 53, 54, 56, 57, 65, 68, 69, 72, 73, 74, 75, 77, 79, 81, 85, 86, 87, 88, 91, 93, 95, 98.
- _MAX_HANDLERS*: 93, 94, 95.
- _P_BLOCK_NB*: 54, 56, 69, 98.
- _P_BLOCK_SIZE*: 48, 53, 54, 56.
- _P_COPY*: 48, 57, 77.
- _P_Cur_mapping*: 84, 85.
- _P_Errors*: 100, 101.
- _P_esterror*: 100, 101, 102.
- _P_FLAGS_DIRTY*: 85, 86, 87, 91, 96.
- _P_GNAME*: 48, 54, 57, 69, 74, 77, 79, 81, 91.
- _P_handlers*: 93, 94, 95, 98.
- _P_lineno*: 84, 88.
- _P_Map***: 92, 93, 95.
- _P_MAX_DEC_LENGTH*: 24, 27, 30, 43.
- _P_Name***: 60, 61, 73, 79, 81, 98.
- _P_nb_handlers*: 93, 95, 98.
- _P_ninsert*: 48, 77, 78, 79.
- _P_nremove*: 48, 77, 78, 80, 81.
- _P_yyerror*: 88.
- _P_yyin*: 84, 85, 88.
- _P_yyparse*: 84, 85.
- _Pair*: 48.
- argc*: 104.
- argv*: 104.
- ASCII_SUBELT: 14.
- BASE: 3.
- blocks*: 48, 53, 54, 56, 69, 98.
- buffer*: 40, 41, 43, 48, 68, 69, 75, 91.
- c*: 75.
- calloc*: 48, 56, 79, 95.
- CHAR_BIT: 24.
- choice*: 104.
- close*: 98.
- complément*: 7.
- creation*: 82, 85.
- DEBUG: 91.
- description*: 22, 23, 91, 104.
- end*: 85, 86, 87, 90, 91.
- errno*: 43, 48, 56, 86, 91, 95, 104.
- errors*: 102, 103.
- exit*: 104.
- EXIT_FAILURE: 104.
- EXIT_SUCCESS: 104.
- fclose*: 98.
- fd*: 86, 91, 92, 95, 98.
- fdopen*: 86.
- fflush*: 91.
- flags*: 32, 36, 85, 86, 87, 91, 92, 104.
- floor*: 24.
- fp*: 86, 88, 91, 92, 98.
- fprintf*: 91, 104.
- free*: 43, 81, 98, 104.
- fseek*: 91.
- ftruncate*: 91.
- G_open*: 86.
- G_ENOENT: 82.
- G_EOPEN: 86.
- G_ERROR_MAX: 102.
- G_ERROR_SHIFT: 43, 48, 56, 86, 91, 95, 104.
- G_Errors*: 86, 91, 104.
- G_FATAL: 104.
- G_fd_to_file_mode*: 86.
- G_find_file2*: 82.
- G_gisinit*: 104.
- G_INFO: 104.
- G_mapset*: 82.
- G_msg*: 86, 91, 104.
- G_Msgs*: 4, 101.
- G_O_RDONLY: 15, 82, 91, 104.
- G_O_RDWR: 15, 82, 85.
- G_O_WRONLY: 15, 82, 91, 104.
- G_store*: 23, 39, 46, 48, 68, 71, 104.
- G_SUCCESS: 30, 36, 43, 48, 51, 68, 71, 74, 75, 77, 83, 85, 87, 90, 104.
- G_WARNING: 86, 91.
- get_a_num*: 104.
- get_a_pair*: 104.
- get_a_string*: 104.
- gid*: 7.
- GISDATA: 3.
- GISDATABASE: 3, 82.
- gname*: 2, 13, 48, 57, 62, 68, 69, 70, 71, 73, 74, 75, 77, 79, 81, 91, 104.
- gname_max_len*: 26, 27, 28, 30, 43, 48, 53, 54, 56, 57, 73, 91, 104.
- group*: 2, 13, 44, 48, 54, 56, 57, 60, 64, 65, 69, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 91, 104.
- group_max*: 49, 51, 55, 56, 65, 69, 78, 79, 83, 91, 98.
- group_other*: 45, 46, 48, 104.
- h*: 73, 79, 81.
- handler*: 15, 83, 85, 90, 93, 94, 95, 98.
- HASH_SIZE: 61, 62, 98.
- hashes*: 61, 72, 74, 79, 81, 98.
- Header*: 15, 18, 19, 23, 27, 30, 36, 39, 43, 46, 48, 49, 51, 83, 84, 91, 104.
- i*: 56, 62, 75, 98.

- id_kt**: 4, 6, 10, 13, 24, 49, 55, 60, 75, 78, 79, 80, 81, 91, 98, 104.
- initialized*: 30, 31, 36, 48.
- KERGIS_PART_H: 4.
- len*: 43.
- LOCATION: 3.
- M_LN10: 24.
- M_LN2: 24.
- main*: 104.
- malloc*: 43.
- Mapping*: 18, 19, 21, 22, 23, 25, 27, 30, 36, 39, 43, 46, 48, 51, 68, 70, 71, 72, 73, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 89, 90, 91, 93, 94, 95, 98, 104.
- MAPSET: 3.
- mapset*: 82, 85, 86.
- mode*: 82, 85, 86, 91, 92, 95.
- msgnum*: 100, 102.
- name*: 15, 82, 83, 85, 86, 92, 104.
- Next*: 60, 74, 79, 81, 98.
- NO_DATA: 45.
- num*: 104.
- off_t*: 91.
- oflag*: 83.
- OPEN_MAX: 93.
- p*: 56, 73, 79, 81, 98.
- P_close*: 85, 89, 90, 104.
- P_EBADFORMAT: 85, 99.
- P_EBADGNAME: 48, 68, 73, 99.
- P_EBADGROUP: 64, 99.
- P_EBADHANDLER: 94, 99.
- P_EEXIST: 48, 68, 99.
- P_EGNAMENOTFOUND: 73, 75, 99.
- P_EGROUPNOTFOUND: 71, 99.
- P_EIOCTLBADREQUEST: 30, 36, 43, 48, 99.
- P_EIOCTLENTOOSHORT: 30, 99.
- P_EIOCTLUNKNOWNREQUEST: 21, 99.
- P_EMAX: 99, 101.
- P_EMAXNBGROUPS: 65, 99.
- P_EMODE: 77, 99.
- P_EOPEN: 86, 99.
- P_EOPENMAX: 95, 99.
- P_EPARSER: 99.
- P_EPREFIXTOOLONG: 43, 99.
- P_Errors*: 4, 101, 104.
- P_find_by_gname*: 48, 68, 72, 73, 104.
- P_find_by_group*: 70, 71, 104.
- P_FLAGS_AUTOMATIC: 34, 45, 48, 51, 65, 69, 73, 77, 87, 91.
- P_GCOMPLEMENT: 7, 9, 11.
- P_GROUP_ALL: 11.
- P_GROUP_DEFINED: 9.
- P_GROUP_EMPTY: 10, 11, 12, 49, 51, 65, 72, 73, 83.
- P_GROUP_OTHER: 8, 9, 12, 37, 45, 56, 66, 87.
- P_Header**: 14, 15.
- P_ioctl*: 19, 21, 56, 84, 85, 87, 91, 104.
- P_IOCTL_GET_GROUP_MAX: 50, 51, 85, 91.
- P_IOCTL_SET_FLAGS: 35, 36, 87.
- P_IOCTL_SET_GNAME_MAX_LEN: 29, 30, 87.
- P_IOCTL_SET_GROUP_OTHER: 47, 48, 87, 104.
- P_IOCTL_SET_PREFIX: 42, 43, 87, 104.
- P_Map**: 15, 19, 21, 70, 71, 72, 73, 76, 77, 78, 79, 80, 81, 83, 84, 85, 89, 90, 104.
- P_modify*: 66, 76, 77, 104.
- P_open*: 15, 83, 85, 104.
- P_Pair**: 13, 48, 68, 70, 71, 72, 73, 76, 77, 104.
- P_VERSION: 16, 18, 91, 103.
- Pair*: 44, 48, 56, 64, 65, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 104.
- Pair-p*: 68.
- pairs_flags*: 33, 36, 48, 51, 65, 69, 73, 77, 87, 91.
- PART_: 14.
- PART_ASCII_SUBELT: 3, 82, 86.
- PART_MAJOR: 4.
- PART_MINOR: 4.
- PART_REVISION: 4.
- partition*: 3.
- prefix*: 30, 38, 39, 43, 104.
- prefix_end*: 40, 41, 43, 44, 75, 91.
- Prev*: 60, 79, 81.
- printf*: 104.
- q*: 98.
- quit*: 104.
- realloc*: 48, 56, 104.
- request*: 19, 21.
- scanf*: 104.
- SEARCH_PATH: 82.
- SEEK_SET: 91.
- sprintf*: 44.
- sscanf*: 75.
- status*: 85, 86, 87, 90, 104.
- stderr*: 91, 98, 104.
- stdout*: 91, 98.
- strcmp*: 74.
- strcpy*: 43.
- strlen*: 30, 43, 48, 73.
- strncmp*: 75.
- strncpy*: 57.
- support*: 3.
- V_GTYPE_AREA: 6.
- V_GTYPE_LINE: 6.
- V_GTYPE_POINT: 6.
- V_TTYPE_DOT: 6.
- V_TTYPE_EDGE: 6.
- V_TTYPE_PATH: 6.

VECTOR_GEOM_SUBELT: 3.

VECTOR_PART_SUBELT: 3.

version: 17, 18.

- ⟨ Allocate and init new **_P_Map; goto end** if no handler left or problem 95 ⟩ Used in section 85.
- ⟨ C lang errors formats 103 ⟩ Used in section 101.
- ⟨ Compare gname against hashes and return if found 74 ⟩ Used in section 73.
- ⟨ Compare gname with automatic name and return 75 ⟩ Used in section 73.
- ⟨ Compute hash value of *gname* 62 ⟩ Used in sections 74, 79, and 81.
- ⟨ Errors macros 99 ⟩ Used in section 4.
- ⟨ Extend *blocks* if needed and allocate new block if not present 56 ⟩ Used in section 77.
- ⟨ Free **_HANDLER** entry and set *Mapping-handler* ← -1 98 ⟩ Used in section 90.
- ⟨ Get *gname* associated with *Pair-group* 69 ⟩ Used in section 71.
- ⟨ Global variables 84, 93, 101 ⟩ Used in section 5.
- ⟨ Header members 17, 22, 26, 32, 38, 45, 49 ⟩ Used in section 14.
- ⟨ Include files 25, 97 ⟩ Used in section 5.
- ⟨ Init pairs management 87 ⟩ Used in section 85.
- ⟨ Open mapping file if not anonymous 86 ⟩ Used in section 85.
- ⟨ Pairs management 28, 31, 33, 41, 53, 55, 61 ⟩ Used in section 92.
- ⟨ Private macros 24, 54, 57, 94, 96 ⟩ Used in section 5.
- ⟨ Private prototypes 78, 80, 100 ⟩ Used in section 5.
- ⟨ Private structures 60, 92 ⟩ Used in section 5.
- ⟨ Public macros 7, 8, 9, 10, 11, 16, 20, 34 ⟩ Used in section 4.
- ⟨ Public prototypes 19, 70, 72, 76, 83, 89 ⟩ Used in section 4.
- ⟨ Public structures 13, 14, 15 ⟩ Used in section 4.
- ⟨ Restart lexer 88 ⟩ Used in section 85.
- ⟨ Set default Header values 18, 23, 27, 39, 46 ⟩ Used in section 85.
- ⟨ Set default gname in *_HANDLER-buffer* 44 ⟩ Used in sections 48, 68, and 69.
- ⟨ Set mapset and creation flag 82 ⟩ Used in section 85.
- ⟨ Set new group number if *Pair-group* $\equiv 0 \wedge \neg(\text{_HANDLER-pairs_flags} \& \text{P_FLAGS_AUTOMATIC})$ 65 ⟩ Used in section 77.
- ⟨ Write mapping information if requested 91 ⟩ Used in section 90.
- ⟨ *gname* is valid or return an error 68 ⟩ Used in section 77.
- ⟨ *group* is valid or return an error 64 ⟩ Used in sections 71 and 77.
- ⟨ *ioctl* macros 29, 35, 42, 47, 50 ⟩ Used in section 20.
- ⟨ *ioctl* requests 30, 36, 43, 48, 51 ⟩ Used in section 21.
- ⟨ *part.h* 4 ⟩
- ⟨ *test_part.c* 104 ⟩