The stdin Correspondence back-end

(Work in progress—handle with care)
$Id: libstdin.w,v 1.8 2005/09/12 18:09:53 tlaronde Exp

Thierry LARONDE

Abstract

The ability to take correspondence specification via stdin can be handy. Hence this driver.

La possibilité de prendre la description de la correspondence et des attributs depuis l'entrée standard peut être utile. C'est l'objet de ce soutier.

November 26, 2006 at 11:44

1. Licence.

2. 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` ≡ 0 for OK, or the error number if there was one.

```
⟨ corr/stdin.h 2 ⟩ ≡
#ifndef STDIN_H
#define STDIN_H
#include <string.h>
#include "gis.h"
#include "corr.h"
  ⟨ Public prototypes 3 ⟩
#endif    /* STDIN_H */
```

3. When dealing with stdin, opening is quite simple. There is indeed nothing to really do, except verifying that one does not try to invoke us twice.

```
⟨ Public prototypes 3 ⟩ ≡
  int C_stdin_open(struct C_DFile *Cstdin);
```

See also sections 4, 5, and 6.

This code is used in section 2.

4. For closing, this simply releases the handler.

```
⟨ Public prototypes 3 ⟩ +≡
  int C_stdin_close(struct C_DFile *Cstdin);
```

5. Reading att after att.

```
⟨ Public prototypes 3 ⟩ +≡
  int C_stdin_get_image(struct C_DFile *Cstdin, struct C_Image *Image);
```

6. Writing is non sense.

```
⟨ Public prototypes 3 ⟩ +≡
  int C_stdin_set_image(struct C_DFile *Cstdin, struct C_Image *Image);
```

7. Implementation.

When dealing with stdin, opening is quite simple. The main caveat is that the *Corr.Data* must have been filled since this back-end does not retrieve the description of the data from the values source (stdin) (`psql` and `dbf` have a description of the data structure with the values, this is not the case here).

```
⟨ stdin/libstdin.c 7 ⟩ ≡
#include "corr/stdin.h"
  static int _Cstdin_handler;      /* 0 initially */
#define _CHECK_HANDLER if (Cstdin→handler ≠ 1) return C_EBADHANDLER
  int C_stdin_open(struct C_DFile *Cstdin)
  {
    int status ← G_SUCCESS;

    Cstdin→handler ← −1;
    if (Cstdin→mode ≠ G_O_RDONLY) {
      status ← C_EMODE;
      goto end;
    }
    if (Cstdin→Data ≡ Λ ∨ Cstdin→Data→nb_datums ≡ 0 ∨ Cstdin→Data→nb_atts ≡ 0 ∨ Cstdin→Data→Datums ≡ Λ) {
      status ← C_stdin_EBADDATA;
      goto end;
    }
    ⟨ Set defaults 8 ⟩
    if (¬_Cstdin_handler) {
      _Cstdin_handler ← 1;
      Cstdin→handler ← _Cstdin_handler;
    }
    else status ← C_EOPENMAX;
  end: return status;
  }
```

See also sections 9, 10, and 13.

8. We are setting some defaults (the same are used for `stdout`).

```
⟨ Set defaults 8 ⟩ ≡
  if (Cstdin→Head→separator ≡ '\0') Cstdin→Head→separator ← '|';
  if (Cstdin→Head→null_as ≡ Λ) Cstdin→Head→null_as ← "NULL";
  if (Cstdin→Head→encoding ≡ Λ) Cstdin→Head→encoding ← KT_ICONV_UTF_8;
```

This code is used in section 7.

9. On close we simply release and reset values.

```
⟨ stdin/libstdin.c 7 ⟩ +≡
  int C_stdin_close(struct C_DFile *Cstdin)
  {
    int status ← G_SUCCESS;

    _CHECK_HANDLER;
    _Cstdin_handler ← 0;
    Cstdin→handler ← −1;
    return status;
  }
```

10. So we will get values from stdin. The values will be separated by *Corr.separator*, and on getting we need the *Corr.Data* filled since the description of the data is separated from the values (this is not the case with `psql` or `dbf`) : so some test has to be achieved.

Note that this version does not make extensive checking of the consistency of the input (this has to be done sooner or later).

There is no selection for stdin: it shall only be used to store values in another format.

```
⟨ stdin/libstdin.c 7 ⟩ +≡
  int C_stdin_get_image(struct C_DFile *Cstdin, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    unsigned long nb_atts_here;
    struct C_Att *Att;
    char *buf;    /* set by G_fread_line */
    _CHECK_HANDLER;
    if (Image→group ≠ P_GROUP_ALL) {
      status ← C_stdin_EBADGROUP;
      goto end;
    }
    Image→multiplicity ← Cstdin→Data→nb_atts;    /* overwritting: shall be fixed */
    if (Image→multiplicity > Image→offset) {    /* something to do */
      nb_atts_here ← (Image→multiplicity − Image→offset > Cstdin→Head→chunk_size) ? Cstdin→Head→chunk_size :
          Image→multiplicity − Image→offset;
      if ((status ← C_alloc_atts(Cstdin→Data→nb_datums, Image, nb_atts_here))) goto end;
      for (i ← 0; i < Image→nb_atts_here; i++) {    /* Image→nb_atts_here set by C_alloc_atts */
        Att ← &Image→Atts[i];
        ⟨ Set the fields from line of input; goto end on error 11 ⟩
      }
    }
  end: return status;
  }
```

11. Getting the values from a line of input such as to have the *oid* and all the columns is simple if the values are separated by a well known separator.

```
⟨ Set the fields from line of input; goto end on error 11 ⟩ ≡
  {
    G_fread_line(&buf, stdin);
    ⟨ Copy each string between SEPARATOR in the corresponding field 12 ⟩
    free(buf);    /* was set by G_fread_line */
    buf ← Λ;
    if (status) goto end;    /* we may have break in loop */
  }
```

This code is used in section 10.

12. The values are printed separated by SEPARATOR. So we use two pointers, one after the previous separator, one before the next one, to copy the string between the two. We need to copy since the buffer will be freed.

There are *nb_datums* $+ 2$ fields, with *aid* and *flags*.

```
⟨ Copy each string between SEPARATOR in the corresponding field 12 ⟩ ≡
  {
    int j;
    char *bp, *ep;    /* our two pointers, begin and end */
    ep ← buf;    /* begin at the beginning */
    for (j ← 0; j ≤ Cstdin→Data→nb_datums + 1; j++) {
      bp ← ep;
      for ( ; *ep ≠ Cstdin→Head→separator ∧ *ep; ep++) ;
      *ep ← '\0';    /* truncate this portion */
      if (j ≡ 0)    /* oid */
        (void) sscanf(bp, "%ld", &Att→id);
      else {
        if (Cstdin→Data→Datums[j − 1].Logical.type)    /* not C_RAW_L */
          G_strip(bp);    /* remove leading and trailing formatting blanks */
        if (j ≡ 1) {    /* flags */
          if (¬strcmp(bp, "DELETED")) Att→flags |= C_ATT_DELETED;
          if (¬strcmp(bp, "MODIFIED")) Att→flags |= C_ATT_MODIFIED;
        }
        else {
          if (Cstdin→Head→null_as ≠ Λ ∧ (¬strcmp(bp, Cstdin→Head→null_as))) Att→fields[j − 2] ← Λ;
              /* redondant */
          else Att→fields[j − 2] ← bp ≡ Λ ? Λ : G_store(bp);
        }
      }
      ep++;    /* skip Λ for loop, harmless at end */
    }
  }
```

This code is used in section 11.

13. Other functions are non sense.

```
⟨ stdin/libstdin.c 7 ⟩ +≡
  /*␣ARGSUSED␣*/ int C_stdin_set_image(struct C_DFile *Cstdin, struct C_Image *Image)
    {
      return C_EMODE;
    }
```

14. Index.

Here is a list of the identifiers used. Striked entries are the place of definition.

⟨ Copy each string between `SEPARATOR` in the corresponding field 12 ⟩ Used in section 11.
⟨ Public prototypes 3, 4, 5, 6 ⟩ Used in section 2.
⟨ Set defaults 8 ⟩ Used in section 7.
⟨ Set the fields from line of input; **goto** *end* on error 11 ⟩ Used in section 10.
⟨ `corr/stdin.h` 2 ⟩
⟨ `stdin/libstdin.c` 7, 9, 10, 13 ⟩