The stdout Correspondence back-end

(Work in progress—handle with care)
$Id: libstdout.w,v 1.10 2005/09/12 00:09:21 tlaronde Exp

Thierry LARONDE

Abstract

The ability to dump to stdout can be handy, specially when one can specify the separator and the encoding. Attributes will be printed as lines consisting of fields separated by the specified separator, and encoded in the given locale.

La possibilité d'imprimer sur la sortie standard une version des attributs peut être utile, tout particulièrement lorsqu'il est possible de spécifier le séparateur à utiliser pour séparer les champs, et la locale désirée. La sortie consistera en lignes comportant les champs séparés par le séparateur, les attributs étant traduits d'UTF-8 en la locale demandée.

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` $\equiv 0$ for OK, or the error number if there was one.

```
⟨ corr/stdout.h 2 ⟩ ≡
#ifndef STDOUT_H
#define STDOUT_H
#include "gis.h"
#include "corr.h"
  ⟨ Public prototypes 5 ⟩
#endif /* STDOUT_H */
```

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

4. Defaults.

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

```
⟨ Set defaults 4 ⟩ ≡
  if (Cstdout→Head→separator ≡ '\0') Cstdout→Head→separator ← '|';
  if (Cstdout→Head→null_as ≡ Λ) Cstdout→Head→null_as ← "NULL";
  if (Cstdout→Head→encoding ≡ Λ) Cstdout→Head→encoding ← KT_ICONV_UTF_8;
```

This code is used in section 9.

5. Opening.

```
⟨ Public prototypes 5 ⟩ ≡
  int C_stdout_open(struct C_DFile *Cstdout);
```

See also sections 6, 7, and 8.

This code is used in section 2.

6. Closing will simply release the handler.

```
⟨ Public prototypes 5 ⟩ +≡
  int C_stdout_close(struct C_DFile *Cstdout);
```

7. Getting the records is obvisouly not possible!

```
⟨ Public prototypes 5 ⟩ +≡
  int C_stdout_get_image(struct C_DFile *Cstdout, struct C_Image *Image);
```

8. Writing the records is indeed the main purpose

This will simply print to stdout the *Att.id* and the fields, separated by the specified *Cstdout→Head→separator*.

```
⟨ Public prototypes 5 ⟩ +≡
  int C_stdout_set_image(struct C_DFile *Cstdout, struct C_Image *Image);
```

9. Implementation.

When dealing with stdout, opening is quite simple. There is indeed nothing to really do, except verifying that one does not try to invoke us twice, and that the mode is write only...

```
⟨ stdout/libstdout.c 9 ⟩ ≡
#include "corr/stdout.h"
  static int _Cstdout_handler;  /* 0 initially */
#define _CHECK_HANDLER if (Cstdout→handler ≠ 1) return C_EBADHANDLER
  int C_stdout_open(struct C_DFile *Cstdout)
  {
    int status ← G_SUCCESS;
    Cstdout→handler ← −1;
    if (Cstdout→mode ≠ G_O_WRONLY) {
      status ← C_EMODE;
      goto end;
    }
    ⟨ Set defaults 4 ⟩
    if (¬_Cstdout_handler) {
      _Cstdout_handler ← 1;
      Cstdout→handler ← _Cstdout_handler;
    }
    else status ← C_EOPENMAX;
  end: return status;
  }
```

See also sections 10, 11, and 12.

10. On close, will simply reset internal values.

```
⟨ stdout/libstdout.c 9 ⟩ +≡
  int C_stdout_close(struct C_DFile *Cstdout)
  {
    int status ← G_SUCCESS;
    _CHECK_HANDLER;
    _Cstdout_handler ← 0;
    Cstdout→handler ← −1;
    return status;
  }
```

11. There is nothing to get obviously...

```
⟨ stdout/libstdout.c 9 ⟩ +≡
  /* ARGSUSED */ int C_stdout_get_image(struct C_DFile *Cstdout, struct C_Image *Image)
      {
        return C_EMODE;
      }
```

12. Writing will simply print to stdout a *separator* separated *Att→id* and fields, in the given *encoding*.

```
⟨ stdout/libstdout.c 9 ⟩ +≡
  int C_stdout_set_image(struct C_DFile *Cstdout, struct C_Image *Image)
  {
    int status ← G_SUCCESS;
    unsigned long i;
    struct C_Att *Att;
    _CHECK_HANDLER;
      /* do something, unconditionnally if restoring, only if modified or new in normal case */
    for (i ← 0; i < Image→nb_atts_here; i++) {
      Att ← &Image→Atts[i];
      if (Cstdout→flags & C_RESTORING ∨ C_IS_MODIFIED(Att) ∨ ¬Att→id)
        ⟨ Print the flags, Att→id and fields with separator 13 ⟩
    }
    return status;
  }
```

13. The core of the work is to translate the fields in the requested locale, and to print all the fields, plus leading flags and id, separated by the head specified separator.

```
⟨ Print the flags, Att→id and fields with separator 13 ⟩ ≡
  {
    int i;
    (void) printf("%lu%c%s%c", Att→id, Cstdout→Head→separator,
        (Att→flags & C_ATT_DELETED) ? "DELETED" : "ALIVE", Cstdout→Head→separator);
    for (i ← 0; i < Cstdout→Data→nb_datums; i++) {
      (void) printf("%s%c", (Att→fields[i] ≡ Λ) ? Cstdout→Head→null_as : Att→fields[i],
          (i < Cstdout→Data→nb_datums − 1) ? Cstdout→Head→separator : '\n');
    }
  }
```

This code is used in section 12.

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

⟨ Print the flags, *Att¬id* and fields with separator 13 ⟩ Used in section 12.
⟨ Public prototypes 5, 6, 7, 8 ⟩ Used in section 2.
⟨ Set defaults 4 ⟩ Used in section 9.
⟨ `corr/stdout.h` 2 ⟩
⟨ `stdout/libstdout.c` 9, 10, 11, 12 ⟩