bbclib, une bibliothèque C

Bernard Tatin

2013/2017

Voici un premier essai de *literate programming*, concept inventé par D. Knuth il y a plus de trente ans. À partir de ce seul fichier on génère la documentation et le code. Ici, je reprend du vieux code, cela m'oblige, même s'il est simple, à le repenser et donc, espérons le, à l'améliorer. Même si je passe beaucoup de temps sur la présentation...

Ce code est très orienté *embarqué* car il a été développé principalement dans cette optique. C'est pour cela que certaines gestions d'erreur n'ont pas été développées, faute de place. Il en est de même pour d'autres choix qui pourront paraître curieux au lecteur.

Pour finir cette présentation, les fichiers crées à partir du document maître comme les sources et le PDF sont inclus dans ce dépôt pour permettre une visualisation simple des résultats obtenus sans avoir à installer quoique ce soit de spécial dont `noweb` et LaTeX.[a]

---

[a] Document crée le November 10, 2017à 9:28

Contents

I

du code standard

Ce code doit être capable de supporter le plus grand nombre posible de compilateurs et de machines. `standard.h` est fait pour ça.

I.1 généralités

Tout d'abord, on gère la compatibilité avec le standard *C11* :

4 ⟨*standard-types* 4⟩≡

```
#if defined(__WATCOMC__)
#define no_c11
#define with_watcominline
#elif defined(_MSC_VER)
#define no_c11
#elif defined(__TURBOC__)
#define no_c11
#endif
```

This definition is continued in chunk 5.
This code is used in chunk 8.
Defines:
`no_c11`, used in chunk 5.
`no_c11,`, never used.
`with_watcominline`, used in chunk 6a.

I.2 les types booléens et entiers

C11 défini de nombreux types supplémentaires qu'il faut absolument utiliser :

5 ⟨*standard-types* 4⟩+≡

```
#ifndef no_c11
  #include <stdbool.h>
  #include <stdint.h>
#else
typedef enum {
        false = 0,
        true = 1
} bool;

typedef char int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short uint16_t;
typedef long int32_t;
typedef unsigned long uint32_t;

typedef unsigned int size_t;

#ifndef no_inline
  #define no_inline
#endif
#endif
```

This code is used in chunk 8.
Defines:
`bool`, never used.
`int16_t`, never used.
`int32_t`, never used.
`int8_t`, never used.
`size_t`, never used.
`uint16_t`, never used.
`uint32_t`, never used.
`uint8_t`, never used.
Uses `no_c11` 4 4 4.

I.3 les modificateurs

I.3.1 inline

Certaines fonctions sont **inline**. La diversité des compilateurs nous obligent à définir un INLINE ainsi (et nous ne couvrons pas tous les cas, loin de là):

6a ⟨*define-inline* 6a⟩≡

```
#if defined(with_watcominline)
        #define INLINE __inline
#elif !defined(no_inline)
        #define INLINE inline
#else
        #define INLINE
#endif
```

This code is used in chunk 8.
Defines:
INLINE, used in chunks 11–13.
Uses with_watcominline 4 4.

I.3.2 volatile

Des membres de structure ainsi que des variables sont définis comme **volatile int**. C'est important dans un système embarqué avec des interruptions pouvant manipuler le buffer. Sans **volatile**, une optimisation trop agressive pourrait placer une des valeurs entières dans un registre. En cas d'interruption modifiant cette valeur, le registre, lui, ne bougera pas et des caractères pourraient se perdre. On pourrait définir un VOLATILE en fonction de l'architecture du type :

6b ⟨*define-volatile* 6b⟩≡

```
#if defined(__with_irqs)
  #define VOLATILE volatile
#else
  #define VOLATILE
#endif
```

This code is used in chunks 8 and 15.
Defines:
__with_irqs, never used.
VOLATILE, used in chunk 11a.

Quoiqu'il en soit, il est fortement recommandé de lire la définition exacte du `VOLATILE` de votre compilateur, certaines variations pouvant rendre votre code totalement inefficace. Et d'autant plus que votre compilateur cible un système embarqué où les variations autour des standards sont choses communes.
Cependant, nous ne résolvons pas tous les problèmes, en particuliers ceux du *multi-threading* qui sont laissés à l'utilisateur dans cette version.

I.3.3 INT, l'entier universel

Le type **`int`** permet d'utiliser le type entier offrant en général le meilleur compromis vitesse/taille. Mais ce n'est pas toujours vrai, tout dépend de l'architecture du processeur et des choix des concepteurs du compilateur. On va donc utiliser un **`define`** ce qui autorise la définition sur la igne de commande du compilateur, contrairement au **`typedef`**:

7 ⟨*define-int* 7⟩≡

```
#if !defined(INT)
#define INT int
#endif
```

This code is used in chunks 8 and 15.
Defines:
`INT`, used in chunks 11–14.

I.4 le code final

I.4.1 standard.h

8 ⟨*standard.h* 8⟩≡

```
/*
 * _standard.h
 * generated by noweb
 */
#ifndef INCLUDE__COMPAT__STANDARD_H_
#define INCLUDE__COMPAT__STANDARD_H_

#include <stdlib.h>
#include <stdarg.h>
#include <string.h>

⟨standard-types 4⟩

⟨define-inline 6a⟩
⟨define-volatile 6b⟩
⟨define-int 7⟩

#endif /* INCLUDE__COMPAT__STANDARD_H_ */
```

Root chunk (not used in this document).
Defines:
INCLUDE__COMPAT__STANDARD_H_, never used.

II

rbuffer

C'est un buffer tournant le plus simple possible, capable de gérer des lignes délimitées par *LF* (`'\n'`) mais *CR* (`'\r'`) n'est pas pris en compte, plus exactement, il est rejetté.

II.1 premières définitions

Pour limiter les calculs, le code..., la taille du buffer est une puissance de 2 d'où la définition du nombre de bits qui ouvre le bal, et en tenant compte du fait que cette définition peut-être donnée en paramètre du préprocesseur :

10a ⟨*intro-bits* 10a⟩≡

```
# if !defined(_RBUFFER_BITS)
#define _RBUFFER_BITS   8
#endif
```

This definition is continued in chunk 10.
This code is used in chunk 15.
Defines:
  `_RBUFFER_BITS`, used in chunk 10b.

La taille du buffer sera donc :

10b ⟨*intro-bits* 10a⟩+≡

```
#define RBUFFER_SIZE    (1 << _RBUFFER_BITS)
```

This code is used in chunk 15.
Defines:
  `RBUFFER_SIZE`, used in chunks 10c and 11a.
Uses `_RBUFFER_BITS` 10a.

Et le masque permettant un rapide *modulo* arithmétique avec un **and** binaire :

10c ⟨*intro-bits* 10a⟩+≡

```
#define RBUFFER_MASK    (RBUFFER_SIZE - 1)
```

This code is used in chunk 15.
Defines:
  `RBUFFER_MASK`, used in chunks 11b and 12a.
Uses `RBUFFER_SIZE` 10b.

II.2 la structure

La voici :

11a ⟨*tsrbuffer-final* 11a⟩≡

```
typedef struct {
    VOLATILE INT in;
    VOLATILE INT out;
    VOLATILE INT line_count;
    VOLATILE char buffer[RBUFFER_SIZE];
} TSrbuffer;
```

This code is used in chunk 15.
Defines:
TSrbuffer, used in chunks 11–14.
Uses **INT** 7 7, **RBUFFER_SIZE** 10b, and **VOLATILE** 6b.

II.3 le fonctionnement

II.3.1 ajout d'un caractère

Le fonctionnement est le suivant pour l'ajout d'un caractère :

- si le caractère est '\r', on ne fait rien,
- on place le caractère dans le buffer à la position **in**,
- on incrémente **in**,
- si on atteint la limite du buffer, on positionne **in** à 0,
- si le caractère est '\n', on incrémente **line_count**.

11b ⟨*add-char* 11b⟩≡

```
static INLINE void rbf_add_char(TSrbuffer *rb, const char c) {
    if (c != '\r') {
        rb->buffer[rb->in++] = c;
        rb->in &= RBUFFER_MASK;
        if (c == '\n') {
            rb->line_count++;
        }
    }
}
```

This code is used in chunk 15.
Defines:
rbf_add_char, used in chunk 13c.
Uses **INLINE** 6a, **RBUFFER_MASK** 10c, and **TSrbuffer** 11a 11a.

La récupération d'un caractère dans le buffer est l'algorithme inverse :

12a ⟨*get-char* 12a⟩≡
```
static INLINE char rbf_get_char(TSrbuffer *rb) {
    INT out = rb->out;
    char c = rb->buffer[out++];
    out &= RBUFFER_MASK;
    rb->out = out;
    if (c == '\n' && rb->line_count) {
        rb->line_count--;
    }
    return c;
}
```
This code is used in chunk 15.
Defines:
`rbf_get_char`, used in chunk 14.
Uses `INLINE` 6a, `INT` 7 7, `RBUFFER_MASK` 10c, and `TSrbuffer` 11a 11a.

Il est cependant très important de déterminer si des caratères sont présents dans le buffer :

12b ⟨*has-chars* 12b⟩≡
```
static INLINE bool rbf_has_chars(TSrbuffer *rb) {
    return rb->in != rb->out;
}
```
This code is used in chunk 15.
Defines:
`bool`, never used.
Uses `INLINE` 6a and `TSrbuffer` 11a 11a.

Le marquage d'une fin de ligne se fait par un '\0' :

13a ⟨*end-of-line* 13a⟩≡

```
static INLINE void rbf_end_of_line(TSrbuffer *rb) {
    rb->buffer[rb->in] = 0;
    rb->line_count++;
}
```

Root chunk (not used in this document).
Defines:
  `rbf_end_of_line`, used in chunk 13c.
Uses `INLINE` 6a and `TSrbuffer` 11a 11a.

Ces fonctions, nécessitant une boucle, ne sont pas déclarées `INLINE` :

13b ⟨*more-functions-h* 13b⟩≡

```
void rbf_add_line(TSrbuffer *rb, char *line);
INT rbf_get_line(TSrbuffer *rb, char *line);
```

This code is used in chunk 15.
Uses `INT` 7 7, `rbf_add_line` 13c, `rbf_get_line` 14, and `TSrbuffer` 11a 11a.

L'ajout d'une ligne est *simple* :

13c ⟨*more-functions-c* 13c⟩≡

```
void rbf_add_line(TSrbuffer *rb, char *line) {
    char c;

    while ((c = *(line++)) != 0) {
        rbf_add_char(rb, c);
    }
    rbf_end_of_line(rb);
}
```

This definition is continued in chunk 14.
This code is used in chunk 16.
Defines:
  `rbf_add_line`, used in chunk 13b.
Uses `rbf_add_char` 11b, `rbf_end_of_line` 13a, and `TSrbuffer` 11a 11a.

Et la lecture d'une ligne :

14 ⟨*more-functions-c* 13c⟩+≡

```
INT rbf_get_line(TSrbuffer *rb, char *line) {
    char c;
    INT r = 0;

    while (rbf_has_chars(rb)) {
        c = rbf_get_char(rb);

        if (c == '\n') {
            break;
        }
        if (c != 0) {
            *(line++) = c;
        }
        r++;
    }
    *line = 0;
    return r;
}
```

This code is used in chunk 16.
Defines:
`rbf_get_line`, used in chunk 13b.
Uses `INT` 7 7, `rbf_get_char` 12a, and `TSrbuffer` 11a 11a.

II.4 le code final

II.4.1 rbuffer.h

15 ⟨*rbuffer.h* 15⟩≡

```
/*
 * rbuffer.h
 * generated by noweb
 */

#if !defined(__rbuffer_h__)
#define __rbuffer_h__

⟨intro-bits 10a⟩

⟨define-volatile 6b⟩
⟨define-int 7⟩

⟨tsrbuffer-final 11a⟩

⟨add-char 11b⟩

⟨get-char 12a⟩

⟨has-chars 12b⟩

⟨more-functions-h 13b⟩

#endif // __rbuffer_h__
```

Root chunk (not used in this document).
Defines:
`__rbuffer_h__`, never used.

II.4.2 rbuffer.c

16 ⟨*rbuffer.c* 16⟩≡

```
/*
 * rbuffer.c
 * generated by noweb
 */

#include "standard.h"
#include "rbuffer.h"
```

⟨*more-functions-c* 13c⟩

Root chunk (not used in this document).

III

annexes

III.1 la ligne de commande

Pour obtenir le fichier LaTeX et le code source, voici ce qu'il faut faire depuis un terminal :

18 ⟨*command-line* 18⟩≡

```
# fichier LaTeX
noweave -delay -autodefs c -index rbuffer.nw > rbuffer.tex
# fichier PDF
pdflatex rbuffer.tex && \
  pdflatex rbuffer.tex && \
  pdflatex rbuffer.tex
# le code source
notangle rbuffer.nw > rbuffer.h
```

Root chunk (not used in this document).

L'option `-autodefs c` permet à `noweave` de déterminer lui-même les éléments du langage C. Sans cette option, dans le cadre de ce fichier, les définitions de `intro-bits` ne seraient pas visibles.

III.2 tables et index

III.2.1 table des extraits de code

III.2.2 index