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 8, 2017à 23:22

Contents

Part 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.

1. généralités

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

2 ⟨*standard* 2⟩≡
```
#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 3.
Root chunk (not used in this document).
Defines:

`no_c11`, used in chunks 3 and 9.
`no_c11,`, never used.
`with_watcominline`, used in chunk 10a.

---

2. les types booléens et entiers

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

---

3 ⟨*standard* 2⟩+≡

```
#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
```

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 `false` 9, `no_c11` 2 2 2, `no_inline` 9, and `true` 9.

---

3. le code final

3.1. standard.h

4 ⟨*standard.h* 4⟩≡

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

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

⟨type-bool 9⟩

⟨define-inline 10a⟩

#endif /* INCLUDE__COMPAT__STANDARD_H_ */
```

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

Part 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.

4. 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 :

5a ⟨*intro-bits* 5a⟩≡

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

This definition is continued in chunk 5.
This code is used in chunk 14.
Defines:
  _RBUFFER_BITS, used in chunk 5b.

La taille du buffer sera donc :

5b ⟨*intro-bits* 5a⟩+≡

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

This code is used in chunk 14.
Defines:
  RBUFFER_SIZE, used in chunks 5c, 6, and 8.
Uses **_RBUFFER_BITS** 5a.

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

5c ⟨*intro-bits* 5a⟩+≡

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

This code is used in chunk 14.
Defines:
  RBUFFER_MASK, used in chunks 10b and 11a.
Uses **RBUFFER_SIZE** 5b.

5. la structure

La voici :

6 ⟨*tsrbuffer* 6⟩≡

```
typedef struct {
    volatile int in;
    volatile int out;
    volatile int line_count;
    volatile char buffer[RBUFFER_SIZE];
} TSrbuffer;
```

Root chunk (not used in this document).
Defines:
  TSrbuffer, used in chunks 10–13.
Uses **RBUFFER_SIZE** 5b.

5.1. remarques diverses

Tous les membres de la structure 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 :

7a ⟨*define-volatile* 7a⟩≡

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

This code is used in chunk 14.
Defines:
`__with_irqs`, never used.
`VOLATILE`, used in chunk 8.

L'utilisation du type `int` permet d'utiliser le type entier permettant 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`:

7b ⟨*define-int* 7b⟩≡

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

This code is used in chunk 14.
Defines:
`INT`, used in chunks 8 and 11–13.

Ce qui donnerait au final :

8 ⟨*tsrbuffer-final* 8⟩≡

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

This code is used in chunk 14.
Defines:
TSrbuffer, used in chunks 10–13.
Uses INT 7b 7b, RBUFFER_SIZE 5b, and VOLATILE 7a.

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.

6. le fonctionnement

6.1. types et modificateurs

On utilise `bool` qui n'est pas défini avant *C11* :

9 ⟨*type-bool* 9⟩≡

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

This code is used in chunk 4.
Defines:
`bool`, never used.
`false`, used in chunk 3.
`no_inline`, used in chunks 3 and 10a.
`true`, used in chunk 3.
Uses `no_c11` 2 2 2.

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):

10a ⟨*define-inline* 10a⟩≡
```
#if defined(with_watcominline)
        #define INLINE __inline
#elif !defined(no_inline)
        #define INLINE inline
#else
        #define INLINE
#endif
```
This code is used in chunk 4.
Defines:
INLINE, used in chunks 10 and 11.
Uses **no_inline** 9 and **with_watcominline** 2 2.

6.2. 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**.

10b ⟨*add-char* 10b⟩≡
```
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 14.
Defines:
rbf_add_char, used in chunk 12b.
Uses **INLINE** 10a, **RBUFFER_MASK** 5c, and **TSrbuffer** 6 8 8.

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

11a ⟨*get-char* 11a⟩≡

```
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 14.
Defines:
`rbf_get_char`, used in chunk 13.
Uses `INLINE` 10a, `INT` 7b 7b, `RBUFFER_MASK` 5c, and `TSrbuffer` 6 8 8.

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

11b ⟨*has-chars* 11b⟩≡

```
static INLINE bool rbf_has_chars(TSrbuffer *rb) {
    return rb->in != rb->out;
}
```

This code is used in chunk 14.
Defines:
`bool`, never used.
Uses `INLINE` 10a and `TSrbuffer` 6 8 8.

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

11c ⟨*end-of-line* 11c⟩≡

```
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 12b.
Uses `INLINE` 10a and `TSrbuffer` 6 8 8.

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

12a ⟨*more-functions-h* 12a⟩≡
```
void rbf_add_line(TSrbuffer *rb, char *line);
INT rbf_get_line(TSrbuffer *rb, char *line);
```
This code is used in chunk 14.
Uses `INT` 7b 7b, `rbf_add_line` 12b, `rbf_get_line` 13, and `TSrbuffer` 6 8 8.

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

12b ⟨*more-functions-c* 12b⟩≡
```
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 13.
This code is used in chunk 15a.
Defines:
`rbf_add_line`, used in chunk 12a.
Uses `rbf_add_char` 10b, `rbf_end_of_line` 11c, and `TSrbuffer` 6 8 8.

Et la lecture d'une ligne :

13 ⟨*more-functions-c* 12b⟩+≡

```
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 15a.
Defines:
`rbf_get_line`, used in chunk 12a.
Uses `INT` 7b 7b, `rbf_get_char` 11a, and `TSrbuffer` 6 8 8.

7. le code final

7.1. rbuffer.h

14 ⟨*rbuffer.h* 14⟩≡

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

#if !defined(__rbuffer_h__)
#define __rbuffer_h__

⟨intro-bits 5a⟩

⟨define-volatile 7a⟩
⟨define-int 7b⟩

⟨tsrbuffer-final 8⟩

⟨add-char 10b⟩

⟨get-char 11a⟩

⟨has-chars 11b⟩

⟨more-functions-h 12a⟩

#endif // __rbuffer_h__
```

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

7.2. rbuffer.c

15a ⟨*rbuffer.c* 15a⟩≡

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

#include "rbuffer.h"
```

⟨*more-functions-c* 12b⟩

Root chunk (not used in this document).

Part III.

annexes

8. la ligne de commande

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

15b ⟨*command-line* 15b⟩≡

```
# 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.

9. tables et index

9.1. table des extraits de code

9.2. index