

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 créés 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 .

Contents

I. rbuffer	2
1. premières définitions	2
2. la structure	3
2.1. les champs	3
2.2. remarques diverses	3
3. le fonctionnement	5
3.1. types et modificateurs	5
3.2. ajout d'un caractère	6
4. le code final	8
4.1. <code>standard.h</code>	8
4.2. <code>rbuffer.h</code>	9

4.3. <code>rbuffer.c</code>	9
II. annexes	10
1. la ligne de commande	10
2. tables et index	11
2.1. table des extraits de code	11
2.2. index	11

Part I.

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 rejeté.

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 :

2a `<intro-bits 2a>≡`
`# if !defined(_RBUFFER_BITS)`
`#define _RBUFFER_BITS 8`
`#endif`

This definition is continued in chunks 2b and 3a.

This code is used in chunk 9a.

Defines:

`_RBUFFER_BITS`, used in chunk 2b.

La taille du buffer sera donc :

2b `<intro-bits 2a>+≡`
`#define RBUFFER_SIZE (1 << _RBUFFER_BITS)`

This code is used in chunk 9a.

Defines:

`RBUFFER_SIZE`, used in chunks 3 and 4b.

Uses `_RBUFFER_BITS` 2a.

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

3a $\langle \text{intro-bits } 2a \rangle + \equiv$
`#define RBUFFER_MASK (RBUFFER_SIZE - 1)`

This code is used in chunk 9a.

Defines:

RBUFFER_MASK, used in chunk 6.

Uses **RBUFFER_SIZE** 2b.

2. la structure

La voici :

3b $\langle \text{tsrbuffer } 3b \rangle \equiv$

```
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 6–8.

Uses **RBUFFER_SIZE** 2b.

2.1. les champs

2.2. 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 :

3c $\langle \text{define-volatile } 3c \rangle \equiv$

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

This code is used in chunk 9a.

Defines:

__with_irqs,, never used.

VOLATILE, used in chunk 4b.

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 ligne de commande du compilateur, contrairement au **typedef**:

4a $\langle \text{define-int } 4a \rangle \equiv$

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

This code is used in chunk 9a.

Defines:

INT, used in chunks 4b and 6–8.

Ce qui donnerait au final :

4b $\langle \text{tsrbuffer-final } 4b \rangle \equiv$

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

This code is used in chunk 9a.

Defines:

TSrbuffer, used in chunks 6–8.

Uses **INT** 4a 4a, **RBUFFER_SIZE** 2b, and **VOLATILE** 3c.

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.

3. le fonctionnement

3.1. types et modificateurs

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

5a `<type-bool 5a>≡`

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

This code is used in chunk 8b.

Defines:

bool, never used.
false, never used.
no_inline, used in chunk 5b.
true, never used.

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):

5b `<define-inline 5b>≡`

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

This code is used in chunk 8b.

Defines:

INLINE, used in chunks 6 and 7a.

Uses **no_inline** 5a.

3.2. 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`.

6a `<add-char 6a>≡`

```
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 9a.

Defines:

`rbf_add_char`, used in chunk 7c.

Uses `INLINE 5b`, `RBUFFER_MASK 3a`, and `TSrbuffer 3b 4b 4b`.

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

6b `<get-char 6b>≡`

```
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 9a.

Defines:

`rbf_get_char`, used in chunk 8a.

Uses `INLINE 5b`, `INT 4a 4a`, `RBUFFER_MASK 3a`, and `TSrbuffer 3b 4b 4b`.

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

6c `<has-chars 6c>≡`

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

This code is used in chunk 9a.

Defines:

`bool`, never used.

Uses `INLINE 5b` and `TSrbuffer 3b 4b 4b`.

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

7a `<end-of-line 7a>`≡

```
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 7c.

Uses **INLINE** 5b and **TSrbuffer** 3b 4b 4b.

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

7b `<more-functions-h 7b>`≡

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

This code is used in chunk 9a.

Uses **INT** 4a 4a, **rbf_add_line** 7c, **rbf_get_line** 8a, and **TSrbuffer** 3b 4b 4b.

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

7c `<more-functions-c 7c>`≡

```
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 8a.

This code is used in chunk 9b.

Defines:

rbf_add_line, used in chunk 7b.

Uses **rbf_add_char** 6a, **rbf_end_of_line** 7a, and **TSrbuffer** 3b 4b 4b.

Et la lecture d'une ligne :

```
8a  <more-functions-c 7c>+≡
    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 9b.

Defines:

rbf_get_line, used in chunk 7b.

Uses **INT** 4a 4a, **rbf_get_char** 6b, and **TSrbuffer** 3b 4b 4b.

4. le code final

4.1. standard.h

```
8b  <standard.h 8b>≡
    /*
     * _standard.h
     * generated by noweb
     */
    #ifndef INCLUDE__COMPAT__STANDARD_H_
    #define INCLUDE__COMPAT__STANDARD_H_

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

    <type-bool 5a>

    <define-inline 5b>

    #endif /* INCLUDE__COMPAT__STANDARD_H_ */
```

Root chunk (not used in this document).

Defines:

INCLUDE__COMPAT__STANDARD_H_, never used.

4.2. rbuffer.h

9a `<rbuffer.h 9a>≡`
`/*`
 `* rbuffer.h`
 `* generated by noweb`
 `*/`

`#if !defined(__rbuffer_h__)`
`#define __rbuffer_h__`

`<intro-bits 2a>`

`<define-volatile 3c>`
`<define-int 4a>`

`<tsrbuffer-final 4b>`

`<add-char 6a>`

`<get-char 6b>`

`<has-chars 6c>`

`<more-functions-h 7b>`

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

4.3. rbuffer.c

9b `<rbuffer.c 9b>≡`
`/*`
 `* rbuffer.c`
 `* generated by noweb`
 `*/`

`#include "rbuffer.h"`

`<more-functions-c 7c>`
Root chunk (not used in this document).

Part II.

annexes

1. la ligne de commande

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

```
10 <command-line 10>≡  
  # 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.

2. tables et index

2.1. table des extraits de code

`<add-char 6a>` [6a](#), [9a](#)
`<command-line 10>` [10](#)
`<define-inline 5b>` [5b](#), [8b](#)
`<define-int 4a>` [4a](#), [9a](#)
`<define-volatile 3c>` [3c](#), [9a](#)
`<end-of-line 7a>` [7a](#)
`<get-char 6b>` [6b](#), [9a](#)
`<has-chars 6c>` [6c](#), [9a](#)
`<intro-bits 2a>` [2a](#), [2b](#), [3a](#), [9a](#)
`<more-functions-c 7c>` [7c](#), [8a](#), [9b](#)
`<more-functions-h 7b>` [7b](#), [9a](#)
`<rbuffer.c 9b>` [9b](#)
`<rbuffer.h 9a>` [9a](#)
`<standard.h 8b>` [8b](#)
`<tsrbuffer 3b>` [3b](#)
`<tsrbuffer-final 4b>` [4b](#), [9a](#)
`<type-bool 5a>` [5a](#), [8b](#)

2.2. index

`__rbuffer_h__`: [9a](#)
`__with_irqs,` [3c](#)
`_RBUFFER_BITS`: [2a](#), [2b](#)
`bool`: [5a](#), [5a](#), [6c](#)
`false`: [5a](#)
`INCLUDE__COMPAT__STANDARD_H_`: [8b](#)
`INLINE`: [5b](#), [6a](#), [6b](#), [6c](#), [7a](#)
`INT`: [4a](#), [4a](#), [4b](#), [6b](#), [7b](#), [8a](#)
`no_inline`: [5a](#), [5b](#)
`rbf_add_char`: [6a](#), [7c](#)
`rbf_add_line`: [7b](#), [7c](#)
`rbf_end_of_line`: [7a](#), [7c](#)
`rbf_get_char`: [6b](#), [8a](#)
`rbf_get_line`: [7b](#), [8a](#)
`RBUFFER_MASK`: [3a](#), [6a](#), [6b](#)
`RBUFFER_SIZE`: [2b](#), [3a](#), [3b](#), [4b](#)
`true`: [5a](#)

TSrbuffer: 3b, 4b, 4b, 6a, 6b, 6c, 7a, 7b, 7c, 8a
VOLATILE: 3c, 4b