

# rbuffer.h, un buffer tournant

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...

---

## Contents

<b>1</b>	<b>rbuffer</b>	<b>1</b>
1.1	premières définitions . . . . .	2
1.2	la structure . . . . .	3
1.2.1	les champs . . . . .	3
1.2.2	remarques diverses . . . . .	3
1.3	le fonctionnement . . . . .	4
1.3.1	ajout d'un caractère . . . . .	4
1.4	le code final . . . . .	5
<b>2</b>	<b>annexes</b>	<b>5</b>
2.1	la ligne de commande . . . . .	5
<b>3</b>	<b>tables et index</b>	<b>6</b>
3.1	table des extraits de code . . . . .	6
3.2	index . . . . .	6

## 1 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.

## 1.1 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 :

2a  $\langle \text{intro-bits } 2a \rangle \equiv$   
`#define _RBUFFER_BITS 8`

This definition is continued in chunk 2.

This code is used in chunk 5a.

Defines:

`_RBUFFER_BITS`, used in chunk 2b.

La taille du buffer sera donc :

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

This code is used in chunk 5a.

Defines:

`RBUFFER_SIZE`, used in chunks 2–4.

Uses `_RBUFFER_BITS` 2a.

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

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

This code is used in chunk 5a.

Defines:

`RBUFFER_MASK`, used in chunk 4b.

Uses `RBUFFER_SIZE` 2b.

## 1.2 la structure

**Note:** tous les membres de la structure sont définis comme **volatile**. 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.

```
3a  <tsrbuffer 3a>≡
    /**
     * @struct TSrbuffer
     * La structure gérant le buffer tournant.
     */
    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 chunk 4b.

Uses RBUFFER\_SIZE 2b.

### 1.2.1 les champs

### 1.2.2 remarques diverses

On pourrait définir un **VOLATILE** en fonction de l'architecture du type :

```
3b  <define-volatile 3b>≡
    #if defined(__with_irqs)
        #define VOLATILE volatile
    #else
        #define VOLATILE
    #endif
```

This code is used in chunk 5a.

Ce qui donnerait au final :

```
4a <tsrbuffer-final 4a>≡
/**
 * @struct TSrbuffer
 * La structure gérant le buffer tournant.
 */
typedef struct {
    VOLATILE int in;
    VOLATILE int out;
    VOLATILE int line_count;
    VOLATILE char buffer[RBUFFER_SIZE];
} TSrbuffer;
```

This code is used in chunk 5a.

Defines:

**TSrbuffer**, used in chunk 4b.

Uses **RBUFFER\_SIZE** 2b.

## 1.3 le fonctionnement

### 1.3.1 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**.

```
4b <add-char 4b>≡
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 5a.

Defines:

**rbf\_add\_char**, never used.

Uses **RBUFFER\_MASK** 2c and **TSrbuffer** 3a 4a 4a.

## 1.4 le code final

5a  $\langle * 5a \rangle \equiv$   
     $\langle intro-bits 2a \rangle$   
     $\langle define-volatile 3b \rangle$   
     $\langle tsrbuffer-final 4a \rangle$   
     $\langle add-char 4b \rangle$

Root chunk (not used in this document).

## 2 annexes

### 2.1 la ligne de commande

Pour obtenir le fichier L<sup>A</sup>T<sub>E</sub>X et le code source, voici ce qu'il faut faire depuis un terminal :

5b  $\langle command-line 5b \rangle \equiv$   
    # 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.

## 3 tables et index

### 3.1 table des extraits de code

`<* 5a>` [5a](#)  
`<add-char 4b>` [4b](#), [5a](#)  
`<command-line 5b>` [5b](#)  
`<define-volatile 3b>` [3b](#), [5a](#)  
`<intro-bits 2a>` [2a](#), [2b](#), [2c](#), [5a](#)  
`<tsrbuffer 3a>` [3a](#)  
`<tsrbuffer-final 4a>` [4a](#), [5a](#)

### 3.2 index

`_RBUFFER_BITS`: [2a](#), [2b](#)  
`rbf_add_char`: [4b](#)  
`RBUFFER_MASK`: [2c](#), [4b](#)  
`RBUFFER_SIZE`: [2b](#), [2c](#), [3a](#), [4a](#)  
`TSrbuffer`: [3a](#), [4a](#), [4a](#), [4b](#)