TP n°4

Informatique embarquée

Vincent Ruello

À rendre avant mercredi 15/11/2023 - 16:00 (CET)

Le but de ce TP est d'implémenter une bibliothèque permettant de « faciliter » l'utilisation du périphérique USART du microcontrôleur ATmega328P.

Contexte

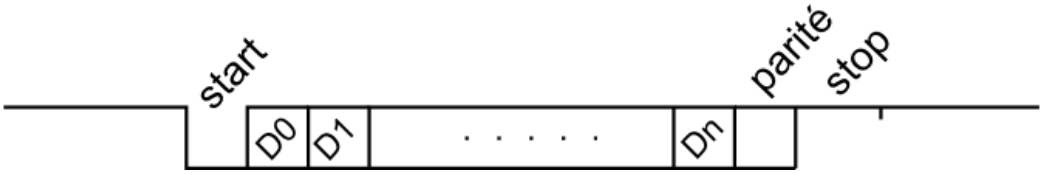
La carte Arduino UNO est une plateforme open-source de développement permettant de programmer un ATmega328P. La documentation du microcontrôleur ATmega328P est disponible ici[1]. AVR Libc est une bibliothèque open-source dont le but est de fournir une libc utilisable sur l'architecture AVR avec GCC. Sa documentation est disponible ici[2].

Partie 1. Trame UART

On rappelle qu'une trame UART est constituée des bits suivants :

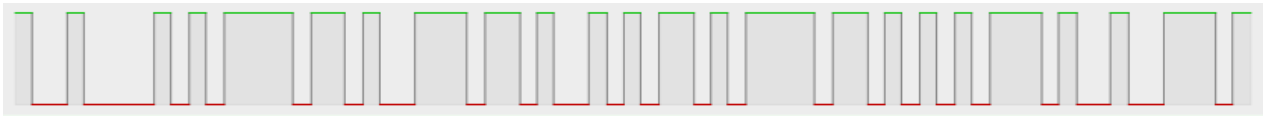
- un bit de start, toujours à 0 ;
- des bits de données, dont le nombre peut être configuré entre 5 et 9 bits. En général, les bits sont envoyés en commençant par le bit de poids faible ;
- optionnellement, un bit de parité ;
- un bit de stop, toujours à 1. Ce bit peut durer 1, 1.5 ou 2 cycles d'horloge.

En l'absence de transmission, le niveau logique est à 1.



A l'aide d'un analyseur logique, on a enregistré le niveau logique d'une ligne au cours du temps lors d'une transmission UART. Durant cette transmission, le nombre de bits de données a été configuré à **8**, **sans bit de parité** et avec le **bit de stop sur un cycle d'horloge**. Voici les niveaux capturés :

1 http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf
2 https://www.nongnu.org/avr-libc/user-manual/index.html



Cette capture peut être représentée par la suite de bits suivante où chaque bit représente la valeur logique prise par la ligne sur un cycle d'horloge :

```
11100100001010111101101001110110100101011010111101101010111010010011101111
```

1. Quelles sont les données échangées ? *(indice : c'est du texte)*
2. Comme de nombreux protocoles de communication, les trames UART peuvent être représentées avec un automate fini. Dessiner cet automate dans le cas où on transmet 8 bits de données, sans bit de parité et avec un bit de stop.

Pour aller plus loin :

3. Dans le langage de votre choix, implémenter un programme qui prend en entrée une chaîne de caractères composée de « 0 » et de « 1 » représentant les niveaux logiques observés lors d'une transmission UART et affiche les données utiles reçues sous forme de caractères. On utilisera l'automate précédement décrit, avec la même configuration.

Partie 2. Périphérique USART

L'ATmega328P possède un périphérique USART0 permettant de recevoir et d'émettre des trames UART.

1. Sur quelles broches du microcontrôleur sont reliées les ports RX et TX ?

2. D'après le schéma de la carte Arduino, à quel élément sont reliées ces broches ? Pourquoi ?

Sous Linux, le logiciel *screen* permet d'émuler un terminal VT100. Il peut interagir directement avec un *device* TTY et le configurer (par exemple le baud rate utilisé ou le nombre de bits de données par trame). Pour se connecter au TTY `/dev/ttyACM0` et utiliser un baud rate 9600, on pourra utiliser la commande suivante :

```
screen /dev/ttyACM0 9600
```

La carte Arduino Uno est faite de telle sorte à ce que l'ATmega328P reçoive un signal RESET lorsqu'un terminal est « prêt » à communiquer avec le microcontrôleur via la liaison série (voir ligne DTR du schéma de la carte).

Le but de cette partie est d'écrire un programme qui renvoie les informations qu'il reçoit (permettant ainsi à l'utilisateur de « voir » ce qu'il tape dans l'émulateur de terminal.

3. Ecrire une fonction d'initialisation du périphérique USART0. Elle prendra en argument le Baud Rate à utiliser. Les autres paramètres de configuration sont fixes : 8 bits de données, pas de bit de parité, 1 bit de stop.

   ```
   void UART__init(uint32_t);
   ```

4. Ecrire une fonction permettant de lire un caractère reçu via le périphérique USART0. La fonction ne retourne pas tant qu'elle n'a pas reçu de caractère (avec du polling) :

   ```
   uint8_t UART__getc();
   ```

5. Ecrire une fonction permettant d'envoyer un caractère via le périphérique USART0. La fonction ne retourne pas tant que le caractère n'a pas été envoyé (avec du polling) :

   ```
   void UART__putc(uint8_t);
   ```

On veillera à implémenter ces 3 fonctions dans un fichier séparé `uart.c`.

6. A l'aide des fonctions précédentes, écrire un programme de sorte à ce que le microcontrôleur transmette via son périphérique USART0 chaque caractère qu'il reçoit via ce même périphérique USART0.

7. A chaque caractère reçu, on allumera la LED intégrée à la carte pendant 5 secondes. On utilisera pour cela une attente active avec la fonction `_delay_ms` de avr-libc. Que se passe-t'il lorsque plusieurs caractères sont reçus par le microcontrôleur lorsque la LED est allumée ?

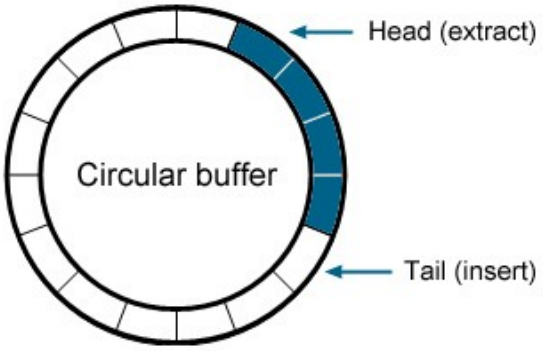
Pour palier à ce problème, il est souhaitable d'utiliser les interruptions du périphérique USART0 et d'implémenter un buffer de réception.

Partie 3. Buffer de réception et interruptions

Lorsqu'un caractère est reçu, le périphérique USART0 peut déclencher une interruption RXC (Receive Complete Interrupt). Le but de cette partie est d'implémenter un handler d'interruption qui va stocker chaque nouveau caractère reçu dans un buffer de réception.

Un buffer de réception se présente souvent sous la forme d'une FIFO. On propose d'utiliser un buffer circulaire, structure de données très utilisée pour implémenter une FIFO avec une taille de buffer fixe (notée N par la suite). Il est souvent implémenté à l'aide d'un tableau (de taille fixe) et de deux indices : *head* et *tail*.

Voici une réprésentation conceptuelle d'un buffer circulaire :



1. Quelle est la relation entre head et tail lorsque le buffer est vide ? plein ?

Il existe plusieurs solutions pour différencier ces états, comme par exemple :
- n'utiliser que N-1 éléments du buffer, avec N la taille du buffer ;
- stocker en mémoire le nombre d'éléments présents dans le buffer.

On propose de choisir la première solution (c'est à dire de n'utiliser que N-1 éléments du buffer).

Lorsque le buffer circulaire est plein, deux approches sont possibles : écraser les données existantes (en commençant par les plus anciennes) ou ne pas tenir des comptes des nouvelles données. On propose de choisir la première solution (écrasement).

De manière similaire, lorsqu'on souhaite récupérer une donnée mais que le buffer est vide, deux approches sont possibles : retourner une « valeur » spécifique signifiant que « le buffer est vide » (fonctionnement non bloquant) ou attendre qu'une donnée arrive et la renvoyer (fonctionnement bloquant). On choisira la première approche (retourner une « valeur » spécifique).

Avant de passer à l'implémentation, quelques remarques et conseils :
- Le buffer circulaire sera manipulé par le thread principal (pop) et par des ISR (push). De fait, une ISR peut s'exécuter au milieu de la fonction pop. Il faut donc prendre en compte les problèmes éventuels de concurrence.
- Le buffer étant manipulé par une ISR, il doit être déclaré comme une variable globale en C.
- Les opérations modulo sont à éviter.

On propose d'utiliser la structure et les prototypes suivants :

```
struct ring_buffer {
    uint8_t* buffer;
    uint8_t head;
    uint8_t tail;
    uint8_t maxlen;
};

void ring_buffer__init(struct ring_buffer* rb, uint8_t* buffer, uint8_t
buffer_size);
void ring_buffer__push(struct ring_buffer* rb, uint8_t data);
uint8_t ring_buffer__pop(struct ring_buffer* rb, uint8_t* data);
```

2. Implémenter le handler d'interruption RXC tel que décrit précédemment et modifier les fonctions `UART__getc` et `UART__init` en conséquence en utilisant la structure

`ring_buffer` et ses fonctions. On pourra obtenir un exécutable en compilant avec le header `ring_buffer.h` et en fournissant au linker la bibliothèque statique `libring_buffer.a` (ring_buffer.tar.gz[3]).

Pour aller plus loin :

3. Ré-implémenter les fonctions `ring_buffer__init`, `ring_buffer__push`, et `ring_buffer__pop` (sans utiliser la bibliothèque statique fournie à la question 2).

Partie 4. Bibliothèque stdio

La bibliothèque stdio contient de nombreuses fonctions permettant de travailler avec les entrées et les sorties. On y retrouve notamment la célèbre fonction `printf`, mais également les définitions de `stdin`, `stdout` et `stderr`.

Si certaines de ces fonctionnalités et de ses concepts n'ont pas de sens sans un système d'exploitation et un système de fichiers, beaucoup d'autres sont implémentés ou/et adaptés dans avr-libc. Il est notamment possible de définir les flux `stdin`, `stdout` et `stderr` et ainsi d'utiliser bon nombre de fonctions usuelles (dont `printf`).

En lisant la documentation[4] de avr-libc, adapter votre code de telle sorte à ce que la fonction `printf` écrive des caractères via le périphérique USART0 et que `gets` lise des caractères via le périphérique USART0.

Partie 5. Application

De nombreux équipements communiquent entre eux en passant par un module UART. Par exemple, le module AT-09 permet de piloter un MCU avec un module Bluetooth Low Energy (CC2540) via une liaison série et un jeu de commandes.

Le but de cette partie est, avec une carte Arduino Uno « maître », de piloter l'état des GPIOs (ports B, C et D) d'une carte Arduino « esclave », les deux cartes étant reliées via leur module UART.

Il est donc nécessaire d'écrire 2 programmes :

- Le programme « maître » : il devra envoyer des commandes via son périphérique USART0 à l'esclave. On pourra par exemple le programmer pour qu'il fasse clignoter la LED intégrée de la carte esclave avec le motif utilisé lors des précédents TPs.
- Le programme « esclave » : il devra écouter, interpréter et exécuter les commandes reçues via son périphérique USART0.

Le format des commandes est libre. On veillera à valider les données échangées.

3 https://vps.vruello.fr/k9npX2nqL7/resources/ring_buffer.tar.gz
4 https://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html