TD Communication dans les systèmes informatiques
Communications séries asynchrones – UART Arduino

1. Etude d'une trame en communication série asynchrone :

1.1 Quel est le rôle du bit de parité dans une trame série asynchrone ?
1.2 En considérant une communication 9600bds, 8 bits de données, pas de parité, 1 bit de stop, quelle est la durée du bit de stop ? Quel est l'impact du nombre de bits de stop sur la communication ?
1.3 A débit fixé (par exemple 9600 bds) comment rendre la communication plus rapide ?
1.4 Dans une transmission asynchrone à 9600 bds 8N1, quelle est la durée nécessaire à l'émission d'un caractère?
1.5 Représentez le chronogramme de la transmission (sans attente) de 0x02A01358 en msB first (au format décrit en 1.2).

2. Gestion d'un buffer circulaire :

Dans les communications entre systèmes informatiques, on utilise une mémoire tampon pour le stockage des caractères reçus ou en attente d'émission. Cette mémoire tampon permet :
- de limiter l'utilisation des signaux de protocole pour le contrôle de flux,
- de construire facilement le programme par couches (par ex. une couche de réception, et une couche d'utilisation, cf. réseaux). Cette démarche est très adaptée à une gestion par interruption.

Nous proposons dans cet exercice de gérer la mémoire de façon circulaire. Cette gestion se fait aussi bien pour le « remplissage » du tampon que pour le « vidage ».

Remplissage du tampon : les données reçues sont écrites dans le tampon tant qu'il y a de la place ; si la fin du tampon est atteinte, on recommence à écrire dès le début.
Vidage du tampon : on ne peut venir lire dans le tampon uniquement si des données sont disponibles.

Le tampon circulaire est caractérisé par :
- un indice de lecture : **read_index**,
- un indice d'écriture : **write_index**,
- une variable définissant le nombre de données présentes dans le tampon : **nb_token_available**
- une variable indiquant la nombre maximum de données stockables dans le tampon: **buffer_size**

Dans tout l'exercice nous utiliserons la structure suivante :

```
struct charFifo{
    char * buffer ;                  //la zone mémoire contenant les données du buffer, chaque donnée est un
                                     //char dans cet exemple
    unsigned int buffer_size;        //une variable définissant la taille du buffer
    unsigned int write_index ;       //l'index d'écriture dans le tableau de données
    unsigned int read_index ;        //l'index de lecture dans le tableau de données
    unsigned int nb_token_available ; //le nombre de données disponibles dans le buffer

};
```

Une variable de type **struct charFifo** doit être définie pour chaque Fifo manipulée par le programme, par exemple:

```
struct charFifo fifo1;
```

Attention, cette structure ne définit pas le tableau servant à stocker les données stocker dans la fifo mais uniquement la structure permettant de les manipuler. Un tableau de taille **FIFOSIZE1** caractères doit être reservé de la manière suivante:

```
#define FIFOSIZE1   3
char  fifoBuffer1[FIFOSIZE1];
```

2.1 Donner l'algorithme de la procédure d'initialisation du remplissage circulaire : **void fifo_init(struct charFifo * ptr_fif, char * ptr_buf, const unsigned int buf_size)** .
En entrée : **struct charFifo * ptr_fif** : pointeur sur la structure de file utilisée
char * ptr_buf : adresse de départ du tableau destiné à stocker les données de la fifo
const unsigned int buf_size : taille de la fifo

2.2 Dessiner l'espace mémoire occupé par **fifoBuffer1** si le tableau commence à l'adresse 4 et l'état des différents champs de la structure **fifo1** après appel de : **fifo_init(& fifo1, fifoBuffer1, FIFOSIZE1);**

2.3 Donner l'algorithme de la procédure de remplissage circulaire : **char fifo_write(struct charFifo * ptr_fif, const char token)** permettant de stocker une donnée dans le buffer circulaire. On admettra que si le tampon est plein, les données supplémentaires ne sont pas stockées et la fonction est non bloquante.

En entrée : **struct charFifo * ptr_fif**: pointeur sur la structure de file utilisée
const char token : valeur à écrire dans la file
En sortie : un caractère = 0 : valeur NON écrite dans la file
= 1 : valeur écrite dans la file

2.4 Donner l'algorithme de la procédure de vidage circulaire : **char fifo_read(struct charFifo * ptr_fif, char * ptr_token)** permettant de lire une donnée dans le buffer circulaire. La fonction est non bloquante.
En entrée : **struct charFifo * ptr_fif**: pointeur sur la structure de file utilisée
char * ptr_token : pointeur vers la variable dans laquelle ranger la donnée lue depuis la file.
En sortie : un caractère = 0 : valeur NON lue depuis la file (aucun caractère n'est disponible)
= 1 : valeur lue depuis la file via le paramètre *ptr_c.

2.5 En considérant la variable **fifo1** telle qu'initialisée à l'exercice 2.2, compléter les colonnes du tableau après l'exécution de chaque appel de fonction :

Appel	**retour**	**lu**	**write_index**	**read_index**	**nb_token_available**	**Case de buffer modifiée et valeur**
fifo_init(&fifo1, fifoBuffer1, FIFOSIZE1);			0	0	0	
fifo_read(&fifo1, &lu) ;						
fifo_write(&fifo1, 5) ;						
fifo_write(&fifo1, 8) ;						
fifo_read(&fifo1, &lu) ;						
fifo_write(&fifo1, 12) ;						
fifo_write(&fifo1, 1) ;						
fifo_write(&fifo1, 7) ;						
fifo_read(&fifo1, &lu) ;						
fifo_read(&fifo1, &lu) ;						
fifo_read(&fifo1, &lu) ;						
fifo_read(&fifo1, &lu) ;						

2.3 : Application à la gestion d'un buffer circulaire

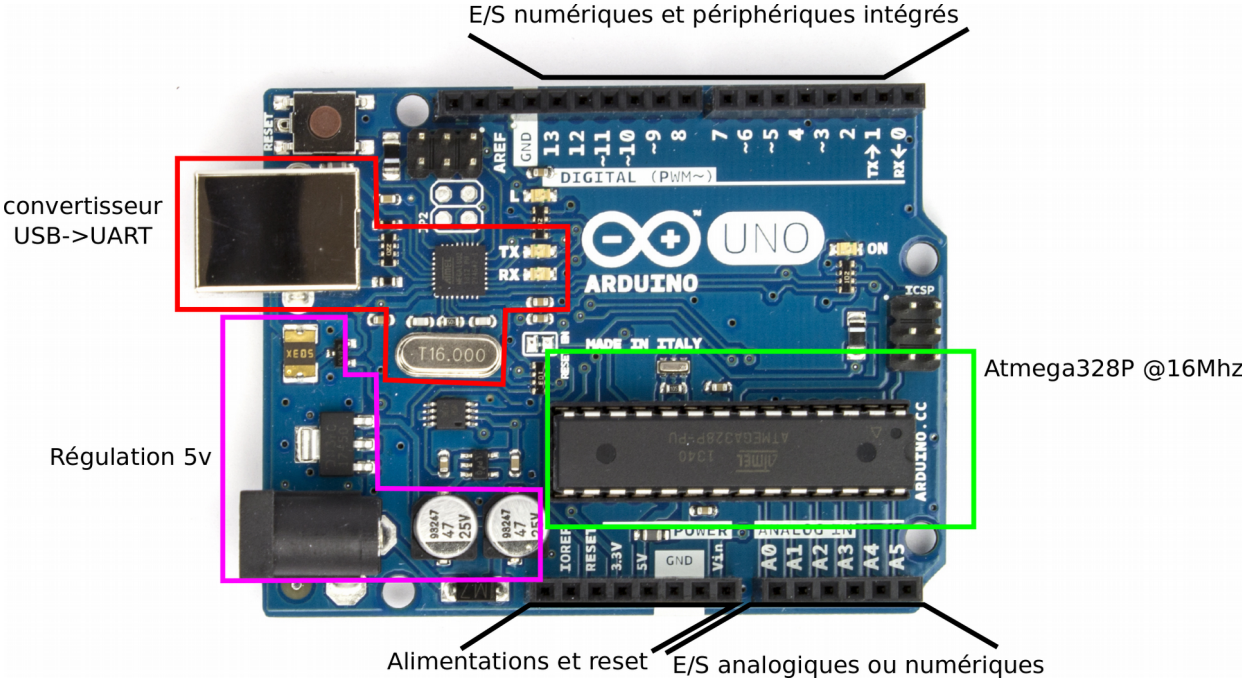
On considère une communication série 9600 bauds, 8 bits, 1 stop, pas de parité entre un ordinateur et une imprimante disposant d'un tampon circulaire de 256 octets. La vitesse d'impression est de 5 ms pour un caractère (la vitesse d'impression correspond à la vitesse de vidage du buffer).

2.3.1 En admettant que la taille du buffer circulaire de l'imprimante est de 256 octets, au bout de combien de temps le buffer sera-t-il plein si l'imprimante reçoit les caractères à vitesse maximale (sans pause)?
2.3.2 Même question si il y a 2 ms d'attente entre l'envoi de chaque caractère?
2.3.3 On cherche à imprimer une page d'un maximum de 3500 caractères sans provoquer de débordement du buffer circulaire. Calculer la taille minimum nécessaire du buffer circulaire si l'imprimante reçoit les caractères à vitesse maximale (sans pause)?

3. Présentation plateforme Arduino

Arduino est un plateforme de développement open-source qui se compose d'une carte microcontrôleur et d'un environnement de développement associé. La carte Arduino UNO se compose de :

- un microcontrôleur 8-bit Atmega328p cadencé à 16Mhz (exécutant une instruction par cycle d'horloge)
- un convertisseur USB/UART TTL
- une régulation de tension 5v
- un connecteur au pas de 2.54mm au standard Arduino



L'environnement logiciel est composé d'un éditeur/compilateur (Arduino IDE) et d'un ensemble de librairies pour contrôler les différents périphériques de la carte. De nombreuses librairies sont également proposées par la communauté de développeurs. Le langage de développement est basé sur C++. Le canevas d'un programme Arduino, se compose de deux fonctions principales à implémenter :

- **void setup()** : fonction d'initialisation appelée une fois au démarrage du microcontrôleur
- **void loop()** : fonction dont le corps est exécute en boucle (programme principal)

Les librairies exposent le plus souvent des classes permettant de voir les périphériques comme des objets manipulables au travers de leur méthodes. Par exemple l'interaction avec le port série de la carte se fait au travers de la classe **Serial** (classe statique) qui dispose des fonctions :

- **void begin(int baud, MODE)** : permettant d'initialiser le baudrate, la parité, le nombre de bits de stop
- **char read()** : permet de lire un octet depuis le port série
- **void write(char c)** : permet d'écrire un octet sur le port série
- **int available()** : permet de vérifier si un octet est disponible en lecture sur le port
- d'autres fonctions facilitant l'envoi et la réception de données (println)

L'utilisation des entrées/sorties numériques de la carte se fait au travers des fonctions :

- **void pinMode(int pin, MODE)** : configure la pin désignée en IN ou OUT
- **int digitalRead(int pin)** : renvoie la valeur binaire présente sur la pin désignée
- **void digitalWrite(int pin, VALUE)** : écrit la valeur LOW ou HIGH sur la pin designée

L'utilisation des entrées analogiques de la carte se fait au travers de la fonction :

- **AnalogRead(int pin)** : lit la valeur analogique présente sur la pin designée A0-5. La valeur retournée est entre 0-1023, un incrément de 1 correspondant à une valeur de 5/1024v soit 4.9mv.

D'autres classes (SPI, Wire) permettent d'interagir avec les bus SPI, I2C présents sur la carte

4. Utilisation de la liaison série “Hard” sur Arduino

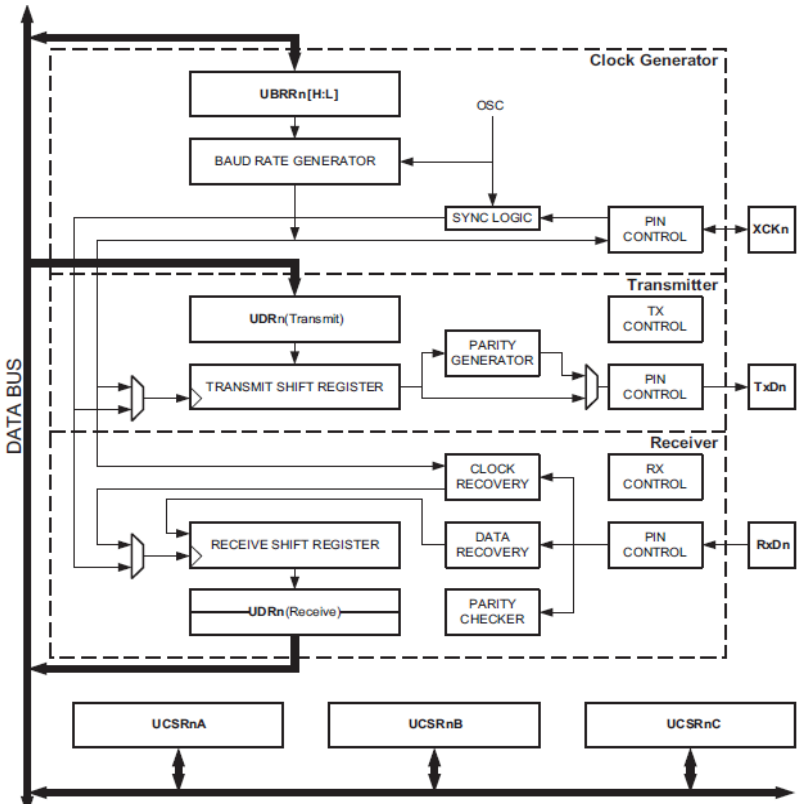
La class **Serial** regroupe l'intégralité des fonctions d'interaction avec l'UART de l'Atmega328p. A l'aide de la documentation, proposer une séquence en langage C qui réalise la fonction suivante :

1) réceptionne un caractère sur l'UART
2) si le caractère reçu est une lettre, le convertir en majuscule, sinon le laisser inchangé
3) envoyer le caractère calculé sur l'UART

5. Configuration d'un périphérique intégré

Le processeur de l'arduino dispose d'un périphérique UART intégré. La librairie Arduino Serial permet d'utiliser ce port à haut niveau

(pas d'interaction avec les registres du processeur) mais il est possible d'utiliser ce périphérique en configurant “à la main” les registres du processeur.



5.1 Configuration du baudrate

Le baudrate est configurable au travers du registre UBRR0 (UBRRn, n=0 pour l'USART 0) (voir tableau de baudrate)
La valeur à charger peut être établie par la formule :
UBRRn = (fosc/16)/baudrate

1) Pourquoi l'utilisation d'un oscillateur principal à 16Mhz ne permet pas d'obtenir un taux d'erreur de 0% ?
2) Proposer une fréquence d'oscillateur permettant un baudrate à 9600bds et un taux d'erreur de 0%
3) Dans le cas de l'utilisation d'un oscillateur à 16Mhz et d'un baudrate à 9600bauds, calculer la longueur de la séquence de bit avant qu'une erreur survienne (on considérera que l’échantillonnage du premier bit (le start) est réalisé au milieu de sa durée).

5.2 Configuration de la trame UART

Le registre USCrnC (USART Status and Control register C) permet de configurer le format de la trame UART (nombre de bits, parité, nombre de bits de stop).

1) Établir la configuration pour une communication 8N1 et proposer une séquence de configuration en langage C
2) Établir la configuration pour une communication 8E2 et proposer une séquence de configuration en langage C

5.3 Envoi et réception de données

L'envoi de données sur la liaison UART se fait par l'écriture dans le registre UDRn. Avant d'effectuer l'envoi d'une donnée, il faut d'abord vérifier que l'UART et disponible pour l'émission (dernière donnée envoyée) à l'aide du registre UCSRnA. Pour recevoir des données, il faut tout d'abord vérifier qu'une donnée est disponible à l'aide du même registre

1) Proposer une séquence en langage C permettant d'émettre un caractère sur l'UART
2) Proposer une séquence en langage C permettant de recevoir un caractère sur l'UART