TP 5 Utilisation du périphérique UART

Le but de ce TP est de réaliser un serveur UART “echo” : il ne fera que renvoyer chaque octet qu'on lui envoie, suivi de la chaîne ASCII `" mississippi, "`. Il sera branché au port série émulé intégré au *ST-Link* et parlera donc à votre ordinateur par USB. À partir d'aujourd'hui, nous utilisons la bibliothèque HAL pour s'abstraire des détails techniques de manipulation des registres. Il faudra donc l'inclure :

```
#include "stm32f3xx_hal.h"
```

et s'aider du code fourni dans `lib/HAL/stm32f3xx_hal_{uart,gpio}.{c,h}`.

1 Serveur echo

1. Initialisez le périphérique USART1, qui est connecté aux broches PC4 et PC5. Pour cela il faut :
   — activer l'horloge du périphérique USART1 ;
   — À l'aide de HAL, configurer les pins PC4 et PC5 sur leur fonction alternative avec “push-pull”, et leur vitesse maximum ; [1] trouver la valeur de la fonction alternative “USART1” dans le fichier `stm32f3xx_hal_gpio_ex.h`, à reporter dans le champ `Alternate` de la structure `GPIO_InitTypeDef`. Appelez `HAL_GPIO_Init()` dessus.
   — remplir une structure `UART_HandleTypeDef` avec les bons paramètres de configuration de l'UART (9600 Bauds, sans correction d'erreur, mode Tx/Rx) ; et appeler `HAL_UART_Init` dessus.
2. Dans la boucle principale, utilisez les fonctions `HAL_UART_Receive` et `HAL_UART_Transmit` pour qu'à chaque fois qu'un octet x soit reçu sur le port UART1, soit renvoyé le message “x `mississippi`” en ASCII. Pour débogguer plus facilement, vous pourrez faire clignoter une LED à chaque itération.
3. Localisez le port COM virtuel instancié par la carte (`/dev/tty.usbmodem`*XXXX* sur ma machine) et lancez un émulateur de terminal dessus. Par exemple avec le programme `screen`, fourni avec toute bonne distribution Linux [2] :

   ```
   $ screen /dev/tty.usbmodemFA1433 9600
   ```

   Tapez des caractères dans ce terminal, vous devriez recevoir une réponse de la carte. Utilisez l'oscilloscope pour inspecter les trames UART.
4. Changez la configuration de l'UART sur la carte (vitesse différente, activation de la correction d'erreur) mais pas sur votre terminal, et observez le résultat sur la qualité de la transmission.
5. Isolez dans un driver `computer_uart.c` des fonctions `UartInit()`, `UartReceive(char *msg, int size)` et `UartTransmit(char *msg, int size)`, spécifiques à cet UART.

1. Attention, il y a une erreur dans la documentation de la carte : elle indique les pins PA9 et PA10, ce n'est pas correct !
2. Pour quitter `screen` : `C-a \`.

2 Contrôle d'erreur

On s'intéresse maintenant à la conception d'un protocole de contrôle d'erreur implanté au dessus de UART. UART dans sa version la plus simple (pas de ligne d'horloge, sans bit de parité) n'est pas un protocole fiable : déphasage et dérive d'horloge peuvent entre autres corrompre le message transmis. Un protocole de contrôle d'erreur devra assurer avec une probabilité forte que le destinataire a bien reçu le message transmis par l'expéditeur, inaltéré. Pour cela, on imagine le protocole suivant :

1. l'émetteur envoit son message M (une suite de n octets) ;
2. immédiatement après réception d'un message M' (potentiellement différent de M puisque altéré) le récepteur envoit un message spécial d'acquittement d'un octet $ack = checksum(M')$;
3. l'émetteur reçoit ack et le compare avec $checksum(M)$;
4. si $ack = checksum(M)$ alors on considère que $M = M'$; l'émetteur envoit alors l'octet spécial "255" qui signifie "réception correcte".
5. sinon, il y a eu une erreur de réception : l'émetteur envoit l'octet spécial "0" qui signifie "erreur d'envoi, je vais réenvoyer le message", et on retourne à 1. pour retenter l'envoi du même message M.

Comment implémenter la fonction $checksum(\cdot)$, qui prend un message M de longueur arbitraire et retourne un octet $checksum(M)$? Voici une proposition : l'octet retourné est la somme de tous les octets de M, modulo 256.

Questions

1. Implémenter une fonction `checksum` qui prend une donnée et sa longueur en argument et renvoie son code correcteur sur 8 bits.
2. Implémenter deux fonctions :

   ```
   void SafeUartTransmit(char *msg, int size);
   void SafeUartReceive(char *msg, int size);
   ```

   qui réalisent notre protocole, de façon blocante. Ces fonctions utiliseront les fonctions `UartTransmit` et `UartReceive` écrite dans la première partie.
3. Pourquoi un tel protocole ne peut-il assurer avec certitude la bonne transmission ? Donner un exemple de faux positif, c'est-à-dire d'erreur de transmission non détectée. Comment amoindrir la probabilité de faux positifs ?