USRS26

Cours 4 HAL et communication série

Matthias Puech

Master 1 SEMS — Cnam

La librairie HAL

Communication série par UART

Périphériques U(S)ART sur STM32

La librairie HAL

Communication série par UART

Périphériques U(S)ART sur STM32

Présentation de HAL

Couche d'abstraction entre utilisateur et accès aux registres des périphériques (CMSIS)

+ plus besoin de se souvenir du nom/arrangement des registres
+ interface standardisée
– code potentiellement moins optimisé

Présentation de HAL

Couche d'abstraction entre utilisateur et accès aux registres des périphériques (CMSIS)

+ plus besoin de se souvenir du nom/arrangement des registres
+ interface standardisée
– code potentiellement moins optimisé

Organisation du code

- dans `lib/HAL/`
- paire de fichiers `.h` et `.c` pour chaque périphérique (ex : `stm32f3xx_hal_gpio.{c,h}`)
- structures de données dans les `.h`
- fonctions et documentation dans les `.c`

HAL Configuration des GPIOs

`lib/HAL/stm32f3xx_hal_gpio.{c,h}`

1. on déclare une structure `GPIO_InitTypeDef`
2. on remplit ses champs
   - `Pin` les pins à configurer (ORées) : `GPIO_PIN_0` … `GPIO_PIN_15`
   - `Mode` un des 4 modes pré-cités, et plus : `GPIO_MODE_INPUT`, `GPIO_MODE_OUTPUT_PP`, `GPIO_MODE_AF_PP`…
   - `Pull` `GPIO_NOPULL`, `GPIO_PULLUP`, `GPIO_PULLDOWN`
   - `Speed` `GPIO_SPEED_FREQ_LOW` … `GPIO_SPEED_FREQ_HIGH`
   - `Alternate` si `Mode == GPIO_MODE_AF_*`, determine la fonction alternative à connecter.
3. on la passe à la fonction `HAL_GPIO_Init()` avec le port à configurer (`GPIOA`, `GPIOB`…)

Le cas particulier de `Alternate`

Les fonctions alternatives de chaque pins sont nombreuses et dépendent de la référence exacte du MCU. Les valeurs que peut prendre `Alternate` sont listées dans :

`lib/HAL/stm32f3xx_hal_gpio_ex.h`

HAL Utilisation des GPIOs

`HAL_GPIO_ReadPin` renvoit SET ou RESET suivant l'état de la pin
`HAL_GPIO_WritePin` met une pin à SET ou RESET
`HAL_GPIO_TogglePin` inverse l'état d'une pin

Remarque

On peut agir sur plusieurs pins à la fois avec des OR

HAL Configuration des horloges de périphérique

`lib/HAL/stm32f3xx_hal_rcc.h`

Pour activer l'horloge de *PERIPH* :

Fonction `__HAL_RCC_`*PERIPH*`_CLK_ENABLE()`

Exemple

```
int main()
{
  /* LEDs initialization */
  RCC->AHBENR |= (1 << 21);   /* enable GPIO E clock */
  GPIOE->MODER |= 0x55550000; /* conf E8-E15 as output */

  while(1) {
    GPIOE->ODR ^= 0x0000FF00;   /* invert pin 8-15 */
    delay();
  }
}
```

Exemple

```
int main() {
  __HAL_RCC_GPIOE_CLK_ENABLE();
  GPIO_InitTypeDef  gpio_init;
  gpio_init.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10
    | GPIO_PIN_11 | GPIO_PIN_12 | GPIO_PIN_13
    | GPIO_PIN_14 | GPIO_PIN_15;
  gpio_init.Mode = GPIO_MODE_OUTPUT_PP;
  gpio_init.Pull = GPIO_PULLUP;
  gpio_init.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(GPIOE, &gpio_init);

  while (1) {
    HAL_GPIO_TogglePin(GPIOE, gpio_init.Pin);
    delay();
  }
}
```

La librairie HAL

Communication série par UART

Périphériques U(S)ART sur STM32

Taxonomie des protocoles de communication

Protocole langage de communication machine ⟷ machine.
(les *parties*)

Nature des données transportées

- signal continu analogique (ex : radio FM)
- flot d'octets arbitraires (ex : SPI)
- paquets d'octets arbitraires intermittents (ex : UART, IP)
- données structurées (ex : XML)

Support du protocole

- couche physique (ex : Ethernet/PHY, RS232)
- superposé à un autre protocole (ex : IP)

Taxonomie des protocoles de communication

Synchrone/Asynchrone

synchrone
: notion de temps commune à toutes les parties (horloge partagée, ex : USART, JTAG, I^2C)

asynchrone
: temps relatif à chaque partie (ex : UART)

Duplex

simplex
: émetteur → destinataire (ex : radio)

half-duplex
: "*à l'alternat*" (ex : talkie-walkie, JTAG)
  à un instant donné, partie 1 → partie 2 **ou** partie 2 → partie 1

full-duplex
: "bidirectionnel" (ex : UART)
  partie 1 → partie 2 **et** partie 2 → partie 1 en même temps

Taxonomie des protocoles de communication

Symétrie

client-serveur partie principale (serveur) qui répond aux requêtes des autres (ex : HTTP) ou qui initie la communication (ex : I^2C)

distribué chaque partie a le même rôle dans la communication (ex : BitTorrent, CAN)

Modalité

série les informations se succèdent sur une seule voie de communication (“une ligne”, ex : UART, USB)

parallèle les informations sont transmisent en même temps sur pluseurs voies (“plusieurs lignes” ; ex : CPU⟷RAM)

Taxonomie des protocoles de communication

Topologie

point-à-point 2 parties seulement (ex : UART, PPP)

étoile un “hub” central connecté à toutes les parties (ex : routeur sur un LAN)

daisy-chain n parties en chaîne, chacun fait passer le message au suivant jusqu’au destinataire (ex : SCSI, MIDI)

bus n parties, toutes connectées sur un seul “câble” (ex : CAN, PCI, les bus internes au STM32)

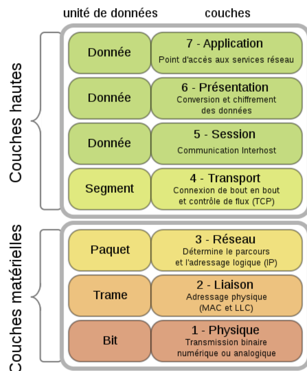
…

Piles de protocoles

Un “protocole” n’est qu’une vue de l’esprit, un moyen de comprendre une communication donnée.
Il n’apparaît que rarement isolé, mais repose sur d’autres protocoles plus élémentaires ⇝ *pile de protocoles*

Exemple

Le modèle OSI :



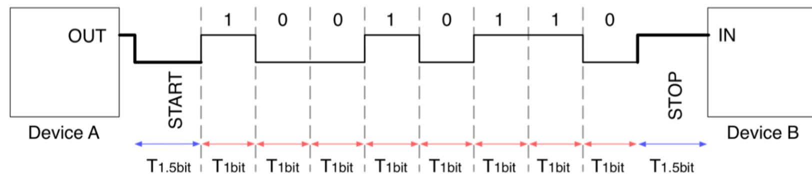
UART

“Universal Asynchronous Receiver Transmitter”

- Le “port série” des ordinateurs des années 90
- Protocole
  - full-duplex (2 lignes RX et TX)
  - point-à-point
  - asynchrone
  - série
  - de transmission de *trames* intermittentes
    (de 4 à 9 bits, suivant la configuration)
- De nombreux périphériques et puces supportent UART
- Adapté à la communication intra- et inter-carte (RS232)

Description du protocole UART

- deux lignes : RX (réception) et TX (transmission)
  (à brancher sur TX et RX de l'autre côté)
- pas d'horloge commune, mais une vitesse de transmission convenue des deux côtés, multiple de 9600 Hz (a.k.a. *Baud*).
  (appelons T la période d'horloge)
- pour chaque ligne RX et TX :
  - niveau haut au repos
  - *bas* pendant $1,5T \rightsquigarrow$ début de trame
  - puis chaque n bits tous les T
    (par défaut LSB : bits de poids faibles d'abord)
  - *haut* pendant $1,5T \rightsquigarrow$ fin de trame
    (en fait, temps configurable : 0.5, 1, 1.5 ou 2T)



Description du protocole UART

Addendum

Il existe une trame spéciale, qui peut être reçue/envoyée :

- *Break* : trame de 0
  (utilisé pour indiquer un “temps mort” dans l’émission)

Description du protocole UART

Addendum

Il existe une trame spéciale, qui peut être reçue/envoyée :

- *Break* : trame de 0
  (utilisé pour indiquer un “temps mort” dans l’émission)

Le problème avec UART

Les horloges des deux parties ne sont pas synchronisées
⇝ *drift* (désynchronisation)
⇝ erreurs possibles de transmission

Solutions

sur-échantillonnage le receveur échantillonne le signal entrant tous les T/n pour connaître sa phase

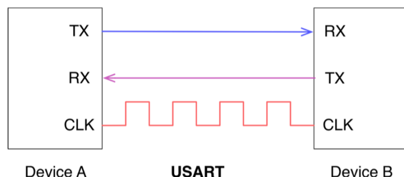
correction d’erreur un bit supplémentaire de parité est transmis

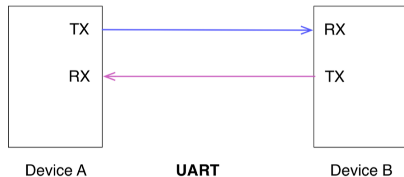
ajout d’une ligne d’horloge commune ⇝ USART

Extensions du protocole UART

U**S**ART

On rend le protocole synchrone en ajoutant une ligne d'horloge, contrôlé par une des parties





Extensions du protocole UART

Correction d'erreur

- on ajoute 1 bit $B[n]$ à chaque trame
- $B[n] = \sum_{i=0}^{n-1} B[i] \mod 2$
- À réception, si $B[n]$ n'a pas cette valeur, la trame est erronée.
- Si $B[n]$ a la bonne valeur… nombre pair d'erreurs !
  (mais la probabilité de 2 erreurs est faible)

$\leadsto$ nécessité d'un protocole superposé d'acquittement
(parfois, le receveur va accuser réception ou demander le renvoi)

Hardware Flow Control

- deux lignes supplémentaires de contrôle de flot :
- *Request to Send* (RTS) haut quand l'émetteur est prêt à émettre
- *Clear to Send* (CTS) haut quand le récepteur est prêt à recevoir
- alternative à la ligne d'horloge

RS232

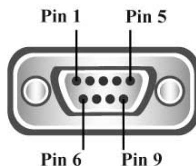
Une couche physique pour U(S)ART :

- standardise les voltages
- le connecteur et le brochage (*pinout*)

RS232

Pin 1	DCD
Pin 2	RXD
Pin 3	TXD
Pin 4	DTR
Pin 5	GND
Pin 6	DSR
Pin 7	RTS
Pin 8	CTS
Pin 9	RI

RS232 Pinout (9 Pin Male)



Autre exemple

Pont USB/USART (FT232RL / Virtual COM Port)

Sur notre carte

Deux options :

- coder le protocole nous même :
    - utiliser les GPIOs pour mettre deux pins on/off
    - implémenter l'attente de 1/9600 par une boucle
    - implémenter fonctions `transmit(char c)` et `char receive()` en suivant le protocole.

Sur notre carte

Deux options :

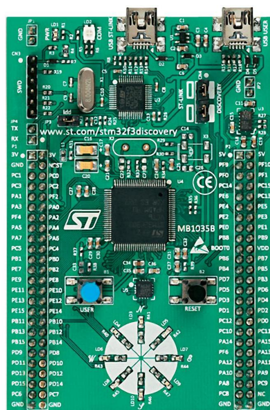
- coder le protocole nous même :
  - utiliser les GPIOs pour mettre deux pins on/off
  - implémenter l'attente de 1/9600 par une boucle
  - implémenter fonctions `transmit(char c)` et `char receive()` en suivant le protocole.
- utiliser un des périphériques UART embarqué sur le MCU :
  - \+ protocole implémenté en hardware
  - \+ le processeur est déchargé du travail de transmission (contrôle d'erreur,)
  - \+ on communique avec par lecture/écriture dans des registres (comme d'habitude)
  - – le périphérique est relié à des pins prédéfinies (voir specsheet)

Sur notre carte

3 périphériques USART (USART1..3)
2 périphériques UART (UART4,5)

USART1
- connecté à PA4/PA5
- connecté à RX/TX
- connecté au ST-Link
- transmis à travers l'USB
  - (port COM virtuel)

⇝ communication facilitée entre ordinateur et MCU



HAL Configuration des USARTs

`lib/HAL/stm32f3xx_hal_usart.{c,h}`

Périphériques USART1…3

Chacun a une dizaine de registres
On va les manipuler à haut niveau, grâce à HAL :

- on consulte la datasheet pour savoir à quelle pins correspondent les E/S
- on configure le GPIO des pins correspondantes : *Alternate Function* USART*n*.
- on déclare et remplit une *handle* `USART_HandleTypeDef` :
  - `Instance` = USART1…3
  - `Init.BaudRate` = vitesse de communication (en Hz)
  - `Init.Mode` = Rx, Tx ou Rx/Tx
  - `Init.OverSampling` = réduit les chances d'erreurs ;
- on la passe à la fonction `HAL_USART_Init`

HAL Utilisation des USARTs

`lib/HAL/stm32f3xx_hal_usart.{c,h}`

Handle

Structure qui contient l'état et la configuration actuelle du périphérique, à passer avec toute communication. (ex : instance, taille de dernière donnée reçue…)

HAL Utilisation des USARTs

`lib/HAL/stm32f3xx_hal_usart.{c,h}`

Handle

Structure qui contient l'état et la configuration actuelle du périphérique, à passer avec toute communication. (ex : instance, taille de dernière donnée reçue…)

API synchrone/blocante

Prennent en argument le *handle*, la donnée, sa taille, et un *timeout*.

`HAL_USART_Transmit()` demande au périphérique d'envoyer un tableau d'octets, et boucle jusqu'à ce que ce soit fait.

`HAL_USART_Receive()` boucle et surveille le périphérique jusqu'à ce qu'un message y soit reçu ; copie ce message dans un tableau.

Problème de l'API synchrone

Exemple

```
int data;
while(1) {
  HAL_USART_Receive(&UsartHandle, &data, 4, 100);
  data = LongComputation(data);
  HAL_USART_Transmit(&UsartHandle, &data, 4, 100);
  blink();
}
```

Problèmes

- requêtes potentiellement perdues
- tâche "parallèle" impossible

Problème de l'API synchrone

Exemple

```
int data;
while(1) {
  HAL_USART_Receive(&UsartHandle, &data, 4, 100);
  data = LongComputation(data);
  HAL_USART_Transmit(&UsartHandle, &data, 4, 100);
  blink();
}
```

Problèmes

- requêtes potentiellement perdues
- tâche "parallèle" impossible

$\rightsquigarrow$ *interruptions*

La librairie HAL

Communication série par UART

Périphériques U(S)ART sur STM32

Les périphériques U(S)ART

Au lieu d'utiliser HAL, on pourrait configurer et utiliser le périphérique “à la main” : (voir manuel de référence pp. 885–951)

Registres de configuration

CR1,2,3 *configuration register* (lecture/écriture)
Configuration du périphérique :

- nombre de bits par trame
- bit de parité
- transmission/réception activée
- ...

BRR *baud rate register* (lecture/écriture)
Fréquence de transmission

Les périphériques U(S)ART

Registres de donnée et d'état

TDR *transmit data register* (écriture)
la donnée à transmettre (TX)

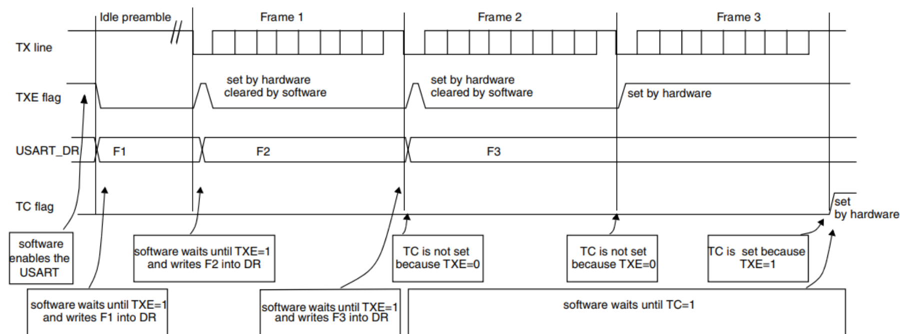
RDR *receive data register* (lecture)
la dernière donnée reçue (RX)

ISR *interrupt and status register* (lecture)
Flags (bits) décrivant l'état courant du périphérique :

- **bit TXE** indique qu'on peut écrire dans TDR (prêt à envoyer). Une écriture dans TDR met TXE à 0.
- **bit RXNE** indique qu'une donnée est prête à être lue dans RDR. Une lecture dans RDR met RXNE à 0.
- **bit TC** transmission complete

Les périphériques U(S)ART

Diagramme de communication en émission



Les périphériques U(S)ART

Pour envoyer des trames

1. configurer CR1,2,3 et BRR
   (suivant les caractéristiques de la ligne)
2. attendre que TXE soit à 1
   (prêt à envoyer)
3. placer les N bits à envoyer dans TDR
4. TXE se met à 0
   (en cours d'envoi)
5. s'il reste des trames à envoyer, GOTO 2.
6. attendre que TC soit à 1
   (envoi terminé)

Les périphériques U(S)ART

Pour envoyer des trames

1. configurer CR1,2,3 et BRR
   (suivant les caractéristiques de la ligne)
2. attendre que TXE soit à 1
   (prêt à envoyer)
3. placer les N bits à envoyer dans TDR
4. TXE se met à 0
   (en cours d'envoi)
5. s'il reste des trames à envoyer, GOTO 2.
6. attendre que TC soit à 1
   (envoi terminé)

(*cf.* `stm32f3xx_hal_uart.c`, fonction `HAL_UART_Transmit()`)

Les périphériques U(S)ART

Pour recevoir des trames

1. configurer CR1,2,3 et BRR
   (suivant les caractéristiques de la ligne)
2. mettre le bit RE de CR1 à 1
   (*receiver enable*, active l'écoute d'une condition start)
3. attendre que RXNE soit à 1
   (données reçues, disponibles dans RDR)
4. lire les N bits reçus dans RDR
5. RXNE se met à 0
   (en attente du prochain message)
6. s'il reste des trames à recevoir, GOTO 3.
7. si données arrivent quand RXNE est à 1, on a un *overrun*
   (on n'a pas lu les données assez vite)

Les périphériques U(S)ART

Pour recevoir des trames

1. configurer CR1,2,3 et BRR
   (suivant les caractéristiques de la ligne)
2. mettre le bit RE de CR1 à 1
   (*receiver enable*, active l'écoute d'une condition start)
3. attendre que RXNE soit à 1
   (données reçues, disponibles dans RDR)
4. lire les N bits reçus dans RDR
5. RXNE se met à 0
   (en attente du prochain message)
6. s'il reste des trames à recevoir, GOTO 3.
7. si données arrivent quand RXNE est à 1, on a un *overrun*
   (on n'a pas lu les données assez vite)

(*cf.* `stm32f3xx_hal_uart.c`, fonction `HAL_UART_Receive()`)

Résumé

- les parties doivent s'accorder sur la configuration des lignes (*baud rate*, longueur de trame, bit de parité…)
- communication série et intermittente (quand pas de trames transmises, ligne haute)
- dans l'API HAL vue, on fait de l'attente active (on boucle en attendant la levée d'un flag)

Addendum Utiliser `scanf()` et `printf()`

Nous n'utilisons par défaut aucune librairie standard ("libC") :

- pas de `scanf()`, `printf()`...
- pas de `malloc()`...

Mais `gcc` embarque une implémentation "par défaut" de certaines primitives, dont `scanf()` et `printf()`, sous réserve de définition des "appels systèmes" sous-jacents.

Addendum Utiliser `scanf()` et `printf()`

Nous n'utilisons par défaut aucune librairie standard ("libC") :

- pas de `scanf()`, `printf()`...
- pas de `malloc()`...

Mais `gcc` embarque une implémentation "par défaut" de certaines primitives, dont `scanf()` et `printf()`, sous réserve de définition des "appels systèmes" sous-jacents.

Rediriger les entrées/sorties vers l'UART

Utile pour débugger !
Il suffit d'implémenter les "appels systèmes" qui font effectivement les lectures et écritures

Addendum Utiliser `scanf()` et `printf()`

Comment faire

- rajouter l'option `-specs=nosys.specs` aux `ARCHFLAGS` (force `gcc` à ne pas include de libC)
- définir les fonctions de bas niveau ("appels système")

```
int _read(int file, char *data, int len)
int _write(int file, char *data, int len)
```

`int file` le descripteur de fichier (ne pas prendre en compte cet argument)

`char *data` les caractères à lire/écrire

`int len` le nombre de caractères à lire/écrire

valeur de retour le nombre de caractères effectivement lus/écrits