USRS26
Cours 5 Interruptions

Matthias Puech

Master 1 SEMS — Cnam

Modèle de machine asynchrone

Interruptions sur STM32

Configuration avec HAL

Évenements asynchrones

Un rôle principal des MCUs et de *réagir* à des évenements externes

Exemple

- appui sur un bouton
- déclenchement d'un capteur
- demande d'information d'un actuateur
- arrivée d'un message sur un canal de communication
- horloge temps réel (timer) : "1 ms s'est écoulé"

Évenements asynchrones

Un rôle principal des MCUs et de *réagir* à des évenements externes

Exemple

- appui sur un bouton
- déclenchement d'un capteur
- demande d'information d'un actuateur
- arrivée d'un message sur un canal de communication
- horloge temps réel (timer) : "1 ms s'est écoulé"

Asynchrone

qui intervient hors du cadre temporel par défaut
(ex : "tâche de fond", tâche périodique)

Les interruptions

Mécanisme matériel qui permet d'interrompre le cours de l'exécution à l'arrivée d'un évenement, d'exécuter du code dédié puis de reprendre le cours de l'exécution.

Interruptions vs. exceptions

Chez ARM,

- une *interruption* est spécifiquement liée à un évenement matériel,
  (ex : arrivée d'un message UART, front montant sur GPIO)
- une *exception* peut être liée à un évenement logiciel
  (ex : appel système, division par zéro, erreur d'accès mémoire)

Polling vs. interruptions

Deux “styles” de programmation réactive à des évenements :

- en *polling*, le code consulte régulièrement un état (registre) ; action spécifique en cas d’évenement (conditionnelle)
  - le temps de réaction dépend de la fréquence de consultation
  - chaque consultation prend du temps processeur (*attente active*)
- en *interruption*, on laisse la consultation à un bout de hardware, qui nous “prévient” en cas d’évenement (saut)
  - hardware = contrôleur d’interruptions
  - temps de réaction court (dépend du hardware)
  - pas de consultation régulière par le processeur ($\rightsquigarrow$ plus de temps pour le reste)
  - un nouveau modèle de programmation (*programmation réactive*)

Application : multitâche

Une interruption déclenchée périodiquement (ex : 1ms) par une horloge (timer) permet d'implémenter le *multitâche préemptif* :

- chaque ms, on interromp la tâche en cours, on sauvegarde son état et on bascule sur la suivante
- on entrelace les tâches pour donner l'illusion du multitâche
- impossible sans interruptions
  (ou multitâche collaboratif, e.g. Windows 3.1, MacOS Classic)

Modèle d'exécution

La machine à registre

Jusqu'à maintenant, on a vu que le processeur ne faisait que :

1. charger l'instruction contenue dans le PC
2. la décoder, et incrémente PC
3. l'exécuter, ce qui modifie les registres et/ou la mémoire
4. GOTO 1.

Modèle d'exécution

La machine à registre

Jusqu'à maintenant, on a vu que le processeur ne faisait que :

1. charger l'instruction contenue dans le PC
2. la décoder, et incrémente PC
3. l'exécuter, ce qui modifie les registres et/ou la mémoire
4. GOTO 1.

Ajoutons les interruptions

0. si une interruption *i* est levée par le contrôleur :
   - 0.1 sauver le *contexte* courant
   - 0.2 lire dans le *vecteur d'interruption* l'adresse stockée à la case *i*
   - 0.3 écrire dans PC cette adresse

3.5. si on vient de sortir d'une interruption :

- 3.5.1. restaurer le *contexte* sauvé

Modèle de machine asynchrone

Interruptions sur STM32

Configuration avec HAL

Le *vecteur d'interruption*

Un tableau V à une adresse fixe en mémoire, qui stocke les adresses de saut en cas d'arrivée d'une interruption i.

Exemple

- dans les Cortex-M4, V est stocké à l'adresse 0x4.
- l'interruption 0 s'appelle `Reset`
  (appui sur bouton, reset software, allumage)
- $V[0]$ contient l'adresse du code où sauter en cas de *reset*

Le *contexte*

8 registres, dont PC, forment le *contexte* : ils sont sauvegardés à l'entrée dans une exception, et restaurés à sa sortie :

R0…R3, R12, LR (*link register*) et PC

Le *contexte*

8 registres, dont PC, forment le *contexte* : ils sont sauvegardés à l'entrée dans une exception, et restaurés à sa sortie :

R0…R3, R12, LR (*link register*) et PC

Sauver le contexte

En cas d'arrivée de l'interruption i :

1. écrire les 8 valeurs des registres du contexte aux adresses $SP, $SP+1, …$SP+7
   (rappel : SP contient l'adresse du haut de la pile)
2. écrire l'adresse contenue dans $V[i]$ dans PC
3. $SP = $SP + 8

Restaurer le contexte

À la fin de l'execution du code d'une interruption :

1. $SP = $SP − 8
2. copier dans les registres les valeurs $SP, $SP+1, …$SP+7

Le *contexte*

Deux “modes” processeur :

handler mode actif quand un handler est exécuté
(activé au moment où une interruption est levée)

thread mode actif quand on est sorti d’un handler
(instruction `ret` en mode handler ;
c’est le mode par défaut)

Context Switch

transition thread → handler sauve le contexte

transition handler → thread restaure le contexte

Le *contexte*

Deux “modes” processeur :

handler mode actif quand un handler est exécuté
(activé au moment où une interruption est levée)

thread mode actif quand on est sorti d’un handler
(instruction `ret` en mode handler ;
c’est le mode par défaut)

Context Switch

transition thread → handler sauve le contexte

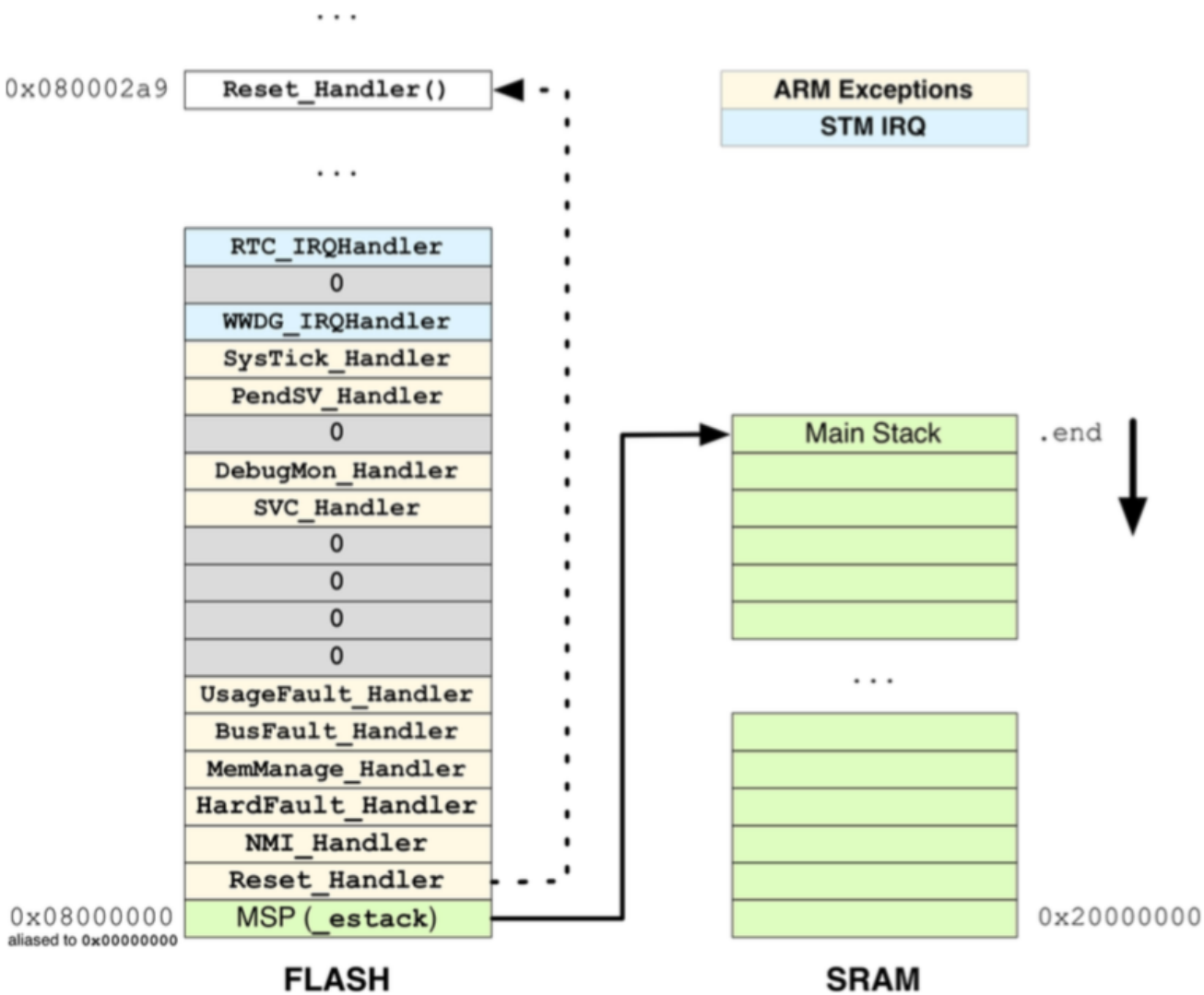
transition handler → thread restaure le contexte

⇝ entrer/sortir d’une interruption prend du temps !
(12/15 cycles CPU)

Interruptions sur Cortex-M4

- le contrôleur d'interruption est un périphérique appelé le *Nested Vectored Interrupt Controller (NVIC)*
- de 32 à 256 sources possibles d'interruption
  (dépend des périphériques installés, voir specsheet)
- le vecteur d'interruption est stocké par défaut à partir de l'adresse 0x4
  (aliasée au début de la mémoire flash par défaut)
- il est placé en début de flash par le *linker script*
- vous programmeurs devez mettre les adresses des bonnes routines dans ce vecteur d'interruption
- la case 0x0 contient l'adresse de début de la pile, copié dans SP au démarrage.
  (aussi appelé MSP ou `_estack`, voir `startup_stm32f303xc.s`)

Interruptions sur Cortex-M4



Quelques sources d'interruptions

`Reset` allumage et au relâchement du bouton Reset

`Hard Fault` division par zéro, mauvaise configuration des registres de périphériques

`Bus Fault` accès illégal en mémoire
(adresse inexistante ou en lecture seule)

`Debug Monitor` breakpoint/watchpoint atteint

`SysTick` 1 ms vient de s'écouler

`PERIPH_IRQ` interruption déclenchée par *PERIPH*. Ex :

- `USART1_IRQ` message reçu/envoyé par USART1
- `ADC1_2_IRQ` conversion finie sur l'ADC1 ou 2

Quelques sources d'interruptions

`Reset`	allumage et au relâchement du bouton Reset
`Hard Fault`	division par zéro, mauvaise configuration des registres de périphériques
`Bus Fault`	accès illégal en mémoire (adresse inexistante ou en lecture seule)
`Debug Monitor`	breakpoint/watchpoint atteint
`SysTick`	1 ms vient de s'écouler
PERIPH`_IRQ`	interruption déclenchée par *PERIPH*. Ex :
`USART1_IRQ`	message reçu/envoyé par USART1
`ADC1_2_IRQ`	conversion finie sur l'ADC1 ou 2

- noms des `Handlers` réservés (weak) dans `lib/CMSIS/startup_stm32f303xc.s`
- si on nomme ainsi une fonction à nous, le handler pointera dessus

Architecture matérielle

Vocabulaire

NVIC contrôleur d'interruption (périphérique hardware)

ISR (Interrupt Service Routine) ou Handler code gestionnaire d'une interruption

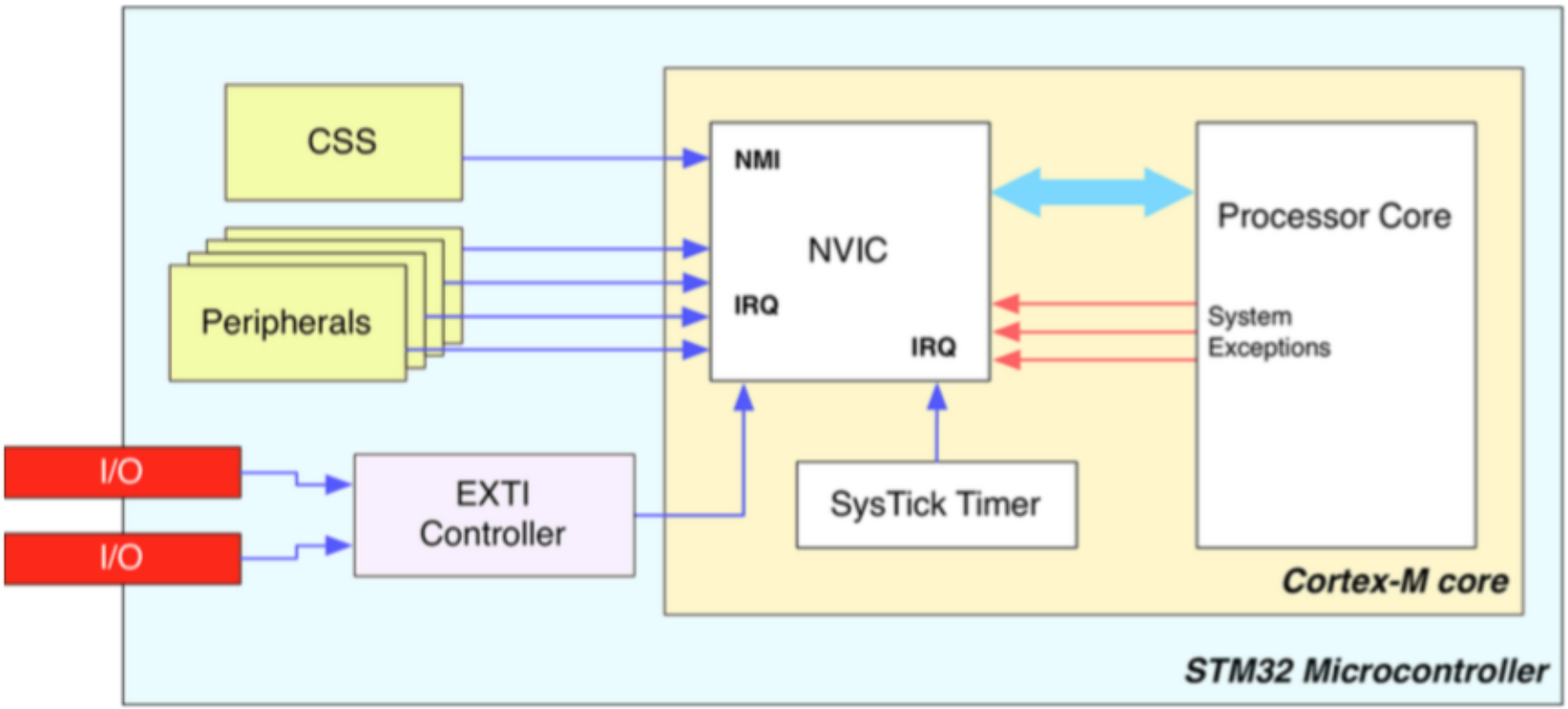
Vecteur d'Interruption tableau contenant les adresses des handlers pour chaque source d'interruption

IRQ fil matériel qui déclenche une interruption

EXTI périphérique de détection de front montants/descendants sur les GPIOs

SysTick timer qui déclenche une interruption toutes les millisecondes

Architecture matérielle



Priorités et préemption

Que se passe-t-il si une interruption B arrive
pendant l'exécution du handler d'une autre A ?

Priorités et préemption

Que se passe-t-il si une interruption *B* arrive
pendant l'exécution du handler d'une autre A ?

Priorités

A chaque interruption est assigné :

- une *priorité* (15 niveaux, nombre bas=priorité élevée)
- une *sous-priorité* (?)

Exemple

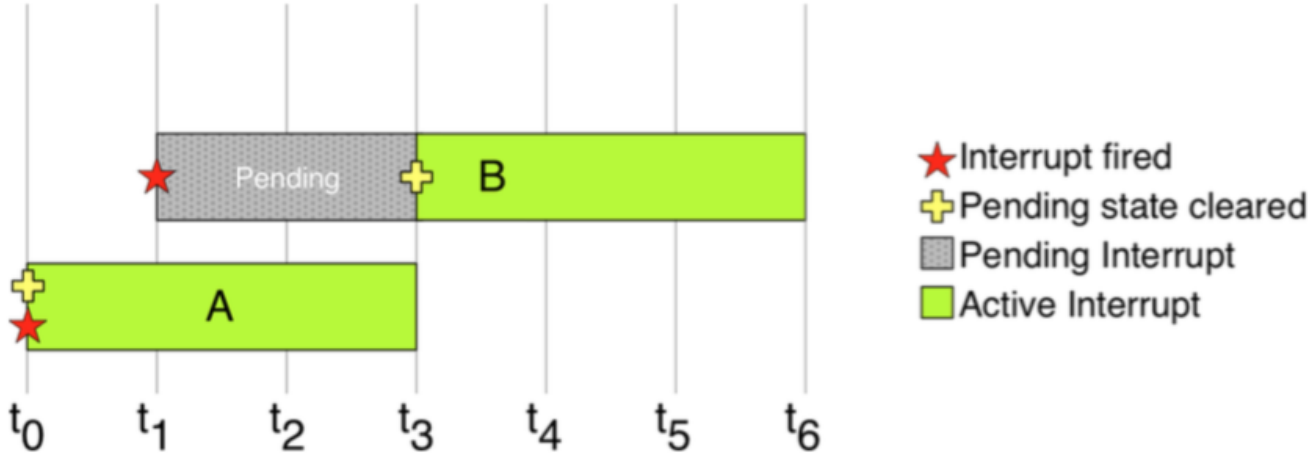
`Reset` priorité -3 (maximum)

`Hard Fault` priorité -1

les autres configurable

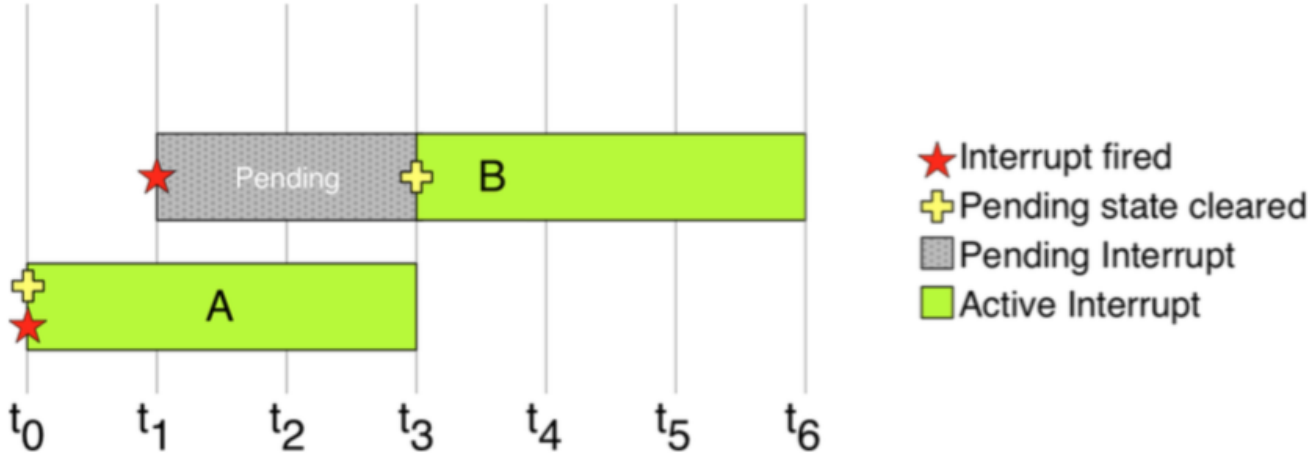
Priorités et préemption

- Si B est moins prioritaire que A, elle est mise en attente :

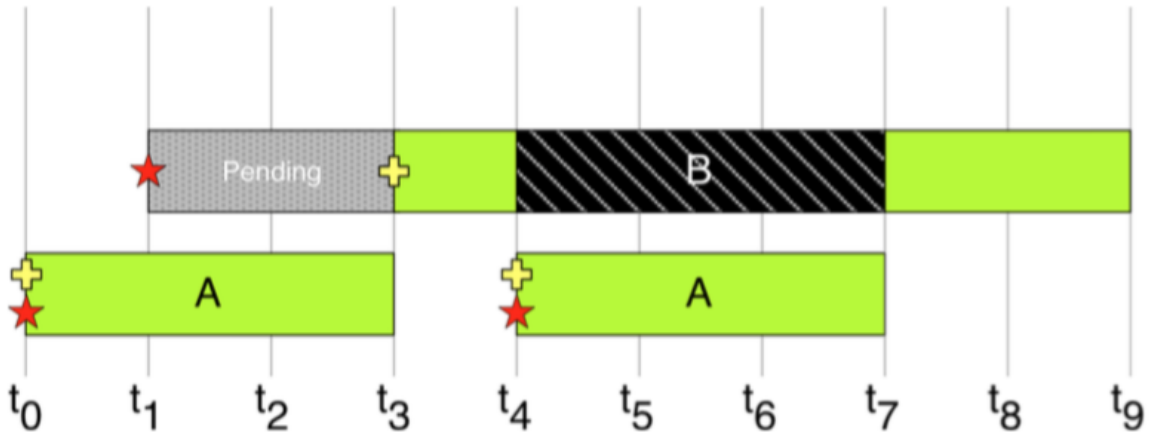


Priorités et préemption

- Si B est moins prioritaire que A, elle est mise en attente :



- Si A est redéclenchée pendant B, B est préemptée :



Priorités et préemption

Remarques

- une interruption STM32 ne peut pas être préemptée par elle-même (pas *ré-entrantes*)
- au démarrage, une interruption `Reset` est déclenchée

La boucle principale, revisitée

```
void main() {
  init();
  while(1) { }
}
void SysTick_Handler() {
  blink_led();
}
```

La boucle principale, revisitée

```
void main() {
  init();
  while(1) { __WFI(); }
}
void SysTick_Handler() {
  blink_led();
}
```

`__WFI();` Wait For Interrupt

Instruction de mise en sommeil du coeur.
Sera réveillé à la prochaine interruption.

Modèle de machine asynchrone

Interruptions sur STM32

Configuration avec HAL

SysTick

- un timer simple commun à tous les Cortex-M
- déclenche interruption toutes les millisecondes
  (période configurable, mais ça n'est pas recommandé)
- utilisé dans les OS pour ordonnancer les tâches

SysTick

- un timer simple commun à tous les Cortex-M
- déclenche interruption toutes les millisecondes
  (période configurable, mais ça n'est pas recommandé)
- utilisé dans les OS pour ordonnancer les tâches

HAL Activer SysTick

Dans `stm32f3xx_hal.c`

- définir fonction `void SysTick_Handler()`
- appeler `HAL_Init()` :
  - configure le NVIC
  - configure SysTick à 1ms
  - active l'IRQ de SysTick

SysTick

- un timer simple commun à tous les Cortex-M
- déclenche interruption toutes les millisecondes
  (période configurable, mais ça n'est pas recommandé)
- utilisé dans les OS pour ordonnancer les tâches

HAL Activer SysTick

Dans `stm32f3xx_hal.c`

- définir fonction `void SysTick_Handler()`
- appeler `HAL_Init()` :
  - configure le NVIC
  - configure SysTick à 1ms
  - active l'IRQ de SysTick
- *(optionnel)* dans `SysTick_Handler()`, appeler `HAL_IncTick()`. On peut alors utiliser :

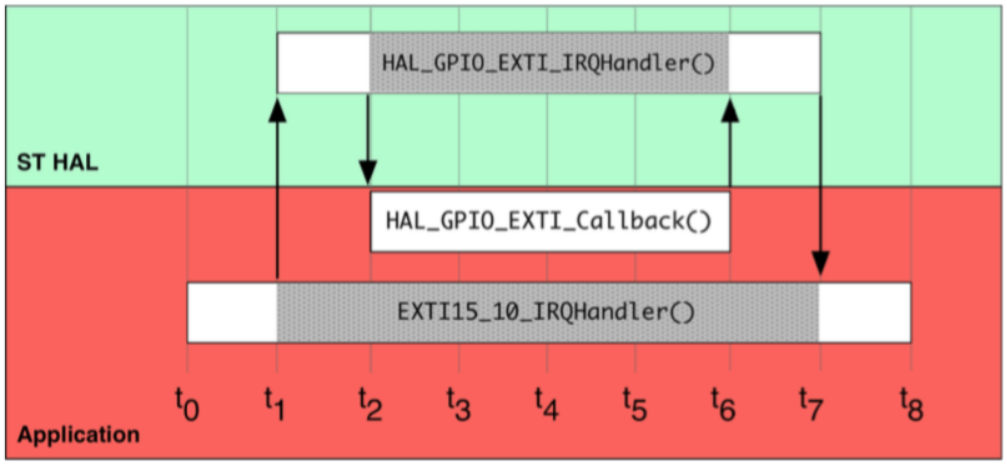
  `HAL_GetTick` retourne nombre de ms écoulées
  `HAL_Delay` attente active pendant n ms

HAL Activation d'interruptions

En général

Pour un périphérique donné :

- configurer le périphérique en mode interruption si nécessaire
- activer l'interruption dans le NVIC
- dans le *handler*, appeler le *handler* HAL correspondant
  (fait le travail de bas niveau lié à l'interruption)
- définir le(s) *callbacks*, qui sont appelés par HAL



HAL Activation d'interruptions

Configurer le périphérique en mode interruption

Dépend du périphérique
(voir fichier source HAL correspondant)

Activer une interruption

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn)
```

(`IRQn_Type` dépend du MCU utilisé ;
défini dans `lib/CMSIS/stm32f303xc.h`)

Exemple : périphérique UART

- on active l'IRQ :

  ```
  HAL_NVIC_EnableIRQ(USART1_IRQn);
  ```

- on définit

  ```
  void USART1_IRQHandler() {
    HAL_UART_IRQHandler(...);
  }
  ```

- pour envoyer/recevoir, on utilise les fonctions

  ```
  HAL_UART_Receive_IT();
  HAL_UART_Transmit_IT();
  ```

- à la fin de la réception/l'émission, `HAL_UART_IRQHandler` appelle le bon *callback* parmi :
  - `HAL_UART_ErrorCallback()` si erreur de transfert
  - `HAL_UART_TxCpltCallback()` après transfert ok
  - `HAL_UART_ErrorCallback()` après réception ok
  - ...(voir `stm32f3xx_hal_uart.c`)

Périphériques U(S)ART et interruptions

On a décrit la dernière fois le processus d'envoi/réception blocant ; voyons comment fonctionnent cette API non blocante.

Registres de configuration

CR1 contient les bits d'"armement" des interruptions ("*interrupt enable*" ou IE) :

- bit TXEIE : si 1 et TXE=1, produit interruption (prêt à envoyer)
- bit TCIE : si 1 et TC=1, produit interruption (émission terminée)
- bit RXNEIE : si 1 et RXNE=1, produit interruption (réception effectuée)

Périphériques U(S)ART et interruptions

On a décrit la dernière fois le processus d'envoi/réception blocant ; voyons comment fonctionnent cette API non blocante.

Registres de configuration

CR1 contient les bits d'"armement" des interruptions ("*interrupt enable*" ou IE) :

- bit TXEIE : si 1 et TXE=1, produit interruption (prêt à envoyer)
- bit TCIE : si 1 et TC=1, produit interruption (émission terminée)
- bit RXNEIE : si 1 et RXNE=1, produit interruption (réception effectuée)

Attention

Une interruption est émise tant que le flag est à 1 ;
on doit donc le mettre à 0 dès qu'on a traité l'interruption ("désarmer l'interruption")

Périphériques U(S)ART et interruptions

Pour recevoir une trame

1. configurer CR1,2,3 et BRR
   (suivant les caractéristiques de la ligne)
2. mettre le bit RXNEIE à 1
   (déclenchera une interruption quand reçoit une trame)
3. dans le handler de l'interruption :
   - si RXNE=1 : (message reçu)
     3.1 lire la donnée depuis RDR (met automatiquement RXNE à 0)
     3.2 appeler un éventuel *callback*
     (action à faire après la réception)

Périphériques U(S)ART et interruptions

Pour envoyer une trame

1. configurer CR1,2,3 et BRR
   (suivant les caractéristiques de la ligne)
2. mettre le bit TXEIE à 1
   (déclenchera une interruption quand prêt à émettre)
3. dans le handler de l'interruption :
   - si TXE=1 : (prêt à émettre)
     3.1 écrire la donnée dans TDR (met automatiquement TXE à 0)
     3.2 mettre le bit TCIE à 1
   - si TC=1 : (transfer complete)
     3.1 mettre TCIE à 0 (“désarme”)
     3.2 appeler un éventuel *callback*
     (action à faire après le transfert)

Périphériques U(S)ART et interruptions

Pour envoyer ou recevoir plusieurs trames

cf. `stm32f3xx_hal_uart.c` :

- fonctions `HAL_UART_{Receive|Transmit}_IT()`
  (arment les interruptions TXE et RXNE, et enregistrent le buffer de donnée et sa taille)
- fonction `HAL_UART_IRQHandler()`
  (dispatche les évenements aux fonctions spécialisées :)
  - `UART_Receive_IT()` en cas de réception
    (stocke l'octet reçu, et appelle le *callback* si c'était la dernière attendue)
  - `UART_Transmit_IT()` en cas de transmission réussie
    (lit l'octet suivant à envoyer, ou appelle le *callback* si c'était le dernier à envoyer)
- fonctions `HAL_UART_*Callback`
  (prototypes *weak* ne faisant rien, à redéfinir)