Révision Système À Microcontrôleurs

FISE S5

ATHENA (Alain PIALLAT)

17 Novembre 2025

BDTech

Question 1

Quelle configuration GPIO permet de lire un interrupteur qui est connecté entre la pin et la tension d'alimentation ?

A. OUTPUT PUSH-PULL
B. INPUT with PULL-DOWN
C. INPUT with PULL-UP
D. ANALOG INPUT
E. Aucune de ces réponses

BDTech

Réponse Question 1

Réponse correcte : B. INPUT with PULL-DOWN

Explication :

- Une configuration INPUT with PULL-DOWN active une résistance interne qui tire la pin vers GND, lorsque l'interrupteur est ouvert, la pin est donc à LOW (0V).
- **PULL-UP** fait l'inverse : il tire la pin vers VCC il sert à lire des interrupteurs connectés entre la pin et GND.
- **OUTPUT PUSH-PULL** est une configuration de sortie, pas d'entrée qui permet de fournir un niveau HIGH ou LOW.
- **ANALOG INPUT** est utilisé pour lire des signaux analogiques, pas des signaux numériques d'interrupteurs.

Question 2

Question ouverte :
Donnez les brancements nécessaires pour connecter deux microcontrôleurs afin de permettre une communication série UART. Indiquez les noms des pins utilisées.

Réponse Question 2

Branchement de la connexion UART :

MCU 1	MCU 2
RX	TX
TX	RX
GND	GND

Explication :

- Le TX (émetteur) d'un MCU doit être connecté au RX (récepteur) de l'autre
- La connexion est croisée : TX1 $\rightarrow$ RX2 et RX1 $\leftarrow$ TX2
- Le GND (masse) doit être commun aux deux systèmes pour avoir une référence de tension commune
- Les deux MCU doivent être configurés avec les mêmes paramètres

BDTech

Question 3

Dans une trame UART standard, quel est toujours le premier bit transmis ?

A. Le MSB (bit de poids fort)
B. Le LSB (bit de poids faible)
C. Un bit de parité
D. Le bit de start

Réponse Question 3

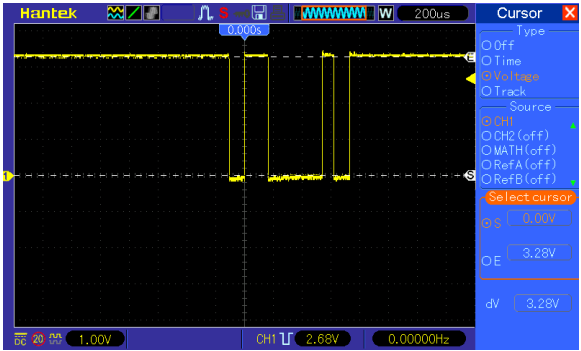
Réponse correcte : D. Le bit de start

Explication :

- Le bit de start est toujours le premier bit d'une trame UART
- C'est une transition de HIGH (état de repos) vers LOW (0)
- Il signale au récepteur qu'une transmission commence
- Après le bit de start viennent les 8 bits de données (LSB first)
- Enfin, un ou plusieurs bits de stop (HIGH) terminent la trame

Question 4

Question ouverte :
Sur un oscilloscope, vous observez la trame suivante : Quel octet hexadécimal est transmis ? Justifiez votre réponse en détaillant la lecture.

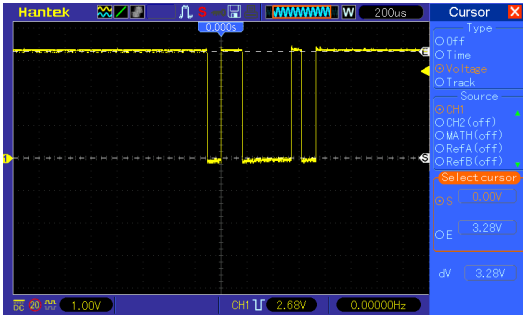


BDTech

Réponse Question 4

Réponse : 0x43 (char 'C')

Explication détaillée :



On lit les bits suivant de gauche à droite : (0110000101)

- Le premier bit est le bit de start (0)
- Les 8 bits de données sont : 11000010 (LSB first) soit l'octet 01000011 en binaire
- Le dernier bit est le bit de stop (1)

Question 5

Que fait le code suivant ?

```
1  int x = 5;
2  while(x--) {
3      printf("%d ", x);
4  }
5  printf("\n\r");
```

A. Affiche : 5 4 3 2 1 0
B. Affiche : 5 4 3 2 1
C. Affiche : 4 3 2 1 0
D. Affiche : 4 3 2 1
E. Boucle infinie

BDTech

Réponse Question 5

Réponse correcte : C. Affiche : 4 3 2 1 0

Explication :

- `while(x--)` utilise l'opérateur de post-décrémentation
- La condition est évaluée AVANT la décrémentation
- Première itération : x=5 (vrai), puis x devient 4, affiche 4
- Deuxième itération : x=4 (vrai), puis x devient 3, affiche 3
- Etc. jusqu'à : x=1 (vrai), puis x devient 0, affiche 0
- Dernière itération : x=0 (faux), la boucle s'arrête

Question 6 ♣

Quelles affirmations sur le code suivant sont vraies ? (Plusieurs réponses possibles)

```
1 int x = 5;
2 int i;
3 for(i = 0; x > 0; i +=--x) {
4     printf("%d ", --x);
5 }
6 printf("\n\r");
```

A. Il y a une erreur de syntaxe
B. le code affiche : 4 3 2 1 0
C. i vaut 3 à la fin de l'exécution
D. La boucle s'exécute 3 fois
E. Aucune de ces réponses

BDTech

Réponse Question 6 ♣

Réponses correctes : C, D

Explication :

- **A - FAUX** : Pas d'erreur de syntaxe, le code compile correctement
- **B - FAUX** : Le code affiche : 4 2 0
- **C - VRAI** : i vaut 3 à la fin
  - Itération 1 : i=0, x=4 → i=0+3=3
  - Itération 2 : i=3, x=2 → i=3+1=4
  - Itération 3 : i=4, x=0 → i=4-1=3
- **D - VRAI** : La boucle s'exécute 3 fois (x=5,3,1)
- **E - FAUX** : Voir explications ci-dessus

Question 7 ♣

Quelles affirmations sur le préprocesseur C sont vraies ? (Plusieurs réponses possibles)

A. Les directives du préprocesseur commencent par #

B. `#define LED_PIN 13 ;` est correct

C. `#pragma once` empêche l'inclusion multiple d'un fichier

D. Le préprocesseur s'exécute après la compilation

Réponse Question 7 ♣

Réponses correctes : A, C

Explication :

- **A - VRAI** : Toutes les directives préprocesseur commencent par # (#include, #define, #ifdef, etc.)
- **B - FAUX** : Les directives préprocesseur ne prennent JAMAIS de point-virgule
  - Correct : `#define LED_PIN 13`
  - Incorrect : `#define LED_PIN 13 ;`(remplace par `13 ;` au lieu de `13`)
- **C - VRAI** : `#pragma once` garantit qu'un fichier header n'est inclus qu'une seule fois (on peut aussi utiliser `#ifndef/#define/#endif`)
- **D - FAUX** : Le préprocesseur s'exécute AVANT la compilation

Ordre de la chaîne de compilation : Préprocesseur → Compilation → Assemblage → Édition de liens

BDTech

Question 8

Question ouverte :
Expliquez la différence entre le passage par valeur et le passage par adresse en C. Donnez un exemple de fonction pour chaque cas.

Réponse Question 8 (1/2)

Passage par valeur :

- Une copie de la variable est passée à la fonction
- La fonction ne peut pas modifier la variable originale

```c
void incrementer_valeur(int x) {
    x = x + 1;  // Modifie seulement la copie
}

int main() {
    int a = 5;
    incrementer_valeur(a);
    printf("%d \n\r", a);  // Affiche 5 (inchange)
    return 0;
}
```

BDTech

Réponse Question 8 (2/2)

Passage par adresse (pointeur) :

- L'adresse mémoire de la variable est passée
- La fonction peut modifier la variable originale

```c
void incrementer_adresse(int *x) {
    *x = *x + 1;  // Modifie la variable originale
}

int main() {
    int a = 5;
    incrementer_adresse(&a);  // Passe l'adresse
    printf("%d \n\r", a);  // Affiche 6 (modifie)
    return 0;
}
```

Utilisation : Le passage par adresse est essentiel pour modifier des variables ou retourner plusieurs valeurs d'une fonction.

Question 9

Quelle affirmations sont vraies ? (Plusieurs réponses possibles)

A. Pour utiliser `printf()` il faut redéfinir `int __io_putchar(int ch)`
B. Les nombres flottants sont supportés automatiquement par `printf()` sur STM32
C. La chaine de charactères "\n" fait un retour à la ligne comme si on appuyait sur "Entrée"
D. `printf()` ne peut être utilisé quand UART
E. Aucune de ces réponses

Réponse Question 9

Réponses correctes : A

Explication :

- **A - VRAI** : Sur STM32, il faut redéfinir la fonction `int __io_putchar(int ch)` pour diriger la sortie de `printf()` vers l'UART ou un autre périphérique
- **B - FAUX** : Par défaut, `printf()` n'inclut pas le support des flottants pour économiser de la mémoire. Il faut l'activer lors de la compilation.
- **C - FAUX** : "\n" fait un saut de ligne, mais sur STM32, il faut souvent ajouter "\r" (retour chariot) pour revenir au début de la ligne.
- **D - FAUX** : `printf()` peut être utilisé avec l'UART mais on peut aussi le rediriger vers d'autres périphériques comme par exemple l'écran que l'on utilise en TP.

Question 10

Quelles affirmations sur la portée des variables en C sont vraies ? (Plusieurs réponses possibles)

A. Une variable déclarée dans une fonction est accessible uniquement dans cette fonction

B. Une variable globale est accessible depuis n'importe quel fichier source sans déclaration préalable

C. Une variable déclarée avec le mot-clé `static` dans une fonction conserve sa valeur entre les appels

D. Les variables locales sont toujours stockées dans la pile (stack)

E. Aucune de ces réponses

Réponse Question 10

Réponses correctes : A, C

Explication :

- **A - VRAI** : Les variables déclarées dans une fonction sont locales à cette fonction et ne sont pas accessibles en dehors.
- **B - FAUX** : Une variable globale doit être déclarée avec le mot-clé `extern` dans les autres fichiers pour être accessible.
- **C - VRAI** : Les variables `static` dans une fonction conservent leur valeur entre les appels successifs de la fonction.
- **D - FAUX** : Les variables locales sont par défaut stockées dans la pile (stack) du programme. Cependant, les variables `static` locales sont stockées dans la section de données statiques du programme comme les variables globales.

NOTE : dans un fichier .h on ne doit jamais définir de variables globales, seulement les déclarer avec `extern`.

Question 11

Quelle fonction HAL permet de lire l'état d'un bouton-poussoir connecté à une GPIO ?

A. `HAL_GPIO_WritePin()`
B. `HAL_GPIO_ReadPin()`
C. `HAL_GPIO_TogglePin()`
D. `HAL_GPIO_GetState()`

Réponse Question 11

Réponse correcte : B. `HAL_GPIO_ReadPin()`

Explication :

- `HAL_GPIO_ReadPin(GPIO_Port, GPIO_Pin)` lit le niveau logique d'une pin configurée en INPUT
- Retourne `GPIO_PIN_SET` (HIGH) ou `GPIO_PIN_RESET` (LOW)

Exemple d'utilisation :

```
1 if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_SET) {
2     // Bouton appuye (selon le cablage)
3     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
4 }
5 else {
6     // Bouton relache
7     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
8 }
```

Question 12

Question ouverte :
Écrivez un programme qui fait clignoter une LED connectée sur PA5 avec une période de 500ms (250ms allumée, 250ms éteinte) en utilisant les fonctions HAL.

Réponse Question 12

```c
// Dans la boucle infinie de la fonction main
while(1) {
    // Allumer la LED (PA5 = HIGH)
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
    HAL_Delay(250);  // Attendre 250ms

    // Eteindre la LED (PA5 = LOW)
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
    HAL_Delay(250);  // Attendre 250ms
}
```

Question 13 ♣

Quelles constantes HAL représentent un niveau logique haut ? (Plusieurs réponses possibles)

A. `GPIO_PIN_SET`

B. `1`

C. `HIGH`

D. `GPIO_PIN_RESET`

Réponse Question 13 ♣

Réponses correctes : A, B

Explication :

- **A - CORRECT** : `GPIO_PIN_SET` est la constante HAL officielle pour le niveau haut
- **B - CORRECT** : `GPIO_PIN_SET` est défini comme `1` dans les bibliothèques HAL
- **C - INCORRECT** : `HIGH` n'est pas défini dans HAL STM32 (c'est une constante Arduino)
- **D - INCORRECT** : `GPIO_PIN_RESET` représente le niveau bas (0)

Définitions HAL :

- `GPIO_PIN_RESET = 0` → Niveau logique bas (GND)
- `GPIO_PIN_SET = 1` → Niveau logique haut (VCC, généralement 3.3V)

BDTech

Question 14

Question ouverte :
Décrivez les étapes de la chaîne de compilation d'un projet C pour microcontrôleur, depuis les fichiers sources jusqu'au fichier exécutable. Indiquez les extensions de fichiers à chaque étape.

Réponse Question 14

Étapes de la chaîne de compilation :

1. **Préprocesseur** :
   - Entrée : Fichiers sources `.c` et headers `.h`
   - Traite les directives # (include, define, ifdef, etc.)
   - Sortie : Code C étendu (sans headers, macros remplacées)
2. **Compilation** :
   - Entrée : Code C préprocessé
   - Traduit le code C en langage assembleur
   - Sortie : Fichiers assembleur `.s` ou `.asm`

Réponse Question 14 (suite)

3. **Assemblage** :
   - Entrée : Fichiers assembleur `.s`
   - Traduit l'assembleur en code machine
   - Sortie : Fichiers objets `.o`
4. **Édition de liens (Linkage)** :
   - Entrée : Fichiers objets `.o` + bibliothèques `.lib` / `.a`
   - Résout les références entre fichiers et lie les bibliothèques
   - Sortie : Fichier exécutable `.hex` pour microcontrôleur

Schéma : `.c/.h` → Préprocesseur → Compilation → `.s` → Assemblage → `.o` → Linkage → `.hex/.elf`

Question 15

Question ouverte : Expliquez ce qu'est un pointeur en C et comment l'utiliser avec un exemple simple.

Réponse Question 15

Définition d'un pointeur :

- Un pointeur est une variable qui stocke l'adresse mémoire d'une autre variables
- Il permet d'accéder et de modifier la valeur de cette variable indirectement

Déclaration et utilisation :

```c
char c = 'A';            // Variable normale
char *p = &c;           // Declaration d'un pointeur vers char, initialise a l'adresse de c
printf("%c\n\r", *p);    // Affiche 'A' (dereferencement du pointeur)
*p = 'B';              // Modifie la valeur de c via le pointeur
printf("%c\n\r", c);     // Affiche 'B'
```

BDTech

Question 16

Que fait ce code ?

```c
int a = 10;
int *p = &a;
*p = 20;
printf("%d\n\r", a);
```

A. Affiche 10
B. Affiche 20
C. Erreur de compilation
D. Comportement indéfini

Réponse Question 16

Réponse correcte : B. Affiche 20

Explication ligne par ligne :

1. `int a = 10;` : Déclare une variable entière a valant 10
2. `int *p = &a;` : Déclare un pointeur p qui stocke l'adresse de a
   - `*p` signifie "pointeur vers un int"
   - `&a` signifie "adresse de a" (opérateur de référencement)
3. `*p = 20;` : Modifie la valeur pointée par p (donc a) à 20
   - `*p` signifie "la valeur à l'adresse contenue dans p" (déréférencement)
4. `printf("%d\n\r", a);` : Affiche la nouvelle valeur de a, soit 20

Question 17 ♣

Quelles déclarations de pointeurs sont correctes en C ? (Plusieurs réponses possibles)

A. `int *ptr;`
B. `int *ptr1, ptr2;` (déclare deux pointeurs)
C. `int *ptr = NULL;`
D. `float **pptr;`

Réponse Question 17 ♣

Réponses correctes : A, C, D

Explication :

- **A - CORRECT** : Déclaration standard d'un pointeur vers int
- **B - PIÈGE** : Cette syntaxe déclare `ptr1` comme pointeur mais `ptr2` comme int simple !
  - Correct pour deux pointeurs : `int *ptr1, *ptr2;` ou `int* ptr1,ptr2;`
- **C - CORRECT** : Initialise un pointeur à NULL (bonne pratique)
- **D - CORRECT** : Pointeur vers pointeur (double indirection)
  - Utile pour tableaux 2D ou modification de pointeurs par fonction

Conseil : Toujours initialiser les pointeurs (à NULL ou une adresse valide) pour éviter des comportements indéfinis (en embarqué il n'y a pas d'OS pour protéger contre les accès mémoire invalides).

BDTech

Question 18

Question ouverte :
Expliquez les trois parties d'une boucle `for` en C et donnez un exemple qui affiche les nombres pairs de 0 à 10.

Réponse Question 18

Structure d'une boucle `for` :

```
for (initialisation; condition; incrementation) {
    // Corps de la boucle
}
```

1. **Initialisation** : Exécutée une seule fois au début (ex : `int i = 0`)
2. **Condition** : Testée avant chaque itération, continue si vraie (ex : `i <= 10`)
3. **Incrémentation** : Exécutée après chaque itération (ex : `i += 2`)

Exemple - Nombres pairs de 0 à 10 :

```
for (int i = 0; i <= 10; i += 2) {
    printf("%d ", i);
}
printf("\n\r");
// Affiche : 0 2 4 6 8 10
```

Question 19

Combien de fois cette boucle s'exécute-t-elle ?

```
1 for (int i = 0; i < 5; i++) {
2     printf("*");
3 }
4 printf("\n\r");
```

A. 4 fois
B. 5 fois
C. 6 fois
D. Boucle infinie

Réponse Question 19

Réponse correcte : B. 5 fois

Explication itération par itération :

1. `i = 0` : $0 < 5$? OUI → Affiche *, puis i devient 1
2. `i = 1` : $1 < 5$? OUI → Affiche *, puis i devient 2
3. `i = 2` : $2 < 5$? OUI → Affiche *, puis i devient 3
4. `i = 3` : $3 < 5$? OUI → Affiche *, puis i devient 4
5. `i = 4` : $4 < 5$? OUI → Affiche *, puis i devient 5
6. `i = 5` : $5 < 5$? NON → Sortie de la boucle

Résultat : Affiche ***** (5 étoiles)

Attention : Si la condition était `i <= 5`, la boucle s'exécuterait 6 fois !

BDTech

Question 20 ♣

Dans un système embarqué, quelles pratiques sont recommandées ? (Plusieurs réponses possibles)

A. Utiliser `HAL_Delay()` dans une routine d'interruption
B. Initialiser les pointeurs à NULL avant utilisation
C. Faire de la programmation multithreadée avec scheduler
D. Éviter les boucles infinies dans le code

Réponse Question 20 ♣

Réponses correctes : B, C

Explication :

- **A - MAUVAISE PRATIQUE** : Ne JAMAIS utiliser de fonctions bloquantes dans une ISR
  - Les interruptions doivent être rapides
  - `HAL_Delay()` bloquerait tout le système
- **B - BONNE PRATIQUE** : Évite les pointeurs sauvages qui causent des crashs
- **C - BONNE PRATIQUE** : La programmation multithreadée avec scheduler permet de gérer plusieurs tâches simultanément, et de garantir à chaque tâche son temps de processeur
- **D - FAUX** : Les boucles infinies `while(1)` sont NORMALES en embarqué
  - Le main() ne doit jamais se terminer sur un microcontrôleur car il n'y a pas d'OS pour reprendre la main

BDTech

Question 21

Question ouverte : Complétez le code suivant.

```c
typedef struct {
    int id;
    char name[20];
} Device;

void updateDevice(Device *devPtr, int newId, char *newName) {
    // Completez ici pour modifier id et name
}

int main() {
    Device myDevice;
    updateDevice(/*Completez ici*/, 101, "Sensor");
    // Completez ici pour afficher les nouvelles valeurs
    return 0;
}
```

Réponse Question 21

```c
void updateDevice(Device *devPtr, int newId, char *newName) {
    // Utilisation de '->' pour acceder au membre via pointeur
    devPtr->id = newId;
    // Copier la nouvelle chaine dans name
    strcpy(devPtr->name, newName);
}

int main() {
    Device myDevice;
    // Passe l'adresse de myDevice
    updateDevice(&myDevice, 101, "Sensor");

    // Affiche les nouvelles valeurs
    printf("ID: %d, Name: %s\n\r", myDevice.id, myDevice.name);
    return 0;
}
```

Bon courage pour l'examen !

Un grand merci à **Bryan ING** et **François-Gabriel SOURBÉ**
pour leur aide précieuse dans la relecture et l'animation du cours.

Avez-vous des questions ?

Rejoignez le Discord du BDTech !

Discord BDTech - ATHENA

Scannez le QR code ou cliquez sur le lien pour rejoindre le serveur Discord du BDTech.

BDTech

Sondage

Scannez le QR code ou cliquez sur le lien pour accéder au sondage de feedback.