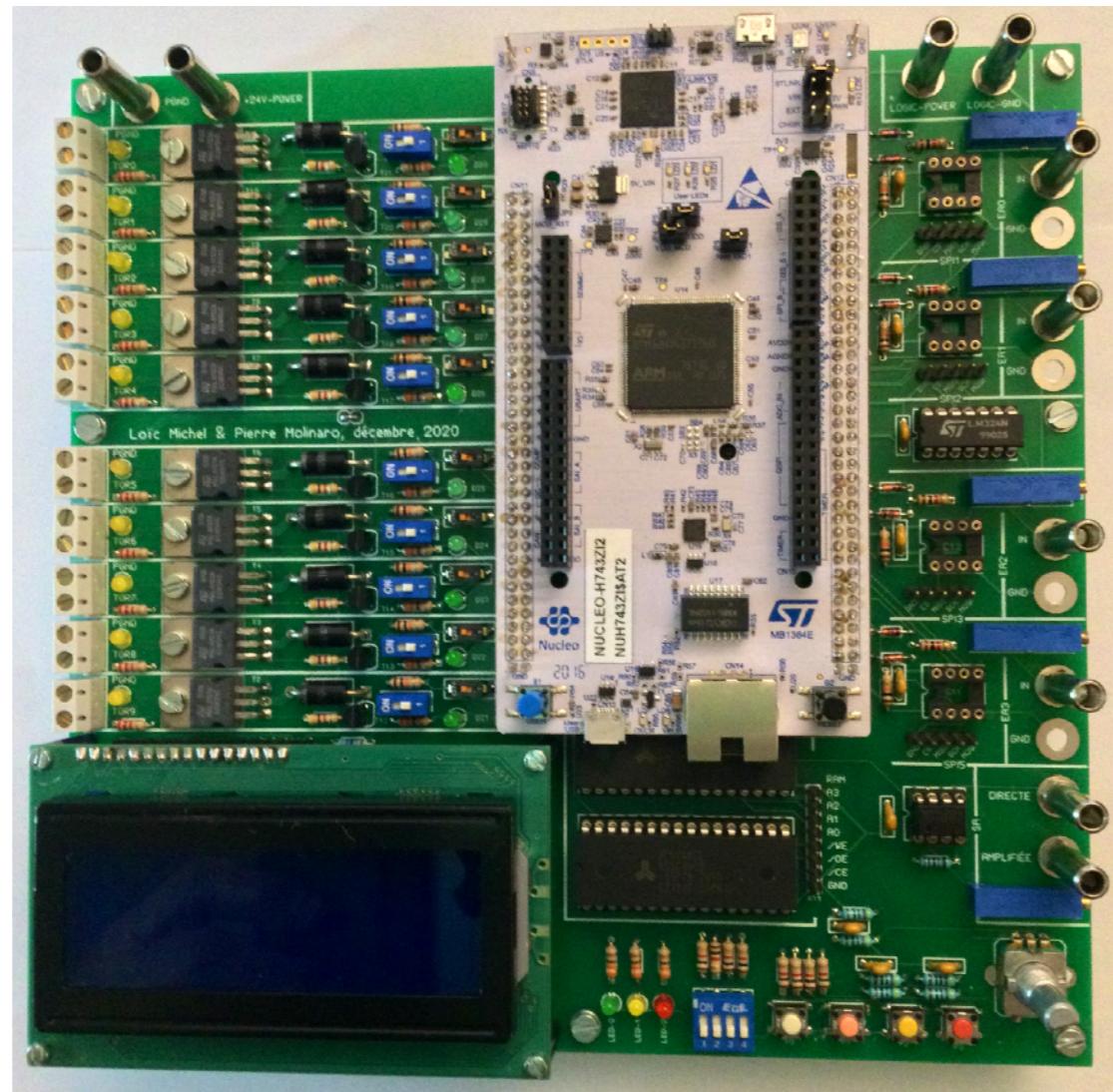


# Mode d'emploi de la carte

# STM32H743 LHEEA



Pierre Molinaro

14 février 2021

A

# Introduction

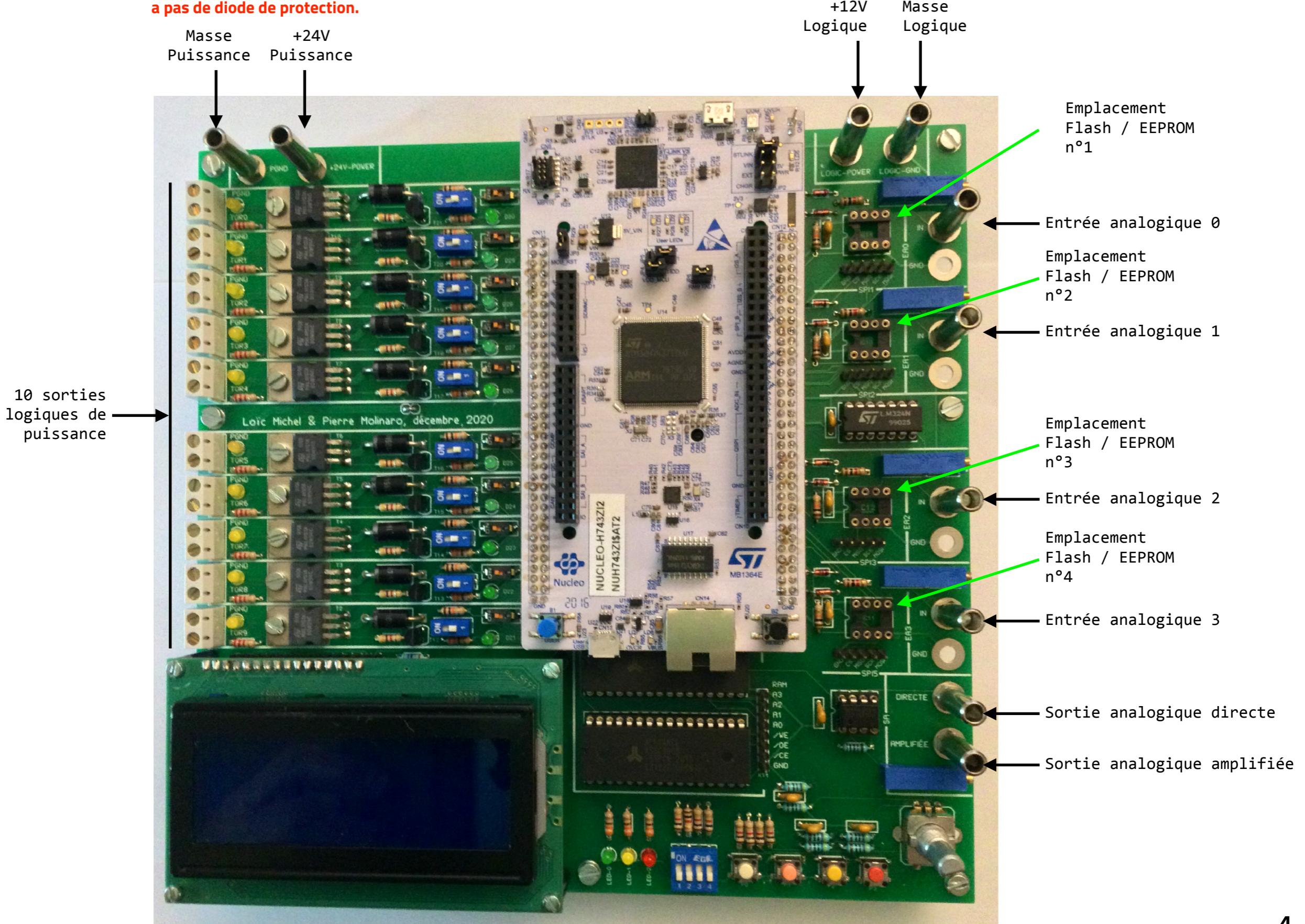
# Carte STM32H743 LHEEA

Le fichier *EICanari* de la carte, les fichiers *keynote* et *pdf*, les croquis d'exemple, sont disponibles à l'URL :

<https://github.com/pierremolinaro/cartes-micro-controleurs-centrale-nantes/tree/master/carte-h743-lheea>

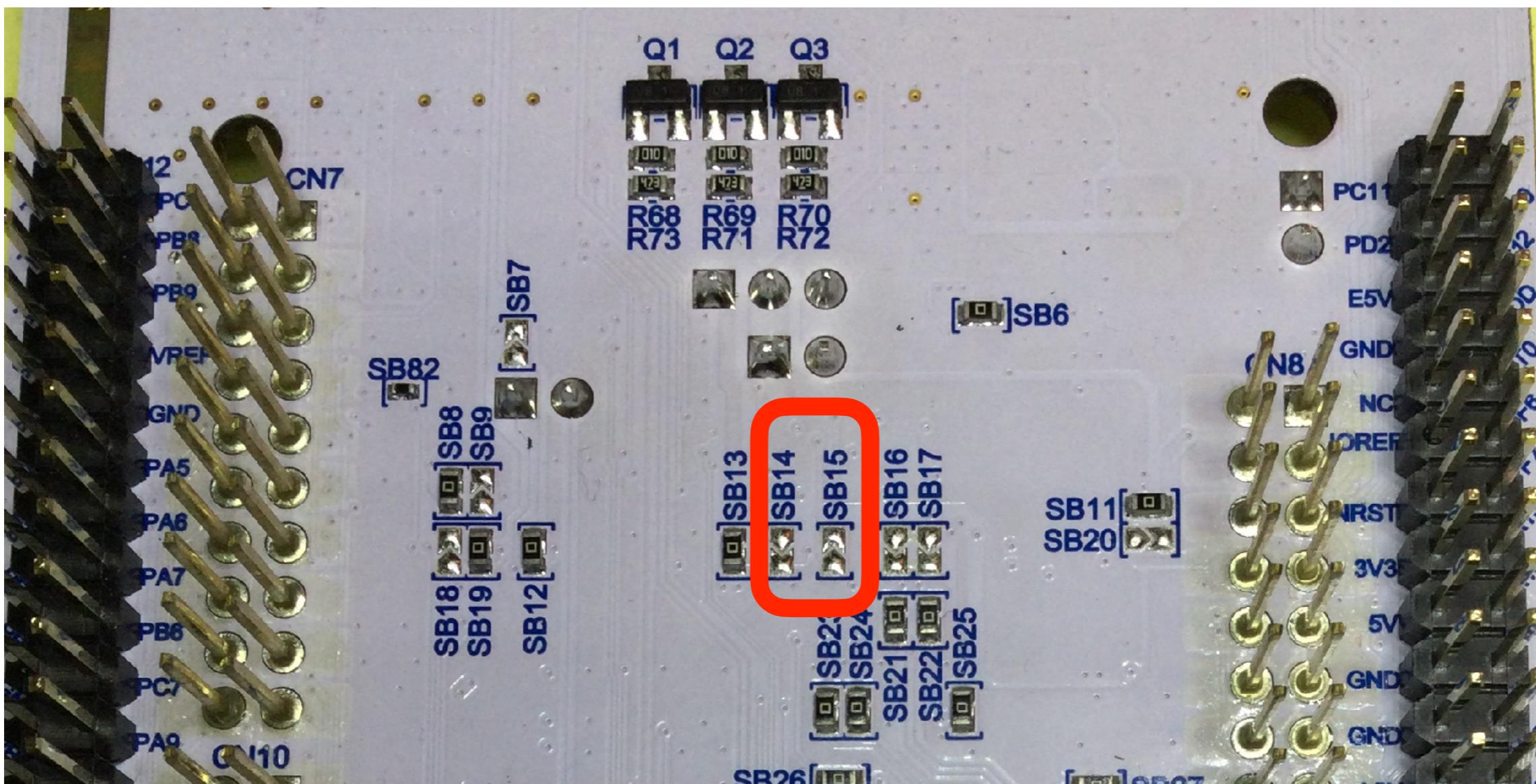
# Schéma de connexion

**Attention ! ne pas se tromper dans la connexion du +24V, il n'y a pas de diode de protection.**



# Avertissement (1/2)

Pour que la rotation de l'encodeur soit pris en compte, il faut effectuer deux ponts de soudure marqués **SB14** et **SB15** au dos de la carte **NUCLEO-H743ZI2**.



# Avertissement (2/2)

**STM32Duino version 1.9.0 présente deux bugs, qu'il faut corriger.**

**La lecture de l'entrée analogique EA0 renvoie toujours 0.** L'entrée EA0 est connectée sur PA6, et STM32Duino v1.9.0 ne reconnaît pas PA6 comme une entrée analogique.

Il faut modifier `~/Library/Arduino15/packages/STM32/hardware/stm32/1.9.0/variants/NUCLEO_H743ZI/PeripheralPins.c`, en décommentant la ligne 57 :

```
{PA_6, ADC1, STM_PIN_DATA_EXT(STM_MODE_ANALOG, GPIO_NOPULL, 0, 3, 0)}, // ADC1_INP3
```

Ce bug a été signalé : [https://github.com/stm32duino/Arduino\\_Core\\_STM32/issues/1277](https://github.com/stm32duino/Arduino_Core_STM32/issues/1277)

**La sortie TOR7 n'est jamais active.** La sortie TOR7 est commandée par PG9, et STM32Duino v1.9.0 ne permet pas de programmer ce port comme une sortie logique.

Il faut modifier `~/Library/Arduino15/packages/STM32/hardware/stm32/1.9.0/variants/NUCLEO_H743ZI/variant.h`, en changeant la ligne 174 :

```
#define NUM_DIGITAL_PINS 99
```

En :

```
#define NUM_DIGITAL_PINS 100
```

Ce bug a été signalé : [https://github.com/stm32duino/Arduino\\_Core\\_STM32/issues/1276](https://github.com/stm32duino/Arduino_Core_STM32/issues/1276)

**B**

# Interfaces utilisateur

# Afficheur LCD

L'afficheur LCD contient 4 lignes de 20 caractères.

Sa gestion est effectuée par la librairie **LiquidCrystal** (<https://www.arduino.cc/en/Reference/LiquidCrystal>).

Son initialisation est effectuée par la fonction `configurerCarteH743LHEEA` (définie dans `STM32H743-configuration-lheea.cpp`), à appeler dans `setup`.

La variable à utiliser est `lcd` (déclarée dans `STM32H743-configuration-lheea.h`).

En résumé :

- `lcd.setCursor (x, y)` déplace la curseur à la ligne  $x$  ( $0 \leq x \leq 3$ ), colonne  $y$  ( $0 \leq y \leq 19$ ) ;
- `lcd.print (v)` imprime  $v$  à l'emplacement du curseur ; celui-ci est avancé du nombre de caractères écrits ;
- attention, ne pas déborder d'une ligne, il n'y a pas prolongement à la ligne suivante : par exemple, la suite de la 1<sup>re</sup> ligne est la 3<sup>e</sup>.

# Leds

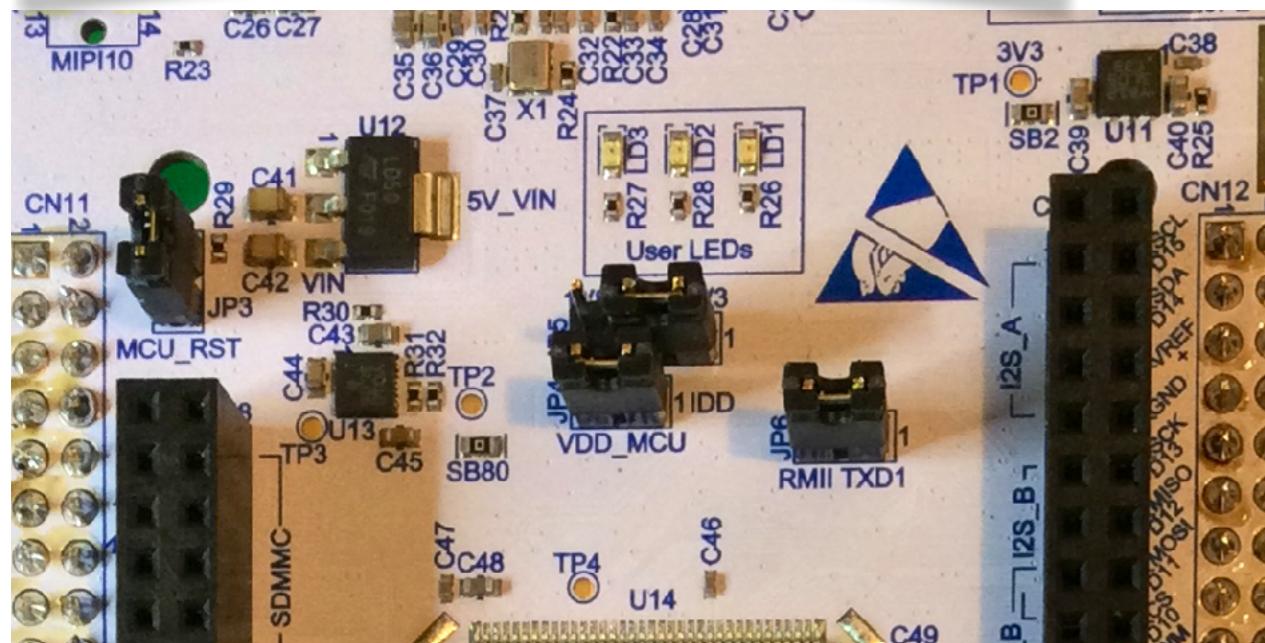
Il y a 6 leds disponibles :

- 3 leds sur la carte Nucleo ;
- 3 leds sur la carte LHEEA.

À chacune correspond un nom (déclaré dans STM32H743-configuration-1heea.h), ce qui permet de les identifier plus facilement. La fonction configurerCarteH743LHEEA (définie dans STM32H743-configuration-1heea.cpp, à appeler dans setup) programme les ports correspondants en sortie.

Écrire un niveau haut par digitalWrite (led, HIGH) allume la led, écrire un niveau bas par digitalWrite (led, LOW) l'éteint.

```
static const uint8_t LED_0_Verte = PA0 ;
static const uint8_t LED_1_Jaune = PA3 ;
static const uint8_t LED_2_Rouge = PB2 ;
static const uint8_t NUCLEO_LD1_Verte = PB0 ;
static const uint8_t NUCLEO_LD2_Jaune = PE1 ;
static const uint8_t NUCLEO_LD3_Rouge = PB14 ;
```



# Interrupteurs DIL

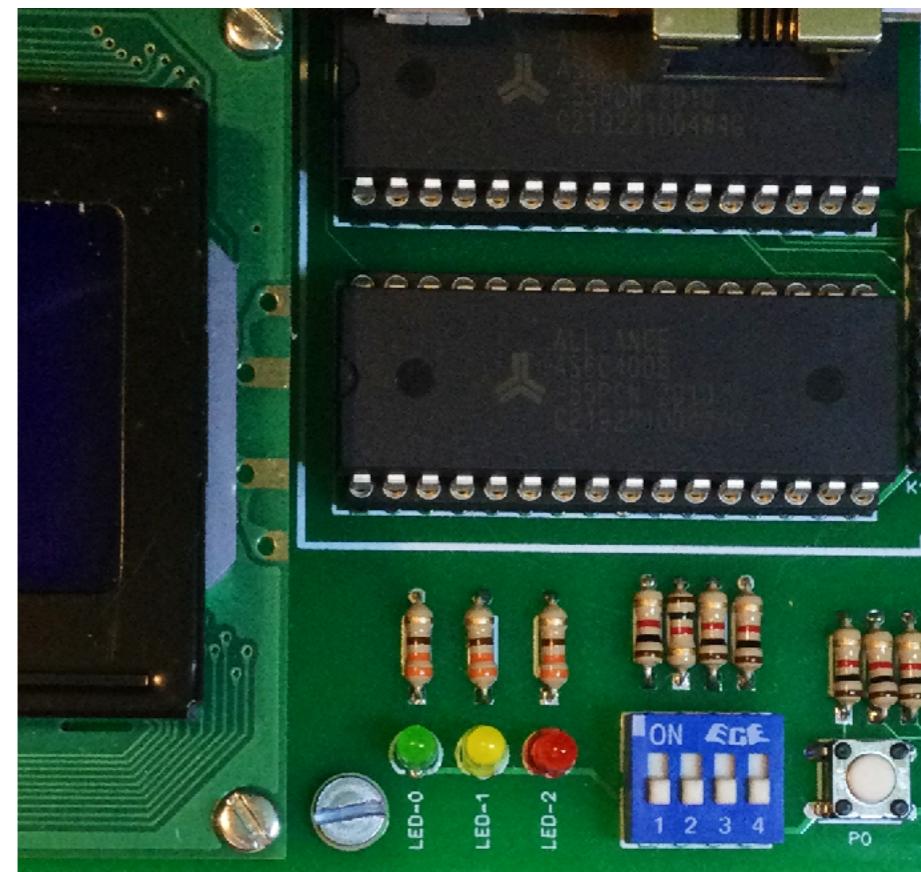
Il y a 4 interrupteurs DIL sur la carte LHEEA.

À chacun correspond un nom (déclaré dans STM32H743-configuration-lheea.h), ce qui permet de les identifier plus facilement. La fonction configurerCarteH743LHEEA (définie dans STM32H743-configuration-lheea.cpp, à appeler dans setup) programme les ports correspondants en entrée (*pullup* activé).

```
static const uint8_t INTER_DIL_1 = PG14 ;  
static const uint8_t INTER_DIL_2 = PB11 ;  
static const uint8_t INTER_DIL_3 = PB9 ;  
static const uint8_t INTER_DIL_4 = PB8 ;
```

Attention, la fonction digitalRead(*interDIL*) renvoie :

- HIGH si l'interrupteur est à OFF ;
- LOW si l'interrupteur est à ON.



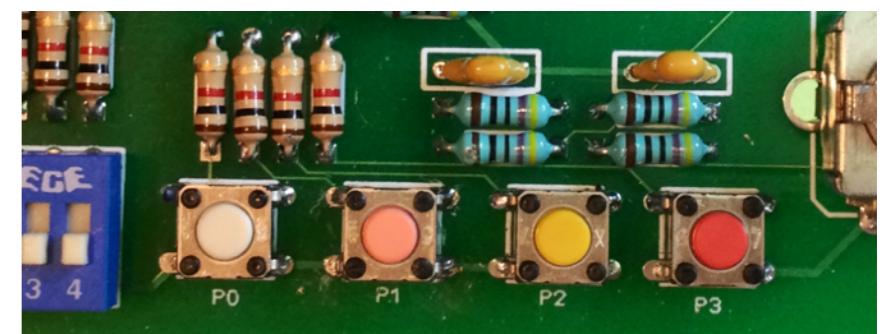
# Poussoirs

Il y a 5 poussoirs :

- un poussoir (bleu) sur la carte Nucleo ;
- quatre poussoirs (blanc, rose, jaune, rouge) sur la carte LHEEA.

À chacun correspond un nom (déclaré dans STM32H743-configuration-lheea.h), ce qui permet de les identifier plus facilement. La fonction configurerCarteH743LHEEA (définie dans STM32H743-configuration-lheea.cpp, à appeler dans setup) programme les ports correspondants en entrée.

```
static const uint8_t POUSSOIR_P0_BLANC = PE0 ;
static const uint8_t POUSSOIR_P1_ROSE = PE2 ;
static const uint8_t POUSSOIR_P2_JAUNE = PE5 ;
static const uint8_t POUSSOIR_P3_ROUGE = PE6 ;
static const uint8_t POUSSOIR_NUCLEO_BLEU = PC13 ;
```



La fonction `digitalRead(POUSSOIR_NUCLEO_BLEU)` renvoie :

- LOW si le poussoir est relâché ;
- HIGH si le poussoir est appuyé.

Pour les autres poussoirs, c'est l'inverse, la fonction `digitalRead(poussoir)` renvoie :

- HIGH si le poussoir est relâché ;
- LOW si le poussoir est appuyé.

# Encodeur (1/2)

L'encodeur a deux fonctions :

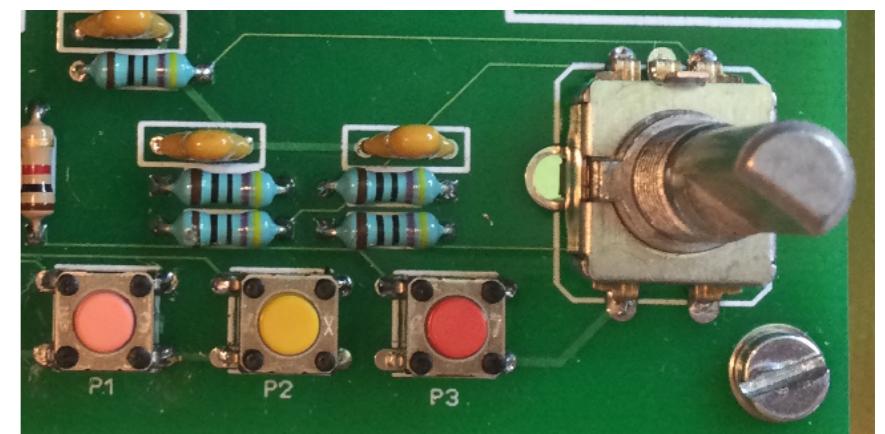
- la première, similaire à celle d'un poussoir ;
- la seconde est la rotation.

**Poussoir.** Pour la fonction poussoir, utiliser la constante ENCODEUR\_CLIC déclarée dans STM32H743-configuration-lheea.h. La fonction configurerCarteH743LHEEA (définie dans STM32H743-configuration-lheea.cpp, à appeler dans setup) programme le port correspondant en entrée.

```
static const uint8_t ENCODEUR_CLIC = PC6 ;
```

La fonction digitalRead(*ENCODEUR\_CLIC*) renvoie :

- HIGH si le poussoir est relâché ;
- LOW si le poussoir est appuyé.



**Rotation.** Prendre en charge la rotation de l'encodeur se fait en deux étapes :

- dans la fonction setup, appeler fixerGammeEncodeur pour fixer la plage des valeurs ;
- ensuite, appeler quand on veut valeurEncodeur pour récupérer la valeur de l'encodeur.

# Encodeur (2/2)

La fonction `fixerGammeEncodeur` a deux arguments :

`fixerGammeEncodeur (borneInf, borneSup)`

où :

- `borneInf` est la borne inférieure de la plage des valeurs ;
- `borneSup` est la borne supérieure de la plage des valeurs.

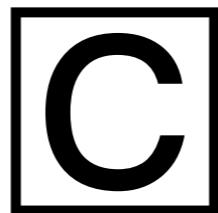
Des valeur négatives sont acceptées (les arguments formels sont de type `int32_t`). Il faut appeler cette fonction avec `borneInf ≤ borneSup`.

Si `borneInf == borneSup`, la fonction `valeurEncodeur` renvoie toujours la valeur commune, indépendamment de la rotation de l'encodeur.

Si `borneInf < borneSup`, la fonction `valeurEncodeur` renvoie toujours une valeur entière dans l'intervalle `[borneInf, borneSup]` :

- une rotation dans le sens trigonométrique décrémente la valeur renvoyée, jusqu'à saturer à `borneInf` ;
- une rotation dans le sens des aiguilles d'une montre incrémente la valeur renvoyée, jusqu'à saturer à `borneSup`.

Initialement, par défaut, `borneInf == borneSup == 0`. La fonction `valeurEncodeur` renvoie alors toujours 0.



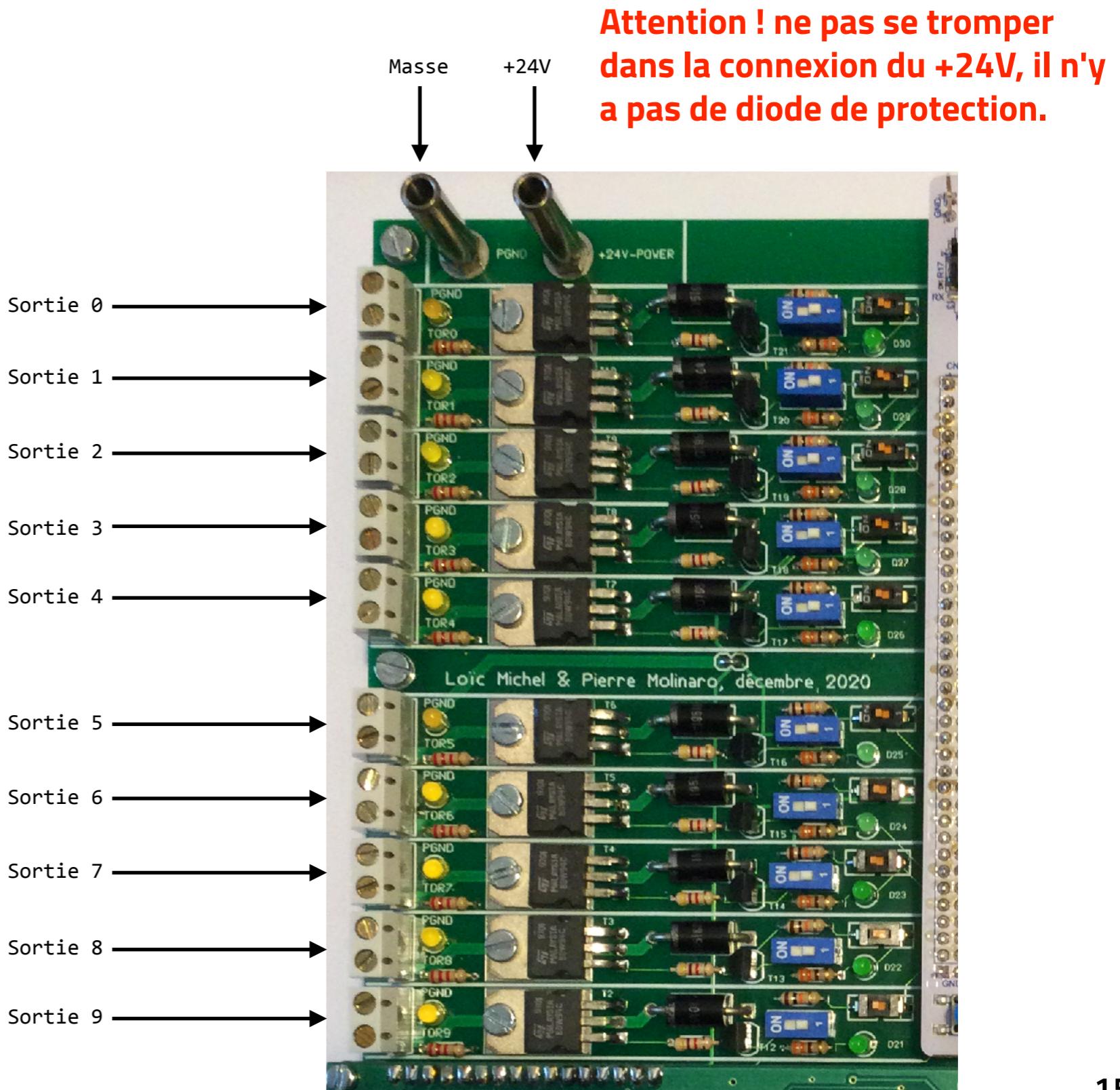
# Sorties TOR

# Les 10 sorties TOR

La carte contient 10 sorties de puissances, appelées sorties TOR (*Tout-Ou-Rien*).

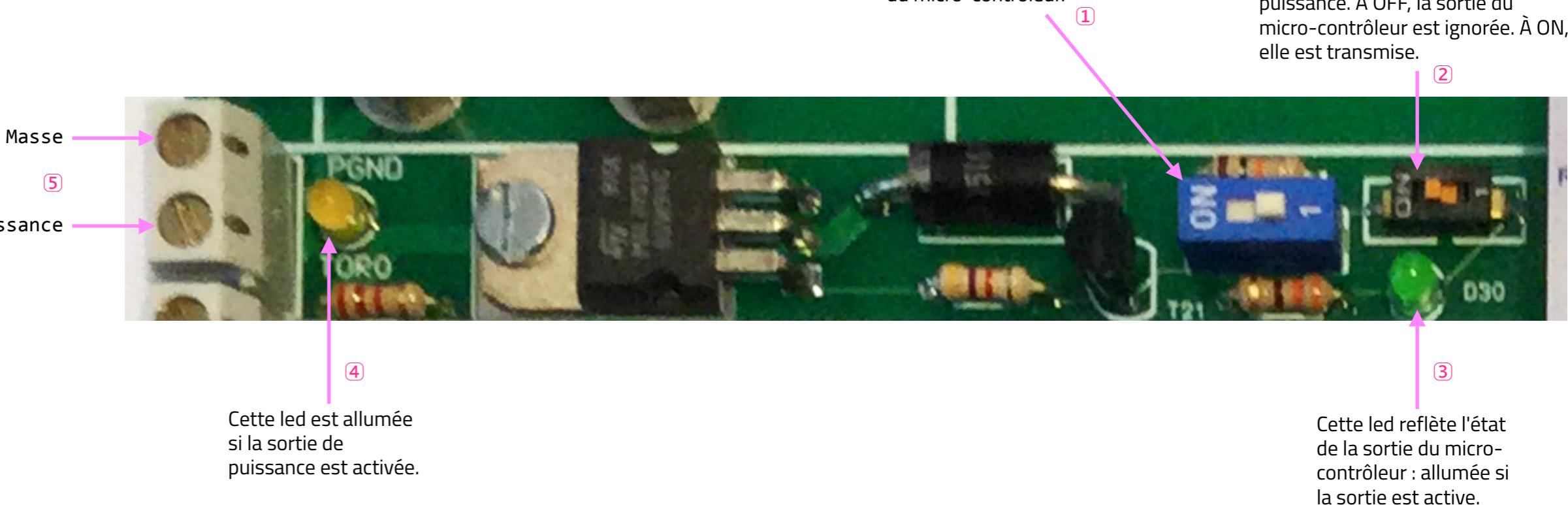
Pour la connexion de l'alimentation de puissance, utiliser exclusivement les bornes PGND et +24V.

La carte présente d'autres bornes de masse, mais il est très déconseillé de les utiliser pour véhiculer les courants importants des sorties de puissance.



# Détails d'une sortie TOR

Les 10 sorties sont disposées de manière identiques.



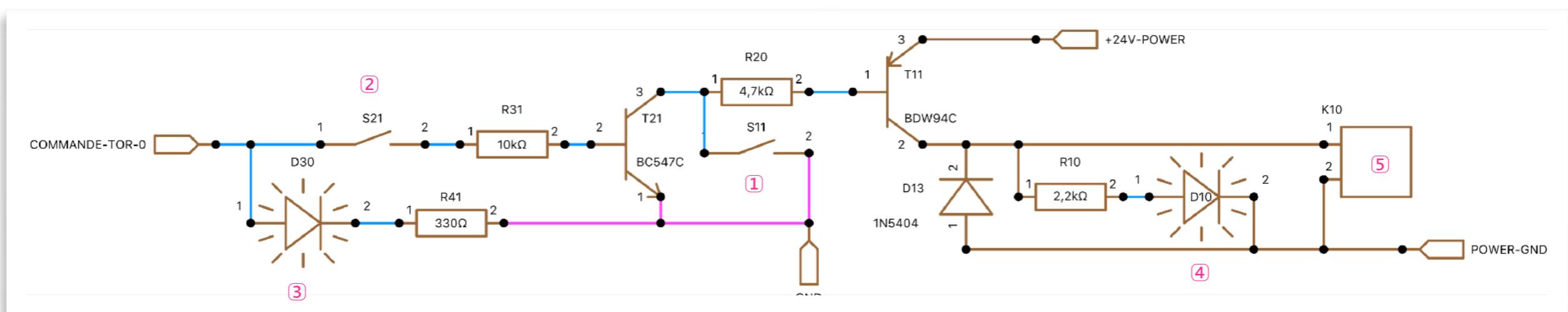
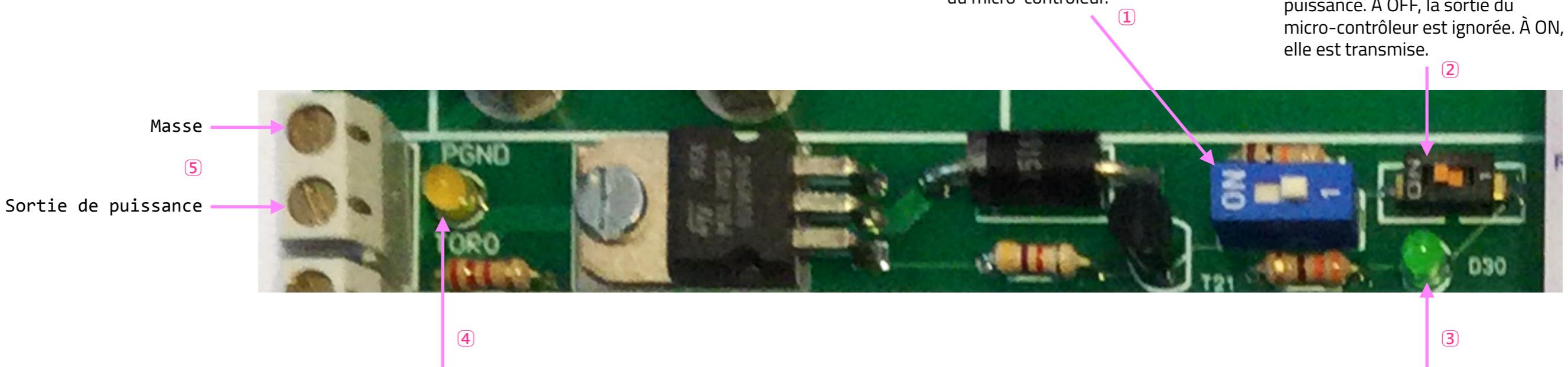
Cet interrupteur contrôle l'activation de la sortie de puissance, indépendamment de la sortie du micro-contrôleur. À ON, la sortie de puissance est activée. À OFF, l'activation dépend de la sortie du micro-contrôleur.

Cet interrupteur contrôle la transmission de la sortie du micro-contrôleur vers l'interface de puissance. À OFF, la sortie du micro-contrôleur est ignorée. À ON, elle est transmise.

| Interrupteur ① | Interrupteur ② | Led ③   | Led ④   | Sortie de puissance ⑤ |
|----------------|----------------|---------|---------|-----------------------|
| ON             | X              | X       | Allumée | Active                |
| OFF            | OFF            | X       | Éteinte | Inactive              |
| OFF            | ON             | Éteinte | Éteinte | Inactive              |
| OFF            | ON             | Allumée | Allumée | Active                |

# Schéma d'une sortie TOR

Les 10 sorties sont disposées de manière identiques.



# Commande d'une sortie TOR

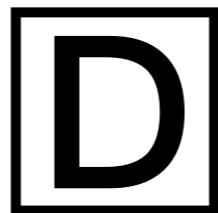
L'initialisation des ports correspondants aux sorties TOR est effectuée par la fonction configurerCarteH743LHEEA (définie dans STM32H743-configuration-1heea.cpp), à appeler dans setup. L'initialisation place les sorties du micro-contrôleur dans l'état inactif (la led ③ éteinte).

Pour commander une sortie, la fonction à utiliser est activerSortieTOR (déclarée dans STM32H743-configuration-1heea.h) :

```
void activerSortieTOR (const uint32_t inIndex, const bool inValue) ;
```

Les arguments sont :

- **inIndex** : le numéro de la sortie (entre 0 et 9) ; si supérieur à 9, l'appel n'a pas d'effet ;
- **inValue** : à true active la sortie du micro-contrôleur (la led ③ est allumée), à false inactive la sortie du micro-contrôleur (la led ③ est éteinte).

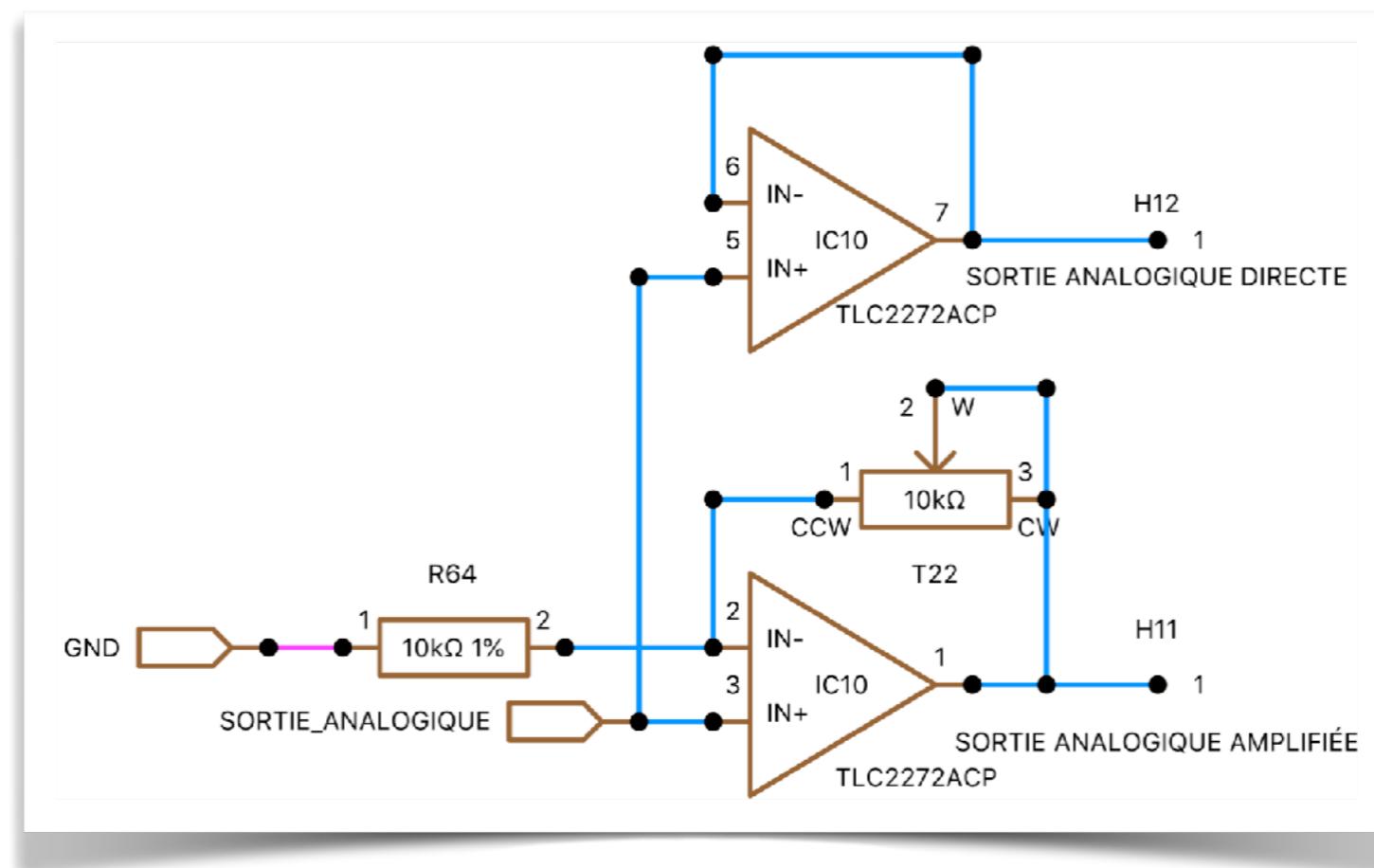
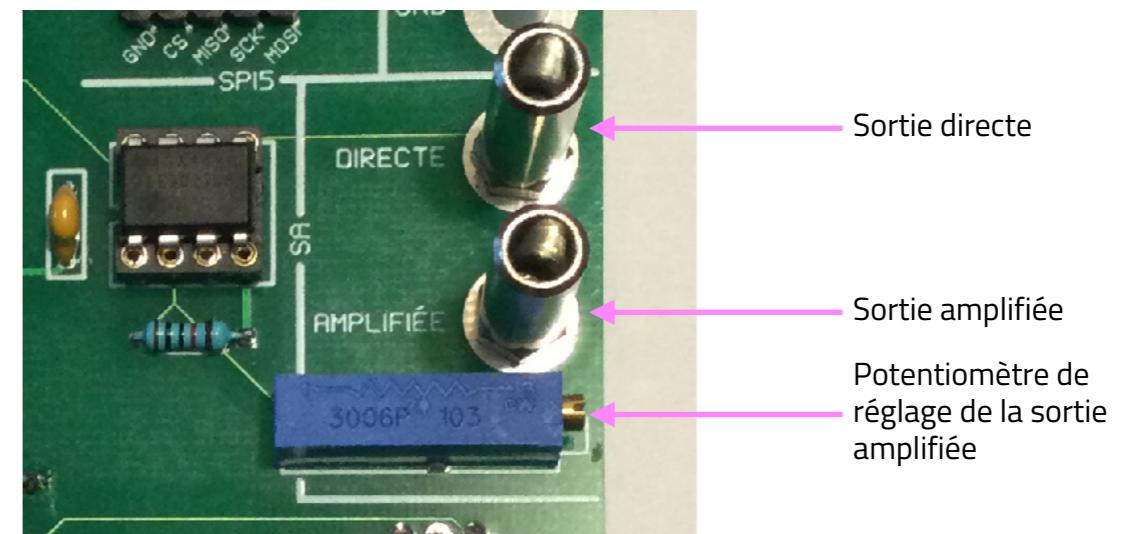


**Sortie analogique**

# Sortie analogique

Le micro-contrôleur intègre un convertisseur numérique / analogique (port : PA4) qui conduit deux sorties à travers deux amplificateur opérationnels :

- la sortie DIRECTE, dont la plage de tension est [0, 3,3V] ;
- la sortie AMPLIFIÉE qui est une image amplifiée de la sortie, réglable grâce au potentiomètre.



# Commander la sortie analogique

```
void commanderSortieAnalogique (const uint8_t inValue) ;
```

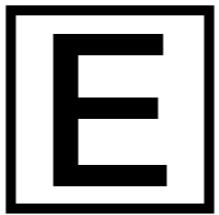
La fonction `commanderSortieAnalogique` a un argument : `inValue`, un entier non signé sur 8 bits qui code la valeur analogique :

- `inValue = 0` -> la sortie directe est à 0V ;
- `inValue = 255` -> la sortie directe est à 3,3V.

La tension de la sortie directe est proportionnelle à `inValue`, par exemple, pour la valeur 100, la tension de la sortie directe est  $100 * 3,3V / 255 = 1,29V$ .

La tension de la sortie amplifiée est aussi proportionnelle à `inValue`, le facteur de proportionnalité dépend du réglage du potentiomètre :

- tourné au maximum dans le sens des aiguilles d'une montre, la tension de la sortie amplifiée est la même que celle de la sortie directe (l'amplificateur opérationnel est configuré en *suiveur*) ;
- tourné au maximum dans le sens trigonométrique, la tension de la sortie amplifiée est le double que celle de la sortie directe (l'amplificateur opérationnel est configuré en *amplificateur non inverseur*) ; en réalité, la tension est un peu inférieure au double, la résistance du potentiomètre a une précision de 10%.



# Entrées analogiques

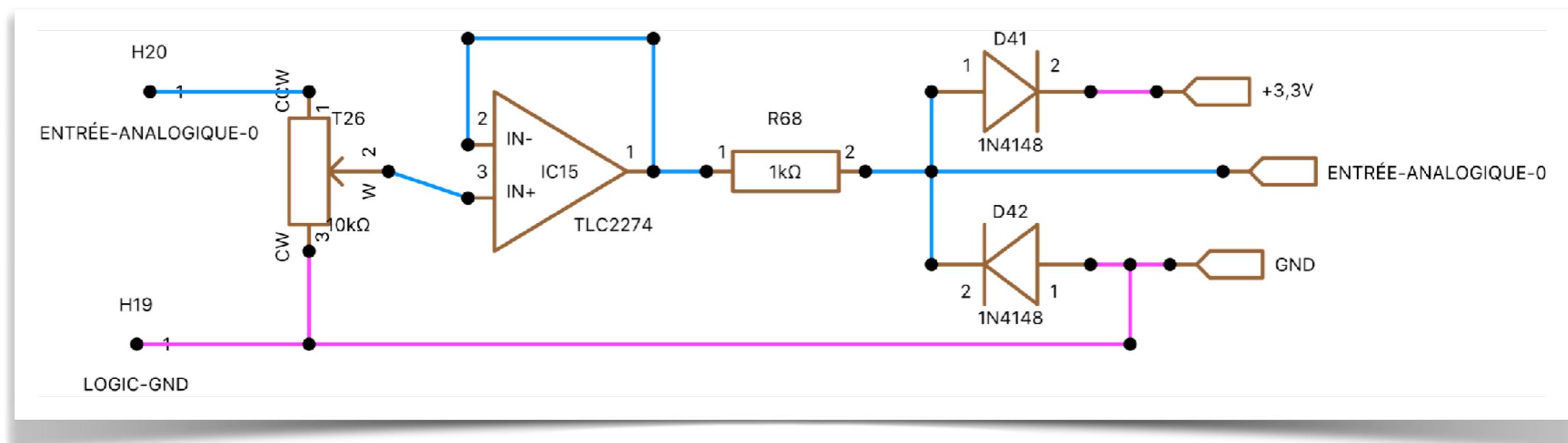
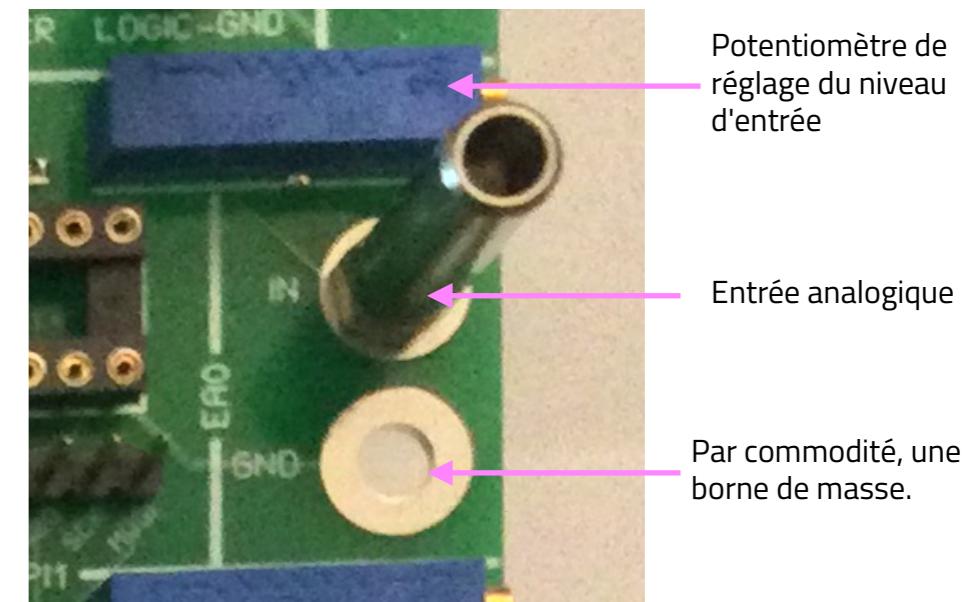
# Les entrées analogiques

La carte contient 4 entrées analogiques dont la disposition et la configuration sont identiques.

Le potentiomètre permet de régler le niveau d'entrée :

- tourné au maximum dans le sens des aiguilles d'une montre, la tension de l'entrée analogique du micro-contrôleur sera toujours à 0 ;
- tourné au maximum dans le sens trigonométrique, la tension de l'entrée analogique du micro-contrôleur sera celle de la borne d'entrée.

Deux diodes 1N4148 protègent le micro-contrôleur d'une surtension à sur l'entrée analogique.



# Lire une entrée analogique

```
uint16_t lireEntreeAnalogique (const uint32_t inNumeroEntree) ;
```

La fonction `lireEntreeAnalogique` a un argument, le numéro (0 à 3) de l'entrée que l'on veut lire (si l'argument est > 3, la valeur renvoyée est toujours 0), et retourne une image de l'entrée analogique.

L'acquisition s'effectue sur 12 bits, c'est-à-dire :

- entrée micro-contrôleur à 0V -> valeur renvoyée 0 ;
- entrée micro-contrôleur à 3,3V -> valeur renvoyée 4095.

La valeur renvoyée est proportionnelle la tension de l'entrée analogique ; par exemple, pour une tension 1,29V, la valeur renvoyée est  $1,29V * 4095 / 3,3V = 1600$ .



**RAM externe**

# Caractéristiques

La carte contient 4 composants SRAM de 512 kio chacun, format une RAM de 2 Mio (2 097 152 octets).

Les composants sont des AS6C4008-55PCN :

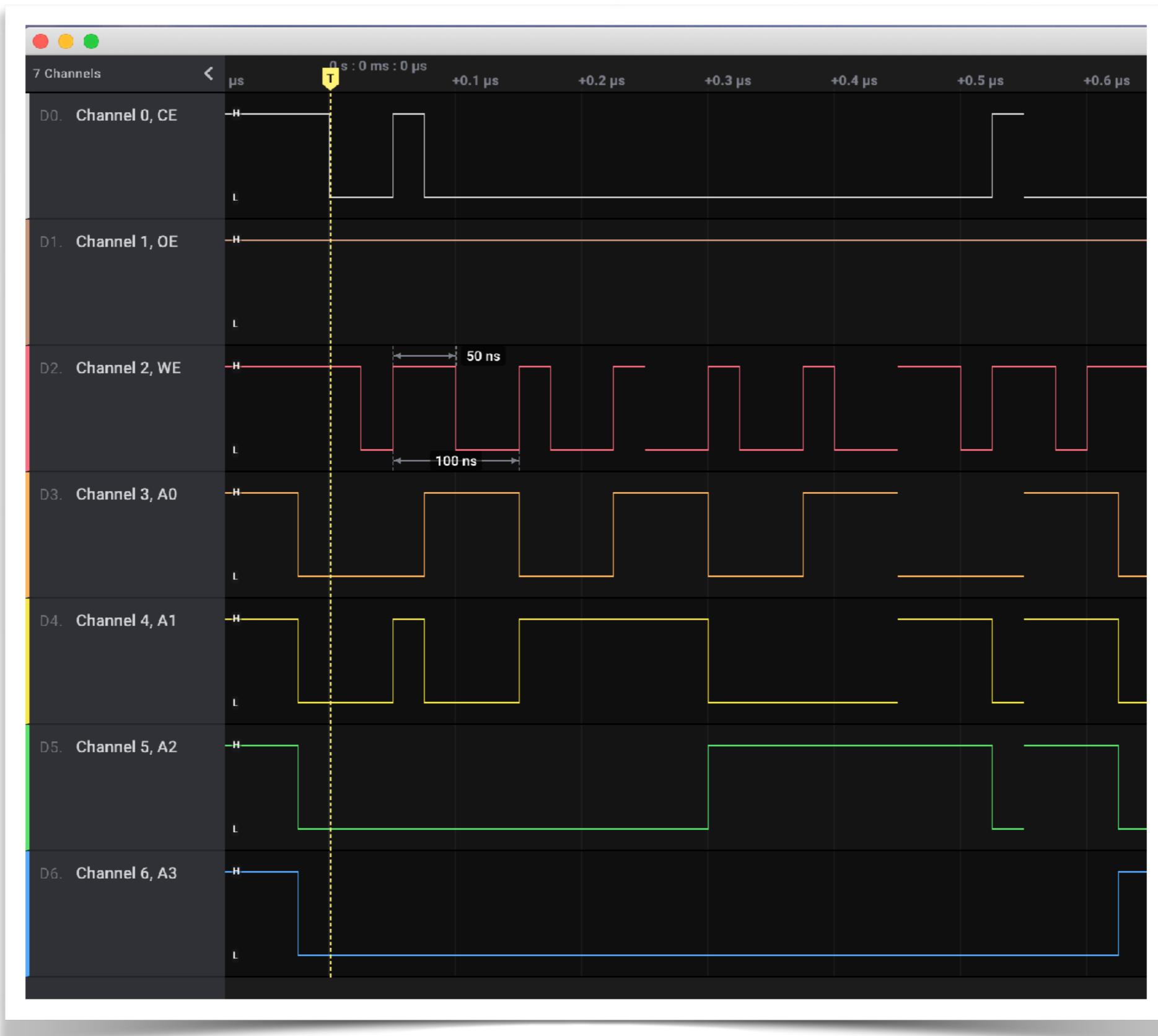
<https://www.tme.eu/fr/details/as6c4008-55pcn/memoires-sram-paralleles/alliance-memory/>

Leur temps d'accès minimum est 55 ns, en pratique, un accès dure environ 100ns.

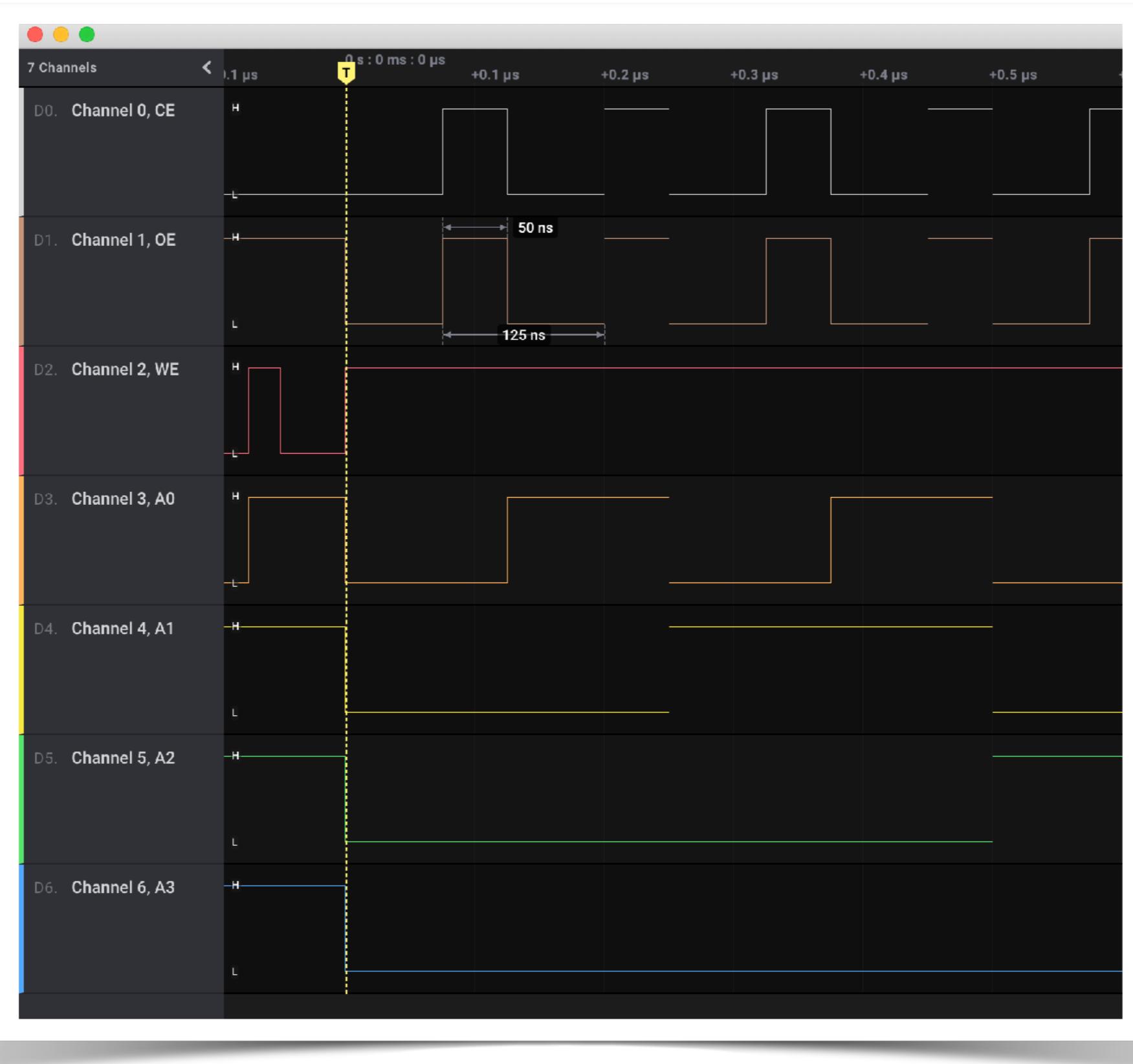
Une boucle de test du croquis « Test RAM externe » dure 419 ms, et chaque octet est écrit puis lu. On calcule ainsi le temps d'accès :

$$\frac{419 \text{ ms}}{2 \cdot 2\,097\,152} \approx 100 \text{ ns}$$

# Chronogramme d'écriture en RAM externe



# Chronogramme de lecture en RAM externe



# Utilisation pratique

La mémoire est à l'adresse 0x6000\_0000, il faut la considérer comme un tableau de 2 097 152 octets.

Les composants sont des AS6C4008-55PCN :

<https://www.tme.eu/fr/details/as6c4008-55pcn/memoires-sram-paralleles/alliance-memory/>

Leur temps d'accès minimum est 55 ns, en pratique, un accès dure environ 100ns.

Une boucle de test du croquis « Test RAM externe » dure 419 ms (code compilé avec l'option « Smallest »), et chaque octet est écrit puis lu. On calcule ainsi le temps d'accès :

$$\frac{419 \text{ ms}}{2 \cdot 2\,097\,152} \approx 100 \text{ ns}$$

# Accès par octet (uint8\_t)

Les deux fonctions d'accès sont :

```
uint8_t readByteAtIndex (const uint32_t inIndex) ;
```

```
void writeByteAtIndex (const uint8_t inValue, const uint32_t inIndex) ;
```

Le paramètre `inIndex` peut prendre toute valeur entre 0 et 2 097 151. Si `inIndex` est  $\geq$  2 097 152 :

- la fonction `readByteAtIndex` retourne 0 ;
- la fonction `writeByteAtIndex` n'a pas effet.

# Accès d'un type quelconque

Deux fonctions génériques d'accès ont été définies :

```
template <typename T> inline T readAtIndex (const uint32_t inIndex) ;
```

```
template <typename T> inline void writeAtIndex (const T inValue, const uint32_t inIndex) ;
```

Le paramètre `inIndex` peut prendre toute valeur entre 0 et 2 097 151, et doit être un multiple de `sizeof(T)`.

Si ces deux conditions ne sont pas remplies :

- la fonction `readAtIndex` retourne la valeur `T()` ;
- la fonction `writeAtIndex` n'a pas effet.

# Accès d'un type quelconque

Deux fonctions génériques d'accès ont été définies :

```
template <typename T> inline T readAtIndex (const uint32_t inIndex) ;  
  
template <typename T> inline void writeAtIndex (const T & inValue,  
                                              const uint32_t inIndex) ;
```

Le paramètre `inIndex` peut prendre toute valeur entre 0 et 2 097 151, et doit être un multiple de `sizeof(T)`.

Si ces deux conditions ne sont pas remplies :

- la fonction `readAtIndex` retourne la valeur `T()` ;
- la fonction `writeAtIndex` n'a pas effet.

Exemple de code :

```
typedef struct { ... } MonType ;  
  
for (uint32_t i = 0 ; i < EXTERNAL_SRAM_SIZE ; i += sizeof (MonType) {  
    MonType variable = ... ;  
    writeAtIndex <T> (variable, i) ;  
}
```



# **FLASHs et EEPROMs**

## **externes**

# EEPROM et FLASH

4 emplacements permettent d'accueillir chacun au choix :

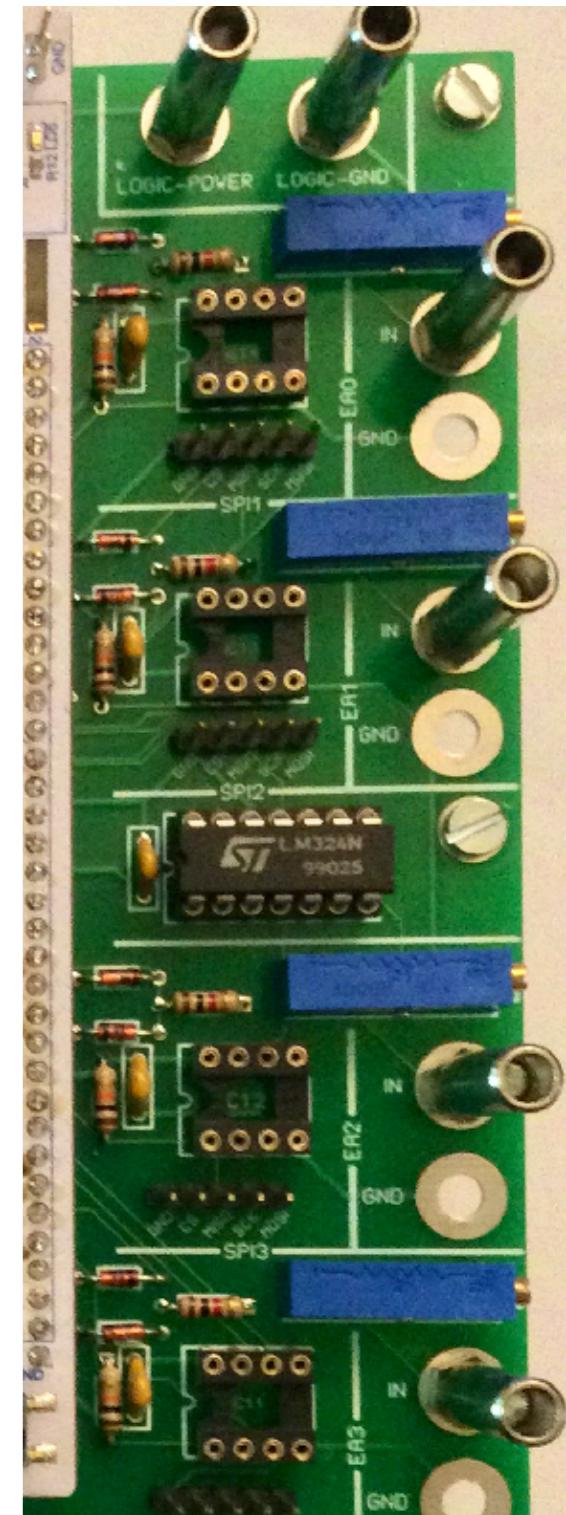
- une EEPROM ;
- une FLASH.

Une **EEPROM** est une mémoire permanente :

- lecture octet par octet, nombre de lectures indéfini ;
- une écriture prend du temps pour être accomplie, typiquement 5 ms ;
- possibilité d'effectuer simultanément l'écriture de plusieurs octets consécutifs, à conditions qu'ils résident tous dans la même page ;
- nombre d'écritures garanties limité (typique  $10^6$ ).

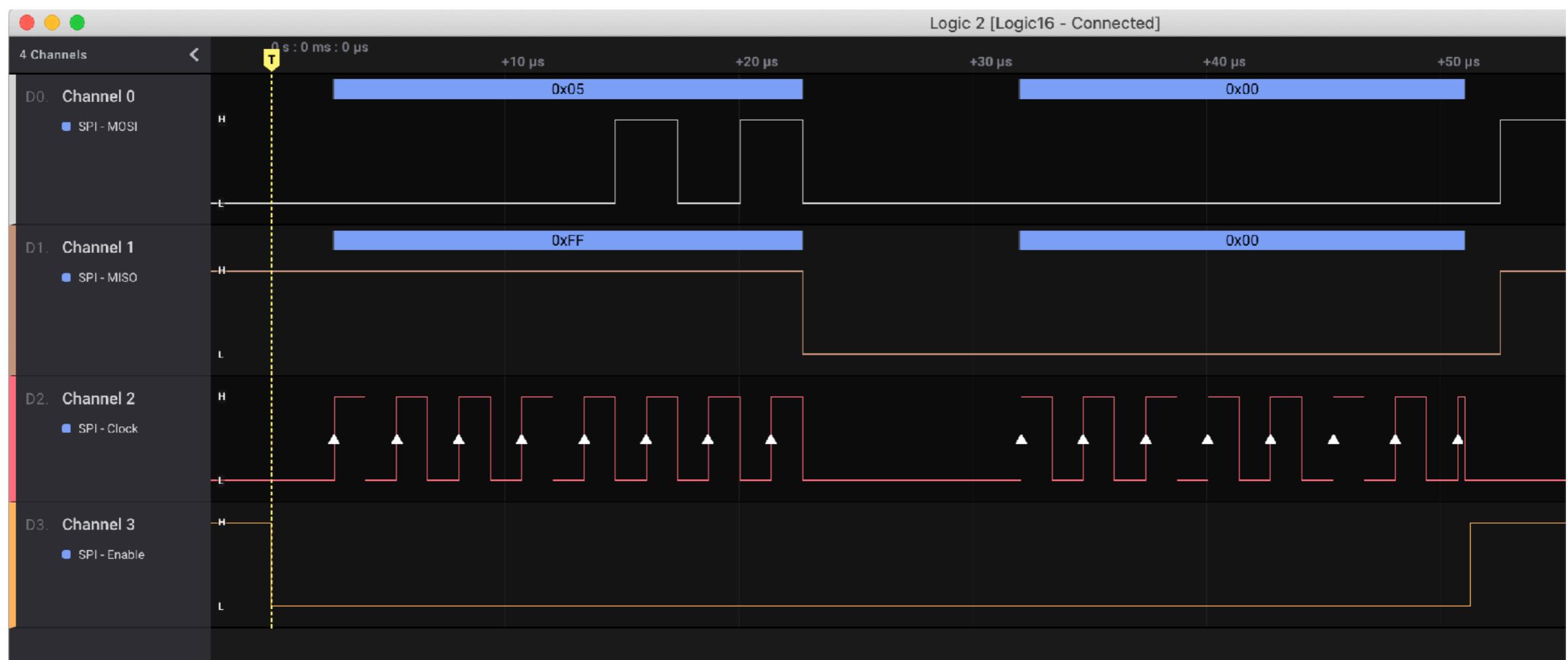
Une **FLASH** est une mémoire permanente :

- lecture octet par octet, nombre de lectures indéfini ;
- une écriture prend du temps pour être accomplie, typiquement 5 ms ;
- possibilité d'effectuer simultanément l'écriture de plusieurs octets consécutifs, à conditions qu'ils résident tous dans la même page ; l'écriture ne peut que changer des bits à 1 en bits à 0 ;
- l'effacement de la mémoire (c'est--à-dire positionner les bits à 1) s'effectue par page ;
- nombre d'écritures garanties limité (typique  $10^5$ ).



# Exemple d'échange SPI

Voici un exemple d'échange SPI, une transaction sur 2 octets.



# Bug SPI dans STM32Duino version 1.9.0 (1/2)

Il y a un bug dans le STM32Duino, mis en évidence dans le croquis `croquis-bug-spi-h743` et signalé par [https://github.com/stm32duino/Arduino\\_Core\\_STM32/issues/1303](https://github.com/stm32duino/Arduino_Core_STM32/issues/1303). Lorsque l'on utilise `SPI.transfer` ou `SPI.transfer16`, SS repasse systématiquement au niveau haut après chaque appel, même si on utilise le paramètre `SPI_CONTINUE`.

```
#include <SPI.h>

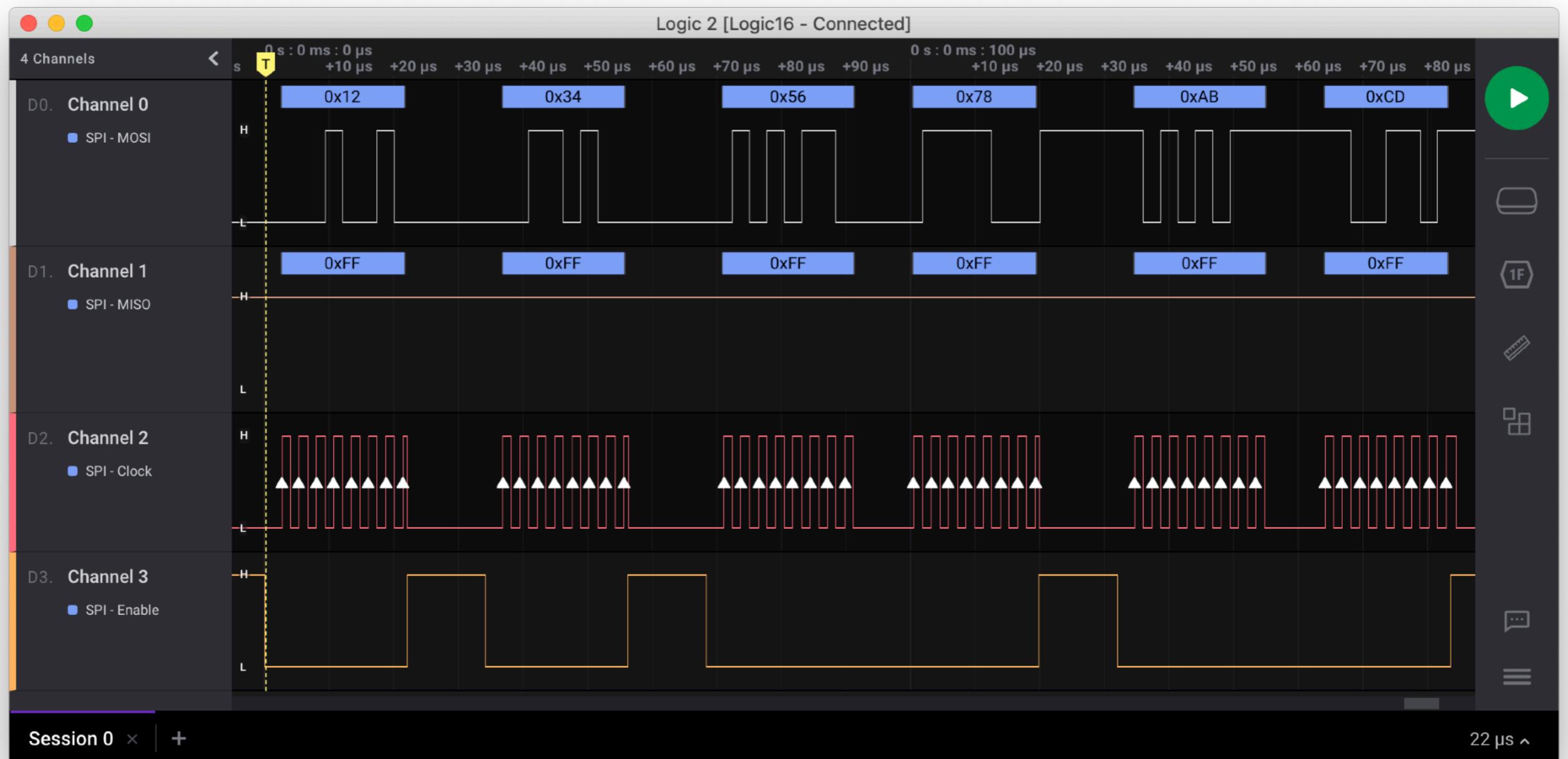
static const uint8_t SPI3_MOSI = PC12 ;
static const uint8_t SPI3_MISO = PC11 ;
static const uint8_t SPI3_SSEL = PA15 ;
static const uint8_t SPI3_SCLK = PC10 ;

void setup() {
    SPI.setMOSI (SPI3_MOSI) ;
    SPI.setMISO (SPI3_MISO) ;
    SPI.setSCLK (SPI3_SCLK) ;
    SPI.setSSEL (SPI3_SSEL) ;
    SPI.begin () ;
}

void loop () {
    delay (100) ;
    SPI.beginTransaction (SPISettings (1000 * 1000, MSBFIRST, SPI_MODE0)) ;
    SPI.transfer (0x12, SPI_CONTINUE) ;
    SPI.transfer (0x34, SPI_CONTINUE) ;
    SPI.transfer16 (0x5678, SPI_CONTINUE) ;
    SPI.transfer16 (0xABCD, SPI_LAST) ;
    SPI.endTransaction () ;
}
```

# Bug SPI dans STM32Duino version 1.9.0 (2/2)

Mise en évidence du bug avec un analyseur logique.





# Accès à une EEPROM externe

# EEPROMs supportées

Actuellement, 2 types d'EEPROM sont acceptées :

- 25LC256 ([https://www.tme.eu/fr/details/25lc256-i\\_p/memoires-eeprom-de-serie/microchip-technology/](https://www.tme.eu/fr/details/25lc256-i_p/memoires-eeprom-de-serie/microchip-technology/));
- 25LC512 ([https://www.tme.eu/fr/details/25lc512-i\\_p/memoires-eeprom-de-serie/microchip-technology/](https://www.tme.eu/fr/details/25lc512-i_p/memoires-eeprom-de-serie/microchip-technology/)).

Les caractéristiques de ces EEPROMS sont résumées dans le tableau suivant :

| EEPROM         | Capacité | Page       | Fréquence max SPI | Durée max écriture | Endurance     |
|----------------|----------|------------|-------------------|--------------------|---------------|
| <b>25LC256</b> | 32 kio   | 64 octets  | 10 MHz            | 5 ms               | $10^6$ cycles |
| <b>25LC512</b> | 64 kio   | 128 octets | 20 MHz            | 5 ms               | $10^6$ cycles |

L'endurance est comptée en nombre de cycles d'effacement / écritures.

Attention, les cycles sont comptés différemment pour les deux mémoires :

- pour la 25LC256, chaque octet est écrit individuellement ;
- pour la 25LC512, une écriture s'effectue toujours sur une page complète, quelque soit le nombre d'octets concernés par l'ordre d'écriture.

# Programmation de l'accès à une EEPROM externe

Pour un exemple pratique, voir à [cette page](#) le croquis d'exemple 07-croquis-test-eeprom-externe.

① Déclarer une variable globale (ici `mySPIEEPROM`, le nom est libre) qui précise l'emplacement utilisé et le type de la mémoire, par exemple :

```
static SPIEEPROM mySPIEEPROM (MySPI::spi5, SPI_EEPROM_TYPE::MCP25LC256) ;
```

Les valeurs possibles du premier argument sont : `MySPI::spi1`, `MySPI::spi2`, `MySPI::spi3`, `MySPI::spi5`.

Celles du second argument : `SPI_EEPROM_TYPE::MCP25LC256`, `SPI_EEPROM_TYPE::MCP25LC512`.

② Dans la fonction `setup`, appeler :

```
mySPIEEPROM.begin () ;
```

③ Dans la fonction `loop`, on peut appeler les méthodes de la classe `SPIEEPROM`, en particulier :

- `eepromRead`, pour lire l'EEPROM ;
- `eepromWrite`, pour l'écrire ;
- `eepromIsBusy`, pour savoir si une écriture est en cours.

# Lecture d'une EEPROM externe : eepromRead

La méthode eepromRead de la classe SPIEEPROM réalise une lecture d'une EEPROM :

```
void eepromRead (const uint32_t inAddress,  
                  uint8_t outBuffer[],  
                  const uint32_t inLength);
```

Les arguments sont :

- `inAddress`, l'adresse de lecture en EEPROM ;
- `outBuffer`, un tableau d'octets, d'une taille d'au moins `inLength` éléments, qui reçoit les octets lus ;
- `inLength`, le nombre d'octets à lire.

Notes :

- si une écriture est en cours, la méthode attend qu'elle soit terminée avant d'effectuer la lecture ;
- le débordement n'est pas testé : aussi, l'octet effectivement lu est celui d'adresse modulo la taille de la mémoire.

# Écriture d'une EEPROM externe : eepromWrite

La méthode `eepromWrite` de la classe `SPIEEPROM` réalise une écriture d'une EEPROM :

```
void eepromWrite (const uint32_t inAddress,  
                  const uint8_t inBuffer[],  
                  const uint32_t inLength) ;
```

Les arguments sont :

- `inAddress`, l'adresse d'écriture en EEPROM ;
- `inBuffer`, un tableau d'octets, d'une taille d'au moins `inLength` éléments, qui contient les octets à écrire ;
- `inLength`, le nombre d'octets à écrire.

Notes :

- si une écriture précédente est en cours, la méthode attend qu'elle soit terminée avant d'effectuer l'écriture ;
- le débordement n'est pas testé : aussi, l'octet effectivement écrit est celui d'adresse modulo la taille de la mémoire ;
- techniquement, une seule page peut être écrite à la fois ; si les données sont à écrire sur plusieurs pages, la méthode scinde automatiquement les données et effectue plusieurs écritures consécutives ; la durée d'exécution s'en trouve rallongée, puisqu'une écriture ne peut être lancée qu'une fois la précédente terminée.

# Classe générique VarInEEPROM

La classe générique VarInEEPROM permet d'utiliser des variables dont la valeur est automatiquement mémorisée en EEPROM.

Pour un exemple pratique, voir à [cette page](#) le croquis d'exemple 08-croquis-test-eeprom-storage.

① Déclarer une variable globale (ici gVar0, le nom est libre) qui précise son type, l'EEPROM et l'adresse de mémorisation dans celle-ci. Par exemple :

```
static VarInEEPROM <uint32_t> gVar0 (mySPIEEPROM, 0) ;
```

Le type de la variable est fixé par le type choisi pour le type générique, ici `uint32_t`. Le premier argument est une variable de type `SPIEEPROM`, qui désigne l'EEPROM dans laquelle est mémorisée la valeur de la variable. Le second argument est l'adresse de mémorisation de la valeur de la variable dans l'EEPROM.

**Attention**, il faut choisir les adresses de façon que les variables ne se chevauchent pas. Ainsi, un `uint32_t` occupe 4 octets : comme `gVar0` est à l'adresse 0, la prochaine variable doit avoir une adresse supérieure ou égale à 4.

② Dans les fonctions `setup` et `loop`, on peut appeler les méthodes de la classe `VarInEEPROM`, en particulier :

- `get`, pour obtenir la valeur de la variable ; au premier appel, la variable est chargée en effectuant une lecture en EEPROM ;
- `set`, pour écrire une valeur dans la variable ; la nouvelle valeur est écrite en EEPROM.

# Type énuméré SPI\_EEPROM\_TYPE

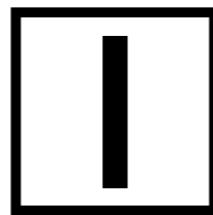
Le type énuméré SPI\_EEPROM\_TYPE définit les EEPROMS utilisables et permet l'accès à leurs caractéristiques.

```
enum class SPI_EEPROM_TYPE {  
    MCP25LC256, // 512 pages de 64 octets -> 32 kio  
    MCP25LC512 // 512 pages de 128 octets -> 64 kio  
};
```

Les fonctions eepromCapacity, eepromPageSize et eepromMaxFrequency permettent d'accéder à leurs caractéristiques :

```
uint32_t eepromPageSize (const SPI_EEPROM_TYPE inEEPROM) ;  
  
uint32_t eepromMaxFrequency (const SPI_EEPROM_TYPE inEEPROMType) ;  
  
uint32_t eepromCapacity (const SPI_EEPROM_TYPE inEEPROMType) ;
```

| EEPROM                      | eepromCapacity | eepromPageSize | eepromMaxFrequency |
|-----------------------------|----------------|----------------|--------------------|
| SPI_EEPROM_TYPE::MCP25LC256 | 32 768         | 64             | 10 000 000         |
| SPI_EEPROM_TYPE::MCP25LC512 | 65 536         | 128            | 20 000 000         |



# Accès à une FLASH externe

# FLASHs supportées

Actuellement, 2 types de Flashs sont acceptées :

- SST26VF064B ([https://www.tme.eu/fr/details/26vf064b-104i\\_sm/memoires-flash-de-serie/microchip-technology/sst26vf064b-104i-sm/](https://www.tme.eu/fr/details/26vf064b-104i_sm/memoires-flash-de-serie/microchip-technology/sst26vf064b-104i-sm/));
- IS25LP128 (<https://www.tme.eu/fr/details/is25lp128-jble/memoires-flash-de-serie/issi/>).

Les caractéristiques de ces Flashs sont résumées dans le tableau suivant :

| Flash              | Capacité | Page       | Secteur      | Fréquence max SPI | Endurance     |
|--------------------|----------|------------|--------------|-------------------|---------------|
| <b>SST26VF064B</b> | 8 Mio    | 256 octets | 4 096 octets | 40 MHz            | $10^5$ cycles |
| <b>IS25LP128</b>   | 16 Mio   | 256 octets | 4 096 octets | 40 MHz            | $10^5$ cycles |

# Différence entre une FLASH et une EEPROM

Pour une EEPROM, l'opération d'écriture (`eepromWrite`) écrit les données.

Pour une Flash, l'opération d'écriture (`flashWrite`) ne peut que transformer les bits à 1 en mémoire en bits à 0. Autrement dit, après écriture le contenu d'un octet en flash est le résultat de l'opération « et » bit-à-bit entre le contenu précédent de la mémoire et la donnée que l'on soumet

Donc en pratique :

- l'écriture en Flash écrit uniquement des zéros ;
- pour revenir à des bits à 1, il faut faire des opérations d'effacement, qui peuvent être effectuées :
  - soit par secteur (`flashEraseSector`) ;
  - soit pour tout le contenu du composant (`flashEraseChip`).

# Exemple pratique

Supposons que l'on veut gérer un tableau de 6 octets à l'adresse 217 055 dans la Flash (LONGUEUR\_DONNEES = 6 et ADRESSE\_EN\_FLASH = 217055). C'est ce qui est mis en œuvre dans le croquis d'exemple 09-croquis-test-flash-externe (voir à [cette page](#)).

Pour la lecture, aucun problème, il suffit d'exécuter :

```
myFlash.flashRead (ADRESSE_EN_FLASH, buffer, LONGUEUR_DONNEES) ;
```

Pour l'écriture de même, il suffit d'exécuter :

```
myFlash.flashWrite (ADRESSE_EN_FLASH, buffer, LONGUEUR_DONNEES) ;
```

Mais l'écriture ne fait qu'écrire des 0. Si on veut être sûr de la valeur écrite, il faut effacer la donnée, avant de l'écrire une seule fois. L'effacement peut se faire soit pour toute la mémoire, en une seule fois : flashEraseChip, soit secteur par secteur.

Pour cette dernière opération, on calcule le numéro du premier secteur à effacer (ici  $\frac{217\ 055}{4\ 096} = 52$ ) :

```
const uint32_t premierSecteur = ADRESSE_EN_FLASH / myFlash.flashSectorSize () ;
```

Puis le dernier numéro du secteur à effacer (ici  $\frac{217\ 055 + 6 - 1}{4\ 096} = 52$ ) :

```
const uint32_t dernierSecteur  
= (ADRESSE_EN_FLASH + LONGUEUR_DONNEES - 1) / myFlash.flashSectorSize () ;
```

Et on appelle flashEraseSector pour tous les secteurs de premierSecteur à dernierSecteur compris.

# Type énuméré SPI\_FLASH\_TYPE

Le type énuméré SPI\_FLASH\_TYPE définit les flashes utilisables et permet l'accès à leurs caractéristiques.

```
enum class SPI_FLASH_TYPE {  
    SST26VF064B, // 8 Mio  
    IS25LP128 // 16 Mio  
};
```

Les fonctions flashCapacity, flashPageSize, flashSectorSize et flashMaxFrequency permettent d'accéder à leurs caractéristiques :

```
uint32_t flashPageSize (const SPI_FLASH_TYPE inFlashType) ;
```

```
uint32_t flashSectorSize (const SPI_FLASH_TYPE inFlashType) ;
```

```
uint32_t flashMaxFrequency (const SPI_FLASH_TYPE inFlashType) ;
```

```
uint32_t flashCapacity (const SPI_FLASH_TYPE inFlashType) ;
```

| Flash                       | flashCapacity | flashPageSize | flashSectorSize | flashMaxFrequency |
|-----------------------------|---------------|---------------|-----------------|-------------------|
| SPI_FLASH_TYPE::SST26VF064B | 8 388 608     | 256           | 4 096           | 40 000 000        |
| SPI_FLASH_TYPE::IS25LP128   | 16 777 216    | 256           | 4 096           | 40 000 000        |

# Programmation de l'accès à une FLASH externe

Pour un exemple pratique, voir à [cette page](#) le croquis d'exemple 09-croquis-test-flash-externe.

① Déclarer une variable globale (ici `myFlash`, le nom est libre) qui précise l'emplacement utilisé et le type de la mémoire, par exemple :

```
static SPIFLASH myFlash (MySPI::spi2, SPI_FLASH_TYPE::SST26VF064B) ;
```

Les valeurs possibles du premier argument sont : `MySPI::spi1`, `MySPI::spi2`, `MySPI::spi3`, `MySPI::spi5`.  
Celles du second argument : `SPI_FLASH_TYPE::SST26VF064B`, `SPI_FLASH_TYPE::IS25LP128`.

② Dans la fonction `setup`, appeler :

```
myFlash.begin () ;
```

③ Dans la fonction `loop`, on peut appeler les méthodes de la classe `SPIFLASH`, en particulier :

- `flashRead`, pour lire la Flash ;
- `flashWrite`, pour l'écrire ;
- `flashEraseSector`, pour effacer un secteur ;
- `flashEraseChip`, pour effacer toute la Flash ;
- `flashIsBusy`, pour savoir si une écriture ou un effacement est en cours.

# Lecture d'une FLASH externe : flashRead

La méthode eepromRead de la classe SPIFLASH réalise une lecture d'une flash :

```
void flashRead (const uint32_t inAddress,  
                uint8_t outBuffer[],  
                const uint32_t inLength);
```

Les arguments sont :

- **inAddress**, l'adresse de lecture en flash ;
- **outBuffer**, un tableau d'octets, d'une taille d'au moins **inLength** éléments, qui reçoit les octets lus ;
- **inLength**, le nombre d'octets à lire.

Notes :

- si une écriture ou un effacement est en cours, la méthode attend qu'elle soit terminée avant d'effectuer la lecture ;
- le débordement n'est pas testé : aussi, l'octet effectivement lu est celui d'adresse modulo la taille de la mémoire.

# Écriture d'une FLASH externe : flashWrite

La méthode `flashWrite` de la classe `SPIFLASH` réalise une écriture d'une flash :

```
void flashWrite (const uint32_t inAddress,  
                 const uint8_t inBuffer[],  
                 const uint32_t inLength) ;
```

Les arguments sont :

- `inAddress`, l'adresse d'écriture en flash ;
- `inBuffer`, un tableau d'octets, d'une taille d'au moins `inLength` éléments, qui contient les octets à écrire ;
- `inLength`, le nombre d'octets à écrire.

Notes :

- si une écriture ou un effacement est en cours, la méthode attend qu'elle soit terminée avant d'effectuer l'écriture ;
- le débordement n'est pas testé : aussi, l'octet effectivement écrit est celui d'adresse modulo la taille de la mémoire ;
- techniquement, une seule page peut être écrite à la fois ; si les données sont à écrire sur plusieurs pages, la méthode scinde automatiquement les données et effectue plusieurs écritures consécutives ; la durée d'exécution s'en trouve rallongée, puisqu'une écriture ne peut être lancée qu'une fois la précédente terminée.

# Effacement d'un secteur d'une FLASH externe : flashEraseSector

La méthode `flashEraseSector` de la classe `SPIFLASH` réalise l'effacement d'un secteur d'une flash (c'est-à-dire met tous les bits du secteur à 1) :

```
void flashEraseSector (const uint32_t inSector) ;
```

L'argument `inSector` est le numéro du secteur à effacer. Le nombre de secteurs d'une flash est donné par :

`flashCapacity (flashType) / flashSectorSize (flashType)`

Soit pour une SST26VF064B :  $\frac{8\ 388\ 608}{4\ 096} = 2\ 048$ . Les numéros de secteurs valides pour ce composant sont entre 0 et 2 047 compris.

Notes :

- si une écriture ou un effacement est en cours, la méthode attend qu'elle soit terminée avant d'effectuer l'effacement du secteur ;
- le débordement n'est pas testé : aussi, si le numéro du secteur trop grand, le secteur effectivement effacé est celui de numéro modulo le nombre de secteurs de la mémoire.

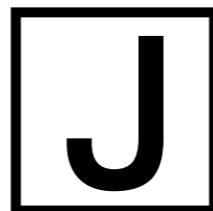
# Effacement complet d'une FLASH externe : flashEraseChip

La méthode `flashEraseChip` de la classe `SPIFLASH` réalise l'effacement complet d'une flash (c'est-à-dire met tous ses bits à 1) :

```
void flashEraseChip (void) ;
```

Notes :

- si une écriture ou un effacement est en cours, la méthode attend qu'elle soit terminée avant d'effectuer l'effacement de la mémoire.



# Croquis d'exemple

# Les croquis d'exemple

Les croquis d'exemple sont dans le répertoire `croquis-exemple-stm32duino` :

- `01-afficheur-lcd` ;
- `02-ne-pas-utiliser-delay` ;
- `03-croquis-test-es-logiques` ;
- `04-croquis-test-es-analogiques` ;
- `05-croquis-test-sorties-tor` ;
- `06-croquis-test-ram-externe` ;
- `07-croquis-test-eeprom-externe` ;
- `08-croquis-test-eeprom-storage` ;
- `09-croquis-test-flash-externe`.

**Note.** pensez à effectuer les opérations précédemment décrites :

- ponts de soudure marqués **SB14** et **SB15** au dos de la carte **NUCLEO-H743ZI2** ([voir à cette page](#)) ;
- modifier STM32Duino pour que PA6 soit reconnue comme une entrée analogique ([voir à cette page](#)) ;
- modifier STM32Duino pour que PG9 soit reconnue comme une sortie logique ([voir à cette page](#)).

# Composition des croquis

Tout le code de gestion de la carte `croquis-exemple-stm32duino` est contenu dans deux fichiers :

- `STM32H743-configuration-lheea.cpp`
- `STM32H743-configuration-lheea.h`

Pour les utiliser dans un croquis, il y a 2 possibilités :

- ① Les inclure simplement dans le répertoire du croquis, à côté du fichier `.ino`.
- ② Inclure les *liens symboliques* vers ces fichiers. C'est ce qui est fait pour tous les croquis d'exemple présentés dans la suite. Pour cela, **ne pas utiliser** *Créer un alias* du menu *Fichier* du Finder. Il faut exécuter dans le terminal :

```
cd répertoire_du_croquis
ln -s chemin/*.cpp .
ln -s chemin/*.h .
```

Attention, cette technique ne fonctionne pas sous Windows (Windows ne connaît pas les liens symboliques).

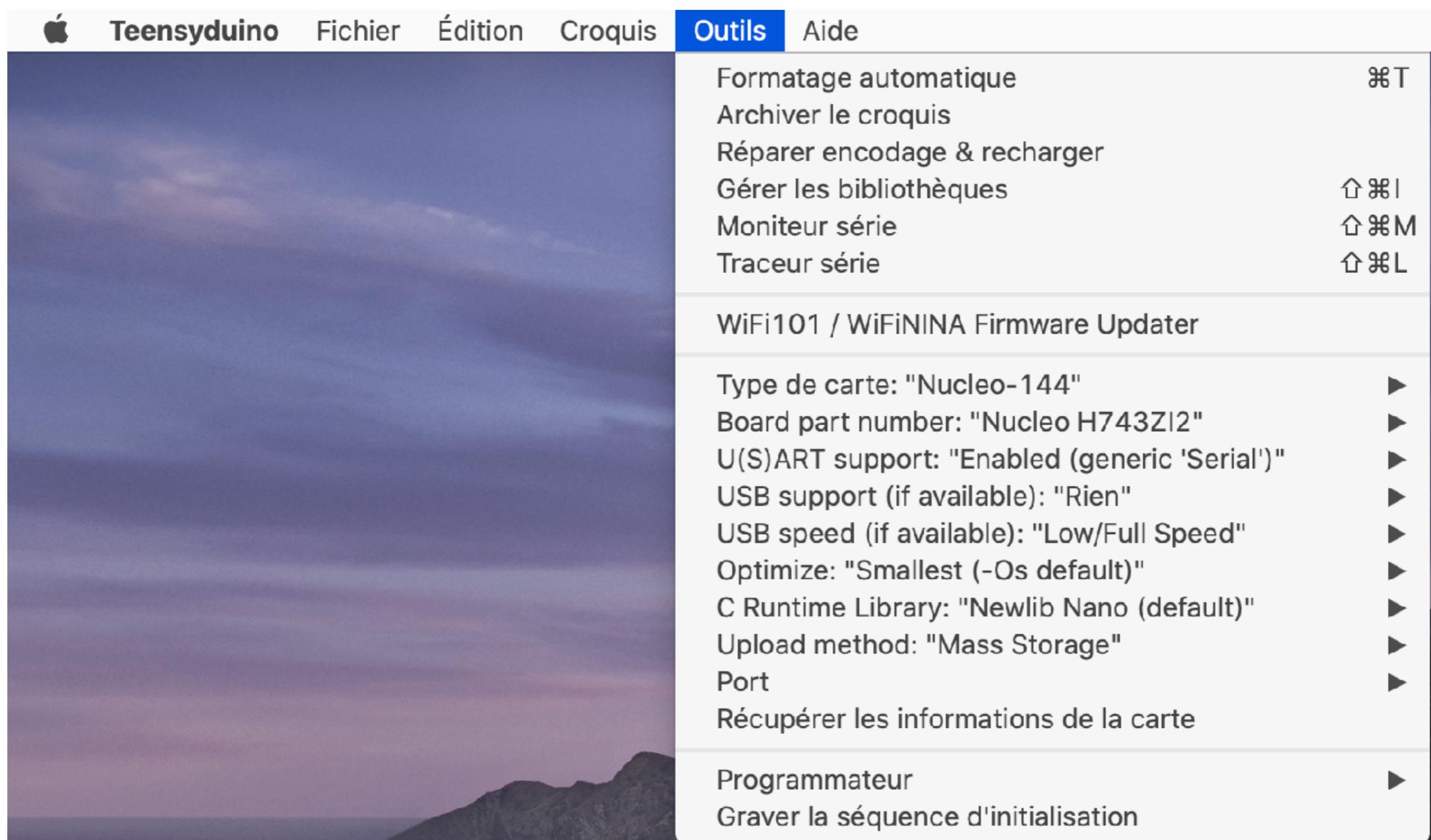
Dans la fonction **setup** du croquis, appeler la fonction **configurerCarteH743LHEEA**.

# Sélection de la carte

La sélection de la carte doit être :

Type de carte : **Nucleo-144** ;

Board part number : **Nucleo H743ZI2** (**Attention, ne pas sélectionner Nucleo H743ZI, c'est une autre carte**).



# Écriture d'un croquis

Pour écrire un croquis pour la carte asservissement, il faut suivre les règles suivantes dans le fichier `.ino` :

- ① Inclure `STM32H743-configuration-lheea.h` au début du fichier :

```
#include "STM32H743-configuration-lheea.h"
```

- ② Dans la fonction `setup`, appeler une et une seule fois la fonction `configurerCarteH743LHEEA` :

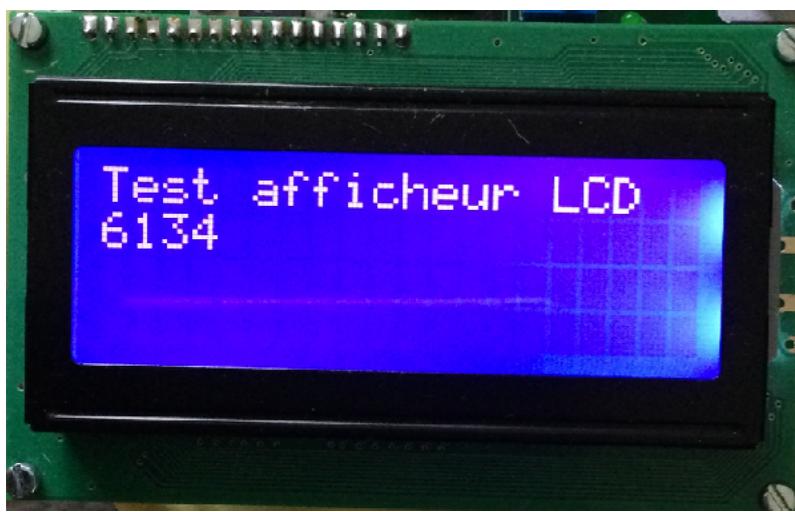
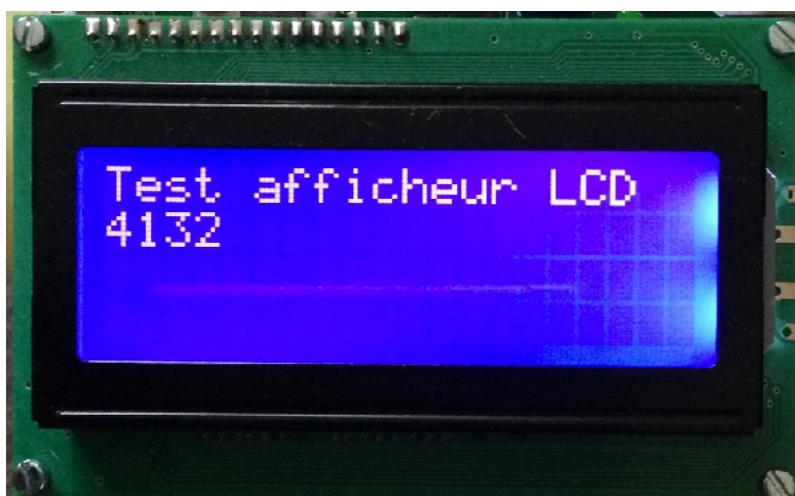
```
void setup () {  
    configurerCarteH743LHEEA () ;  
    ...  
}
```

- ③ Ensuite, toutes les fonctions citées dans `STM32H743-configuration-lheea.h` peuvent être appelées. L'objet des croquis d'exemple qui suivent est d'illustrer leur utilisation.

# Croquis 01-afficheur-lcd

Ce croquis illustre l'utilisation de l'afficheur LCD.

Il montre aussi que la fonction **delay** ne permet pas d'avoir des exécutions vraiment périodiques (le croquis suivant va y remédier) : alors que le délai est 1000, la période est supérieure à cette valeur.



# Croquis 02-ne-pas-utiliser-delay

Utiliser `delay` est simple, mais cela pose beaucoup de problèmes que l'on ne peut pas résoudre facilement lorsque l'on a besoin de gérer des actions avec des périodes différentes.

Le plus simple, pour chaque action périodique, est de maintenir une variable (`gInstantClignotementXXXX` et `gInstantAffichage` dans ce croquis d'exemple) qui détient la date de la prochaine exécution de l'action.

De cette façon :

- si la date d'exécution n'est pas atteinte, le processeur n'est pas immobilisé ;
- on peut facilement fixer indépendamment la période de chaque action.

Le croquis définit 4 actions périodiques :

- clignotement de la led verte de la carte Nucleo ;
- clignotement de la led jaune de la carte Nucleo ;
- clignotement de la led rouge de la carte Nucleo ;
- affichage de la date courante (noter que la dérive observée avec le croquis précédent a disparu).

**Note :** `millis()` retourne la date courante (nombre de millisecondes depuis le démarrage) dans un `uint32_t` (entier non signé de 32 bits). Il y a débordement au bout de  $2^{32}-1$  millisecondes, soit plus de 49 jours. Après débordement, le code ne fonctionne plus.

# Croquis 02-ne-pas-utiliser-delay

Utiliser `delay` est simple, mais cela pose beaucoup de problèmes que l'on ne peut pas résoudre facilement lorsque l'on a besoin de gérer des actions avec des périodes différentes.

Le plus simple, pour chaque action périodique, est de maintenir une variable (`gInstantClignotementXXXX` et `gInstantAffichage` dans ce croquis d'exemple) qui détient la date de la prochaine exécution de l'action.

De cette façon :

- si la date d'exécution n'est pas atteinte, le processeur n'est pas immobilisé ;
- on peut facilement fixer indépendamment la période de chaque action.

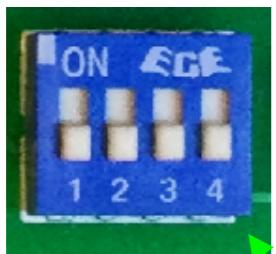
Le croquis définit 4 actions périodiques :

- clignotement de la led verte de la carte Nucleo ;
- clignotement de la led jaune de la carte Nucleo ;
- clignotement de la led rouge de la carte Nucleo ;
- affichage de la date courante (noter que la dérive observée avec le croquis précédent a disparu).

**Note :** `millis()` retourne la date courante (nombre de millisecondes depuis le démarrage) dans un `uint32_t` (entier non signé de 32 bits). Il y a débordement au bout de  $2^{32}-1$  millisecondes, soit plus de 49 jours. Après débordement, le code ne fonctionne plus.

# Croquis 03-croquis-test-es-logiques (1/2)

Ce croquis permet de tester les E/S logiques, et illustre l'utilisation des fonctions correspondantes.



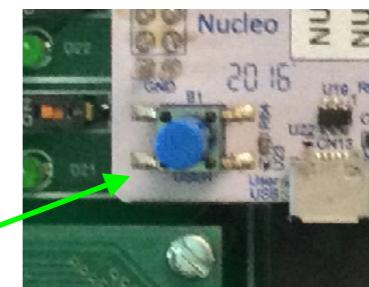
Interrupteur DIL : OFF -> 1, ON -> 0



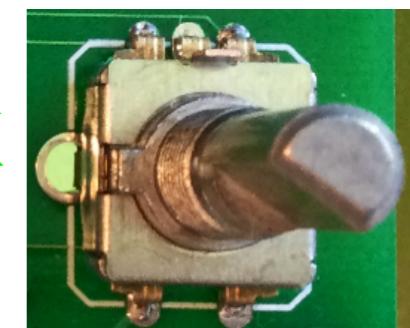
Poussoirs : relâché -> 1, appuyé -> 0



Poussoir bleu carte NUCLEO : relâché -> 0, appuyé -> 1



Poussoir de l'encodeur : relâché -> 1, appuyé -> 0



Rotation de l'encodeur : La gamme de l'encodeur est fixée à [-10, 10], par un appel à fixerGammeEncodeur dans la fonction setup (voir page suivante).

# Croquis 03-croquis-test-es-logiques (2/2)

**Utilisation de l'encodeur.** Utiliser l'encodeur se fait en deux étapes :

- dans la fonction `setup`, appeler `fixerGammeEncodeur` pour fixer la plage des valeurs ;
- ensuite, appeler quand on veut `valeurEncodeur` pour récupérer la valeur de l'encodeur.

La fonction `fixerGammeEncodeur` a deux arguments :

`fixerGammeEncodeur (borneInf, borneSup)`

où :

- *borneInf* est la borne inférieure de la plage des valeurs ;
- *borneSup* est la borne supérieure de la plage des valeurs.

Des valeur négatives sont acceptées (les arguments formels sont de type `int32_t`). Il faut appeler cette fonction avec  $\text{borneInf} \leq \text{borneSup}$ .

Si  $\text{borneInf} == \text{borneSup}$ , la fonction `valeurEncodeur` renvoie toujours la valeur commune, indépendamment de la rotation de l'encodeur.

Si  $\text{borneInf} < \text{borneSup}$ , la fonction `valeurEncodeur` renvoie toujours une valeur entière dans l'intervalle  $[\text{borneInf}, \text{borneSup}]$  :

- une rotation dans le sens trigonométrique décrémente la valeur renvoyée, jusqu'à saturer à *borneInf* ;
- une rotation dans le sens des aiguilles d'une montre incrémente la valeur renvoyée, jusqu'à saturer à *borneSup*.

Initialement, par défaut,  $\text{borneInf} == \text{borneSup} == 0$ . La fonction `valeurEncodeur` renvoie alors toujours 0.

# Croquis 04-croquis-test-es-analogiques

Ce croquis teste la sortie analogique et les quatre entrées logiques :

- la gamme de l'encodeur est fixée à [0, 255] ;
- la valeur de l'encodeur est envoyée sur la sortie analogique ;
- toutes les secondes, les quatre entrées analogiques sont lues et affichées.



Le nombre après SA (ici 100) est directement la valeur issue de l'encodeur rotatif. Cette valeur est envoyée sur la sortie analogique. La valeur théorique de la sortie DIRECTE est affichée, elle vaut  $100 * 3,3V / 255 = 1,29V$  ; le multi-mètre affiche une valeur voisine : 1,30V.

Les quatre entrées analogiques sont affichées sur les deux dernières lignes. Les acquisitions sont sur 12 bits, la gamme des valeurs est donc [0, 4095].

Dans l'affichage ci-contre, les entrées analogiques 0, 1 et 3 ne sont pas connectées : leur valeur théorique est 0.



L'entrée analogique 2 est connectée à la sortie analogique DIRECTE. Le potentiomètre de réglage de l'entrée analogique 2 est réglé de façon à réaliser un gain de 1 (il est tourné au maximum dans le sens trigonométrique). La résolution de la sortie analogique est de 8 bits, celle de l'entrée 12 bits, la valeur théorique qui devrait être affichée pour EA2 est  $2^{12-8}$  fois la valeur de l'encodeur, soit 1600.

# Croquis 05-croquis-test-sorties-tor

Ce croquis effectue indéfiniment l'activation pendant une seconde des sorties TOR, dans l'ordre  
TOR0, TOR1, ... TOR9, TOR0, ...

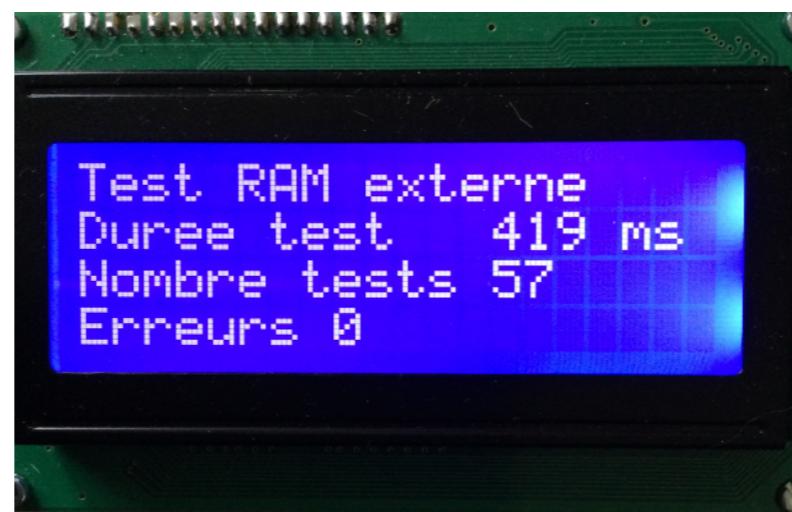
# Croquis 06-croquis-test-ram-externe

Ce croquis effectue chaque seconde un test exhaustif de l'accès à la RAM externe.

le code du test est la fonction `testRamExterne`. Celle-ci écrit une valeur particulière dans chaque octet de la RAM externe (dont la taille est  $4 * 512 * 1024$  octets), puis relie cette valeur et la vérifie.

Le nombre cumulé d'erreurs est affiché à la dernière ligne. Il doit toujours être nul.

La durée d'un test est environ 419 ms. ceci permet de calculer le temps d'une lecture ou d'une écriture d'un octet en RAM externe :  $419 \text{ ms} / (2 * 4 * 512 * 1024) \approx 100 \text{ ns}$ .



# Croquis 07-croquis-test-eeprom-externe

Ce croquis effectue chaque seconde une lecture de 6 octets en EEPROM externe, et, à la demande, une écriture de 6 octets.

L'adresse de lecture et d'écriture est défini par la constante `ADRESSE_EN_EEPROM`.

L'affichage :

- deuxième ligne : nombre de lectures, à raison d'une lecture par seconde ;
- troisième ligne : les 6 octets lus ;
- quatrième ligne : la durée de la dernière exécution de `eepromWrite`.

Pour lancer une écriture : appuyer sur le poussoir P0 (blanc) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-0 (verte) s'allume ; relâcher alors le poussoir. Les 6 valeurs écrites sont des valeurs consécutives, obtenues à partir de la fonction `millis()`.



# Croquis 08-croquis-test-eeprom-storage (1/2)

Ce croquis illustre l'utilisation de la classe générique VarInEEPROM. Quatre variables de ce type sont utilisées dans le croquis :

```
static VarInEEPROM <uint32_t> gVar0 (mySPIEEPROM, 0) ;  
static VarInEEPROM <uint32_t> gVar1 (mySPIEEPROM, 4) ;  
static VarInEEPROM <uint32_t> gVar2 (mySPIEEPROM, 8) ;  
static VarInEEPROM <uint32_t> gVar3 (mySPIEEPROM, 126) ;
```

La variable gVar3 est à l'adresse 126, et elle réside à cheval sur deux pages consécutives.

L'affichage :

- deuxième ligne : nombre de lectures, à raison d'une lecture par seconde ;
- troisième ligne : à gauche, la valeur de gVar0, à droite la valeur de gVar1 ;
- quatrième ligne : à gauche, la valeur de gVar2, à droite la valeur de gVar3.

Comme chaque variable est un `uint32_t`, sa valeur maximum est  $2^{32}-1$ , soit 4294967295, qui s'écrit sur 10 caractères. Comme une ligne de l'afficheur contient 20 caractères, les valeurs sur 10 chiffres de gVar0 et gVar2 ne sont pas séparées des valeurs de gVar1 et gVar3 par des espaces.



# Croquis 08-croquis-test-eeprom-storage (2/2)

**Pour effectuer une écriture de gVar0 :** appuyer sur le poussoir P0 (blanc) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-0 (verte) s'allume ; relâcher alors le poussoir. La valeur écrite est celle du compteur d'itérations.

**Pour effectuer une écriture de gVar1 :** appuyer sur le poussoir P1 (rose) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-1 (jaune) s'allume ; relâcher alors le poussoir. La valeur écrite est celle du compteur d'itérations.

**Pour effectuer une écriture de gVar2 :** appuyer sur le poussoir P2 (jaune) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-2 (rouge) s'allume ; relâcher alors le poussoir. La valeur écrite est celle du compteur d'itérations.

**Pour effectuer une écriture de gVar3 :** appuyer sur le poussoir P3 (rouge) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-0 (verte) et LED-1 (jaune) s'allument en même temps ; relâcher alors le poussoir. La valeur écrite est celle du compteur d'itérations.



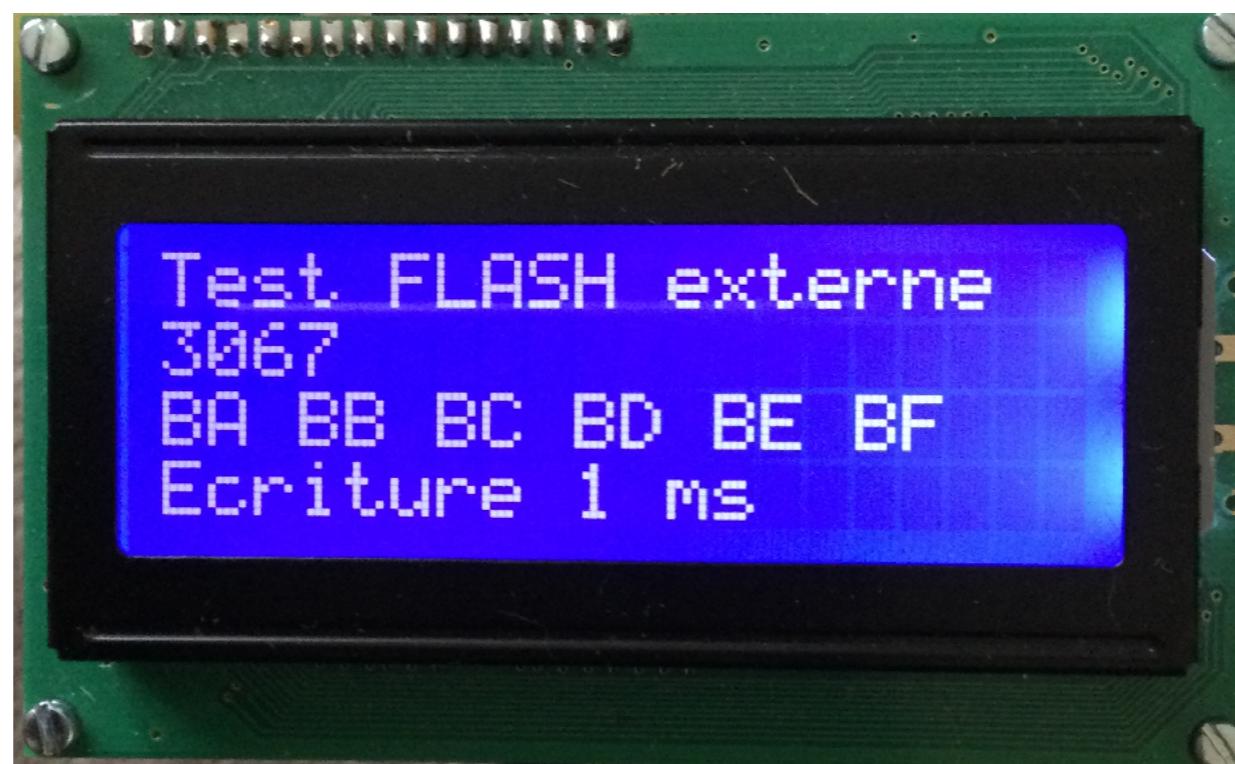
# Croquis 09-croquis-test-flash-externe (1/2)

Ce croquis effectue chaque seconde une lecture de 6 octets en FLASH externe, et, à la demande, une écriture de 6 octets.

L'adresse de lecture et d'écriture est défini par la constante ADRESSE\_EN\_FLASH, et vaut 217 055.

L'affichage :

- deuxième ligne : nombre de lectures, à raison d'une lecture par seconde ;
- troisième ligne : les 6 octets lus ;
- quatrième ligne : la durée de la dernière exécution de flashWrite, flashEraseSector, ou flashEraseChip.

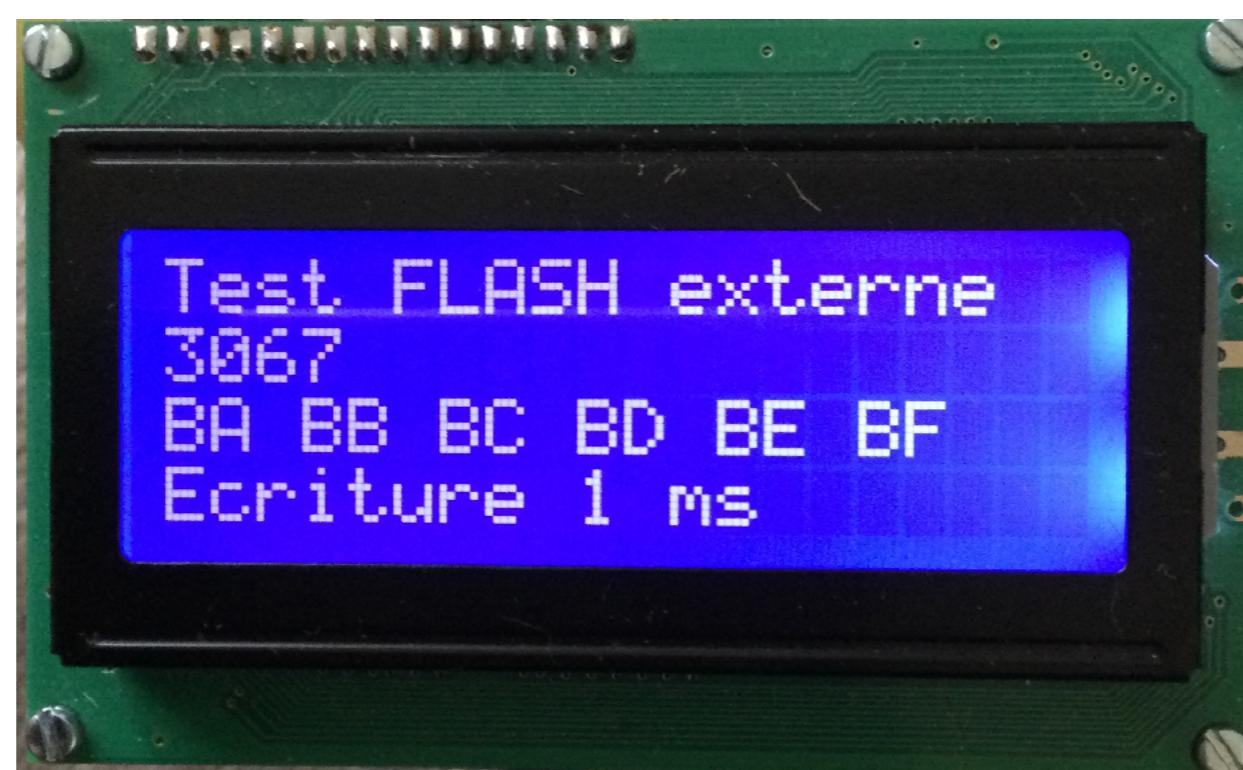


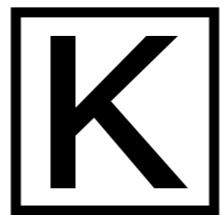
# Croquis 09-croquis-test-flash-externe (2/2)

**Pour lancer une écriture :** appuyer sur le poussoir P0 (blanc) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-0 (verte) s'allume ; relâcher alors le poussoir. Les 6 valeurs écrites sont des valeurs consécutives, obtenues à partir du compteur d'itérations. La dernière ligne affiche la durée de l'opération.

**Pour lancer l'effacement du secteur :** appuyer sur le poussoir P1 (rose) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-1 (jaune) s'allume ; relâcher alors le poussoir. Le ou les secteurs concernés sont effacés, la dernière ligne affiche la durée de l'opération.

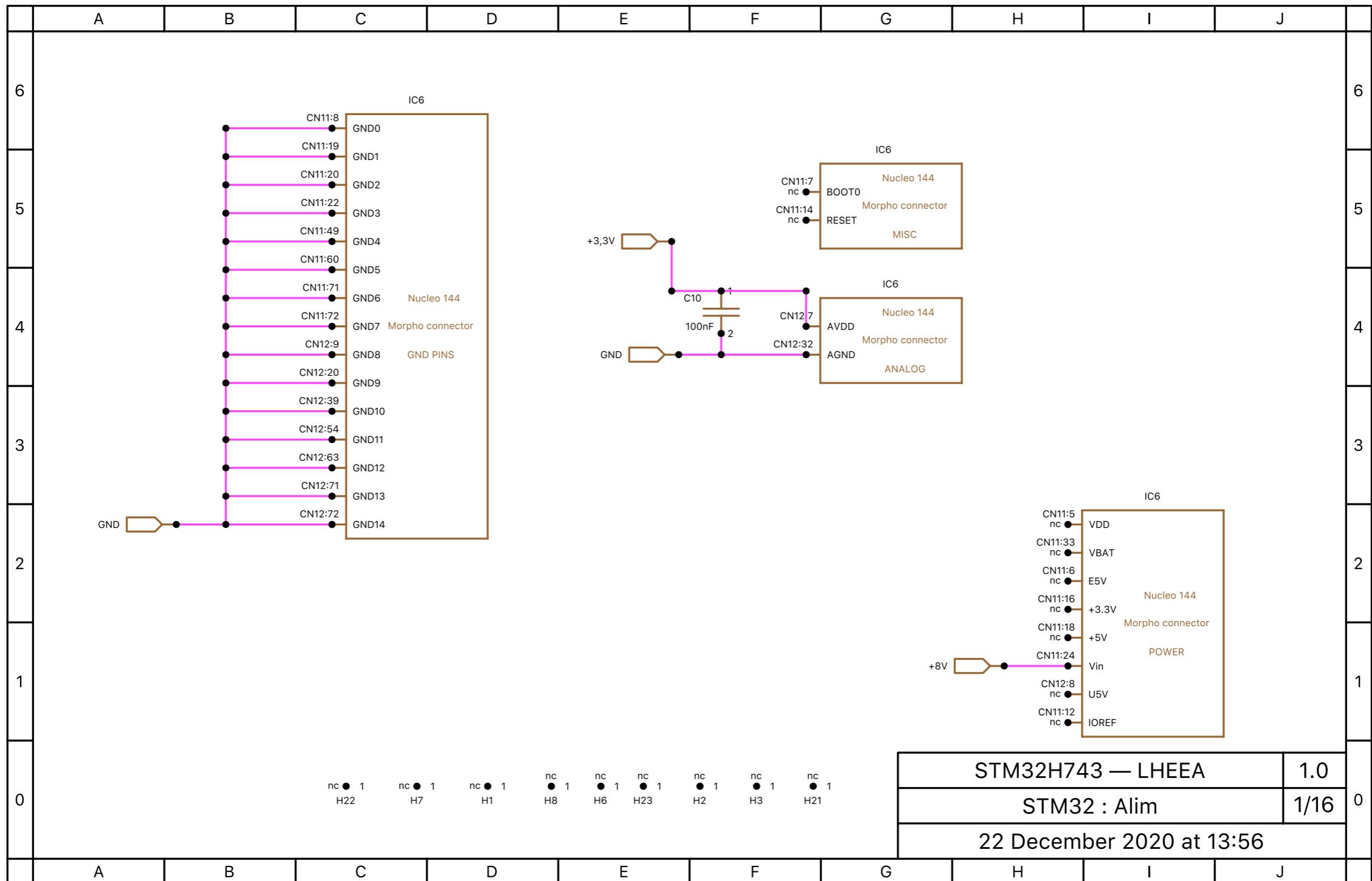
**Pour lancer l'effacement complet de la mémoire :** appuyer sur le poussoir P2 (jaune) ; celui-ci est échantillonné à chaque seconde. Quand la demande est prise compte, la led LED-2 (rouge) s'allume ; relâcher alors le poussoir. La mémoire flash est complètement effacée, et la dernière ligne affiche la durée de l'opération.



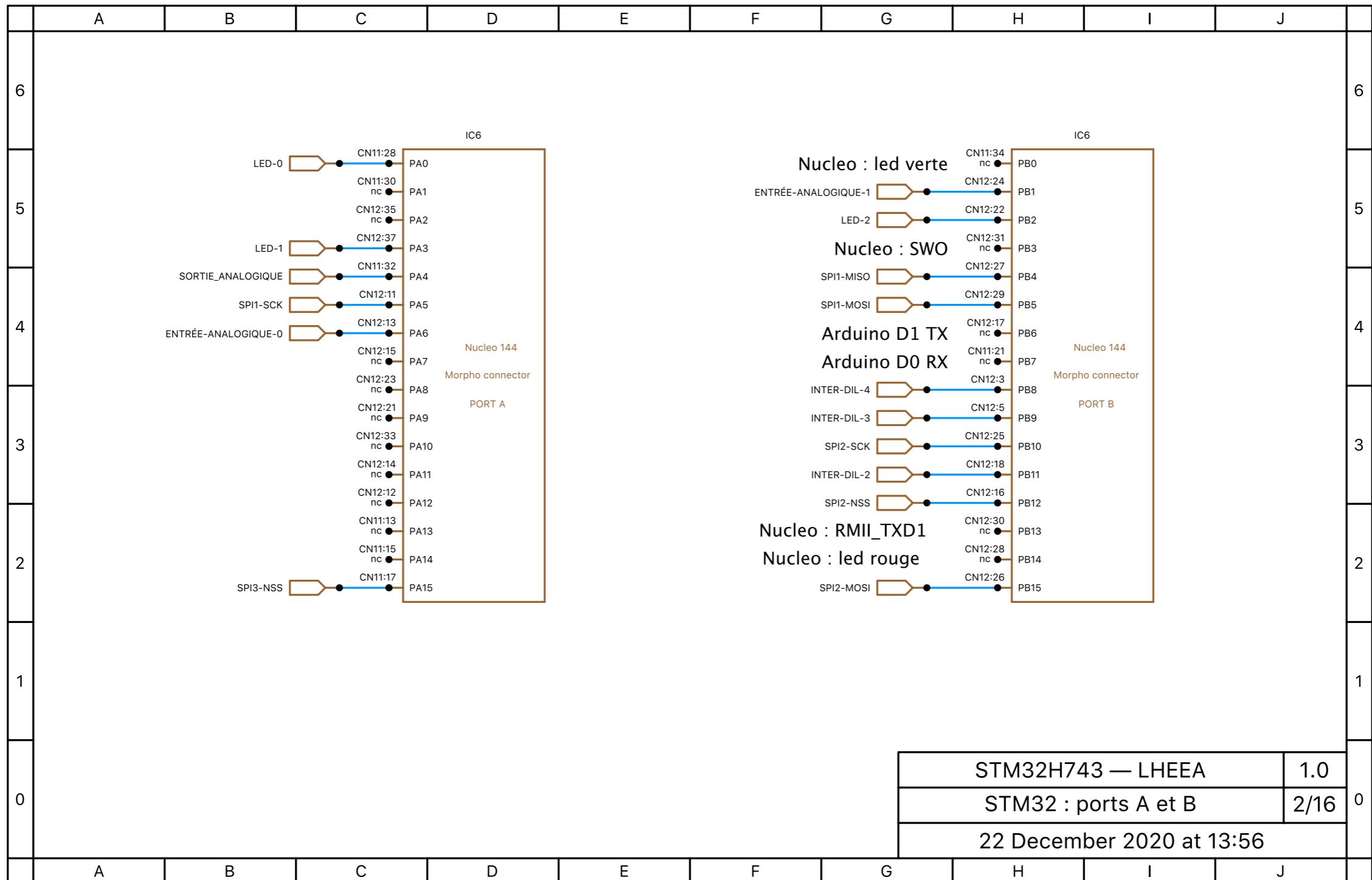


# Plan de la carte

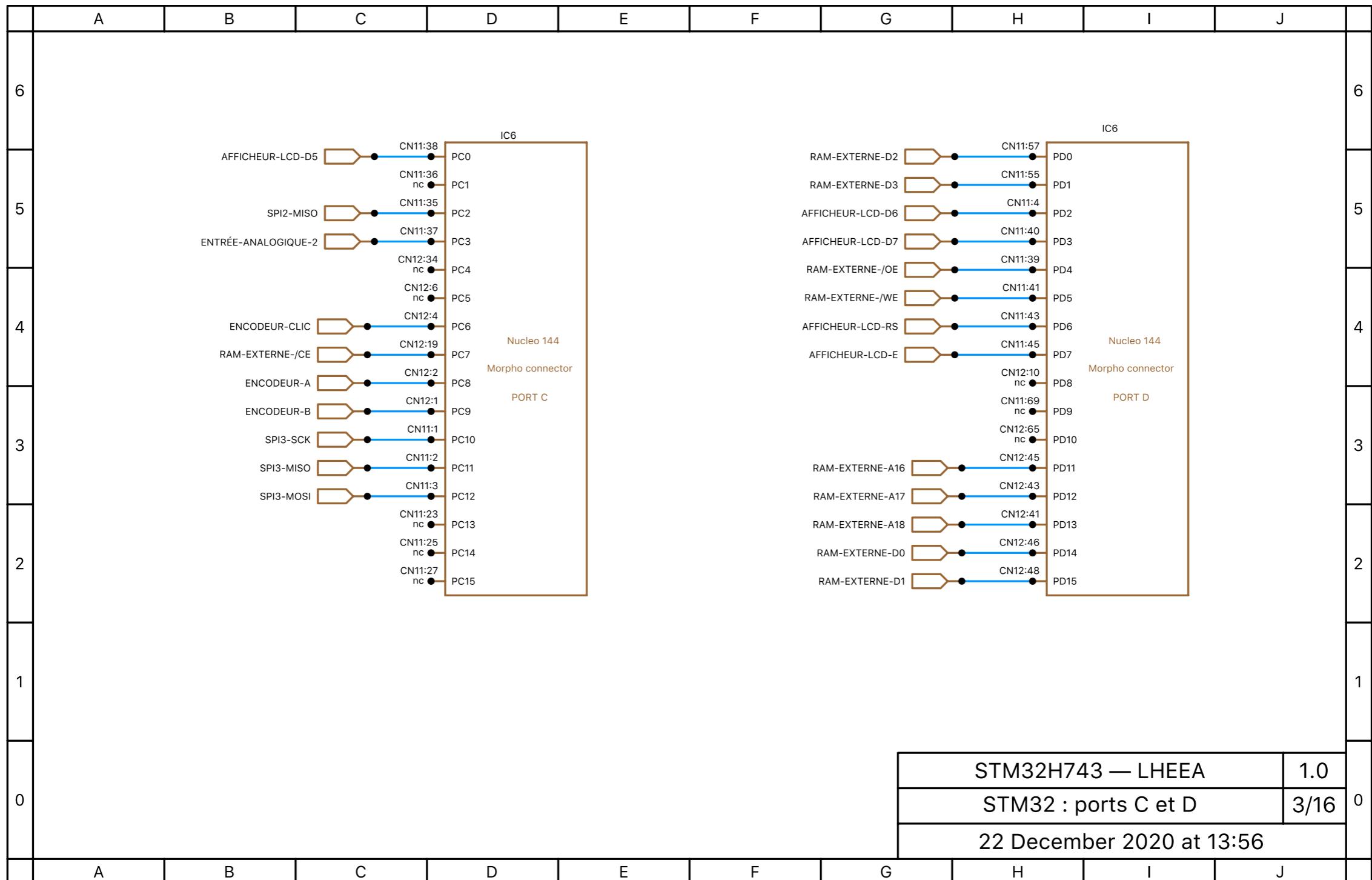
# Plan de la carte (1/16)



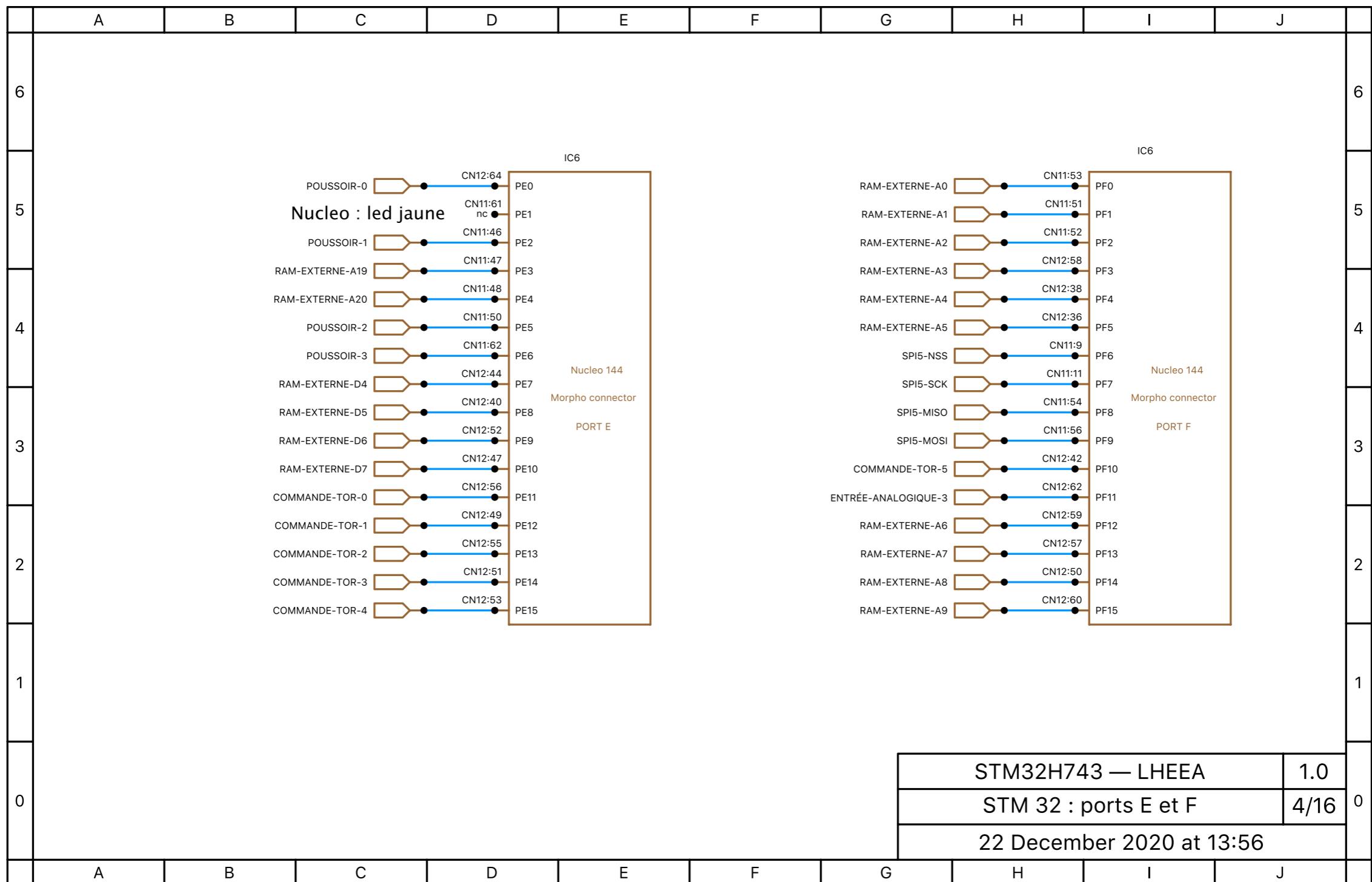
# Plan de la carte (2/16)



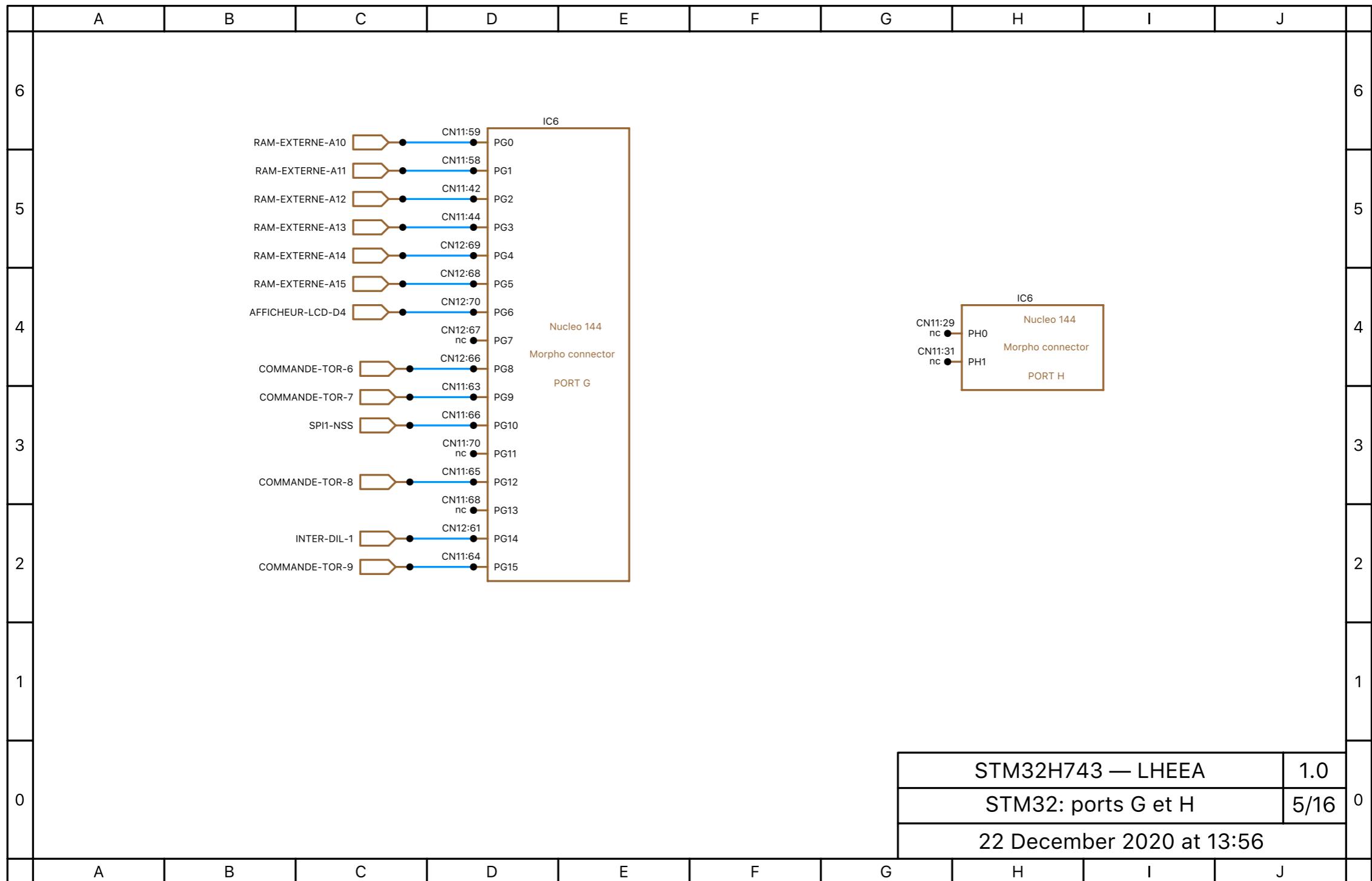
# Plan de la carte (3/16)



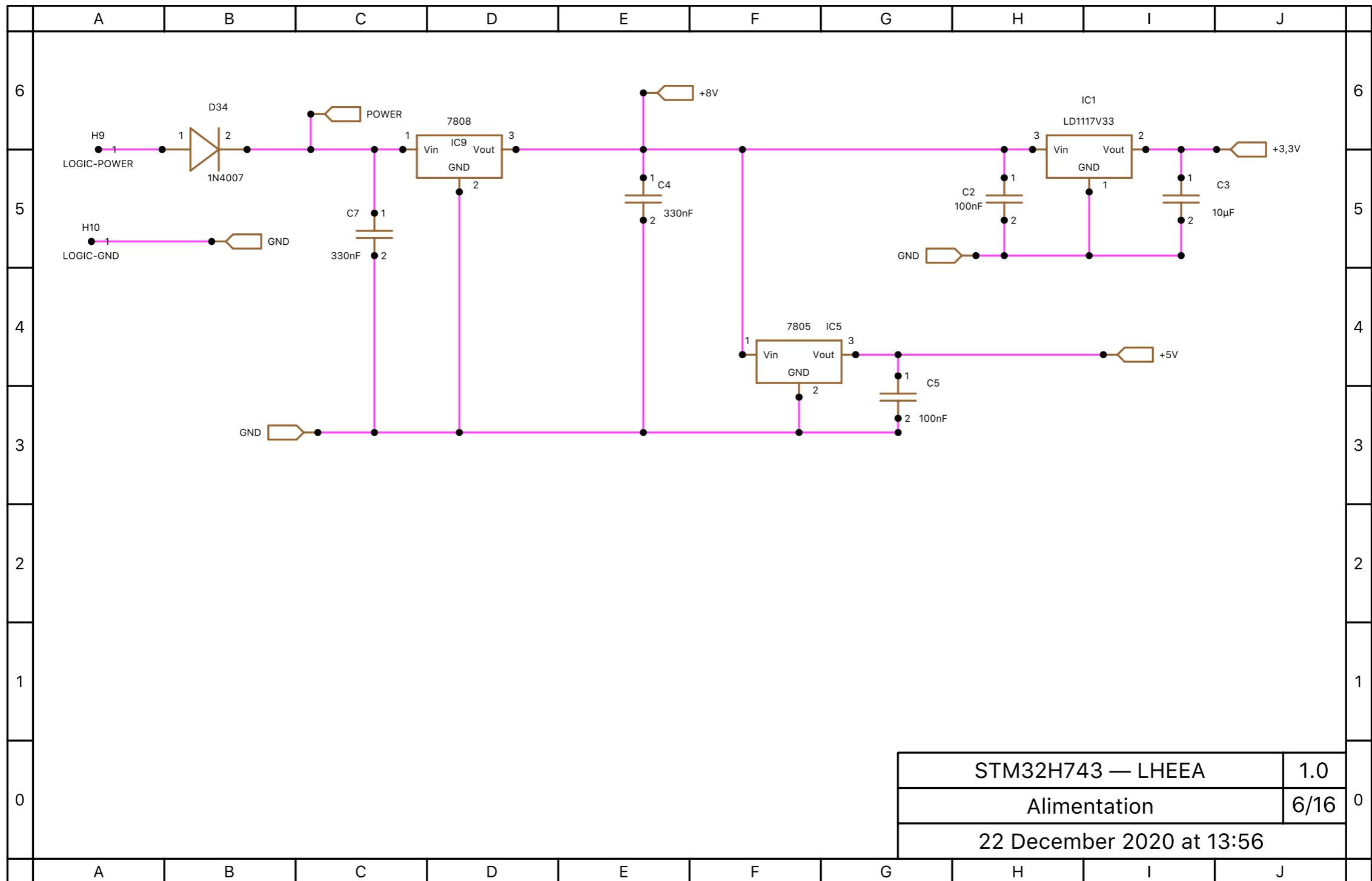
# Plan de la carte (4/16)



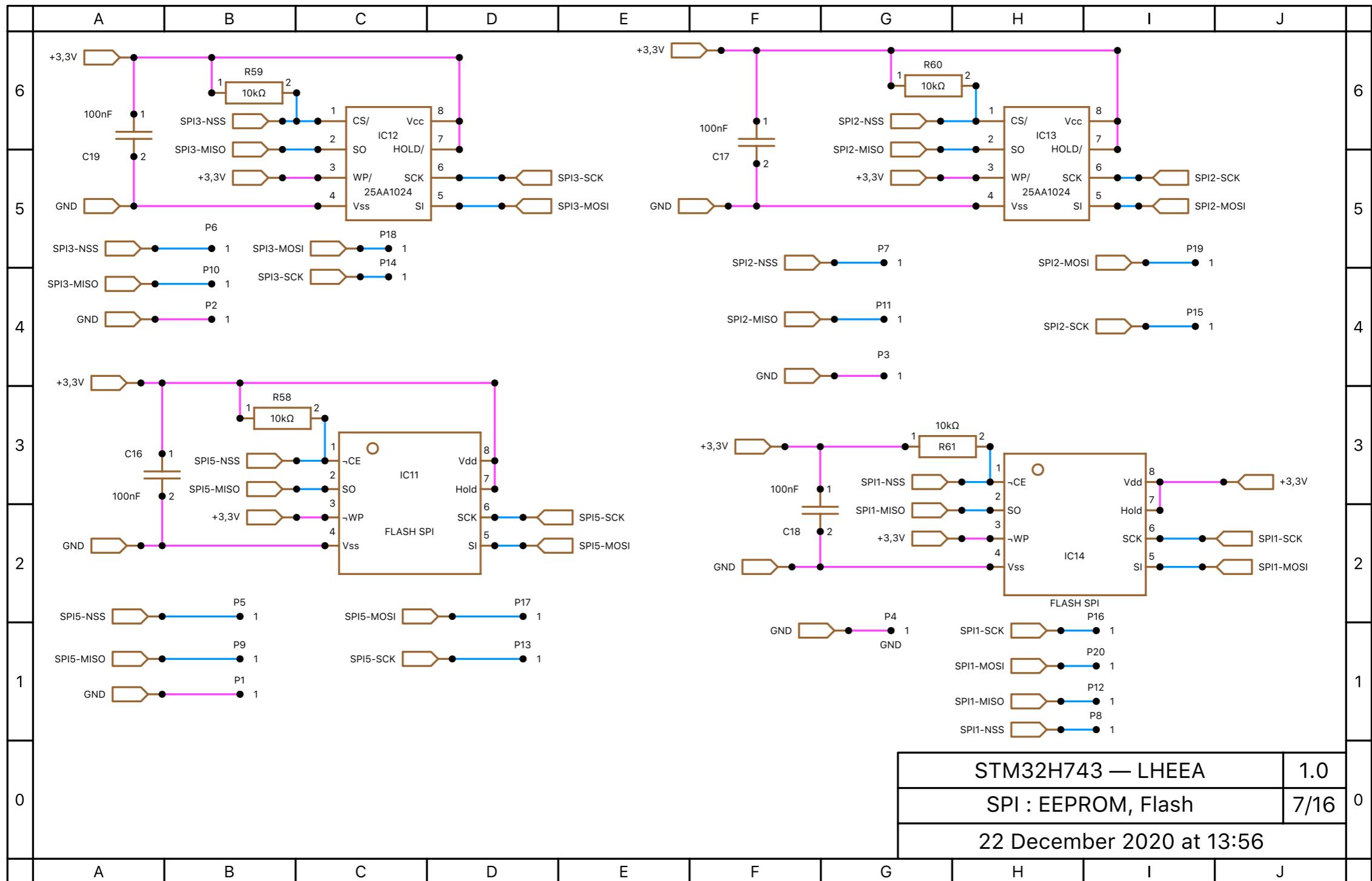
# Plan de la carte (5/16)



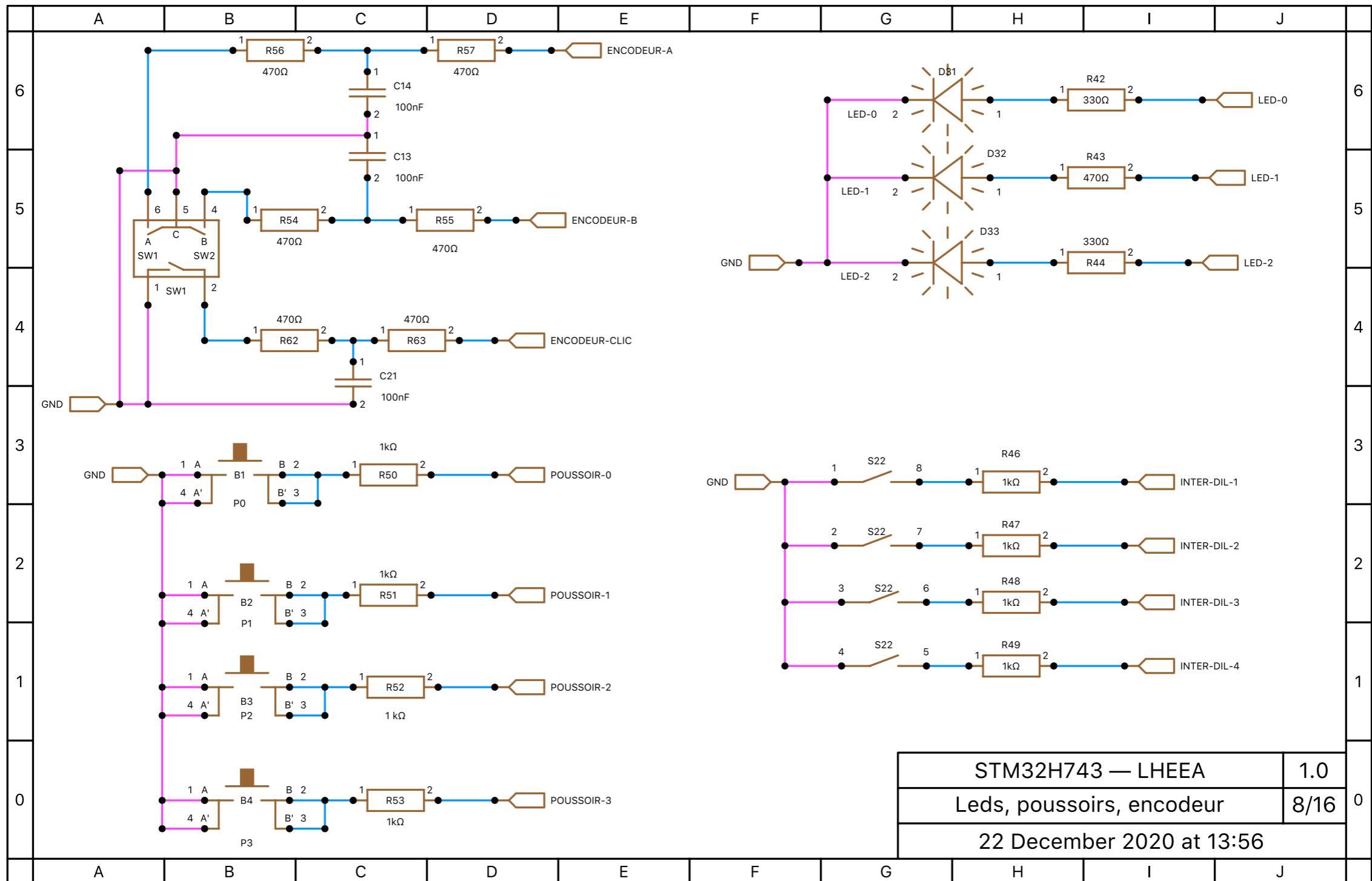
# Plan de la carte (6/16)



# Plan de la carte (7/16)



# Plan de la carte (8/16)



STM32H743 — LHEEA

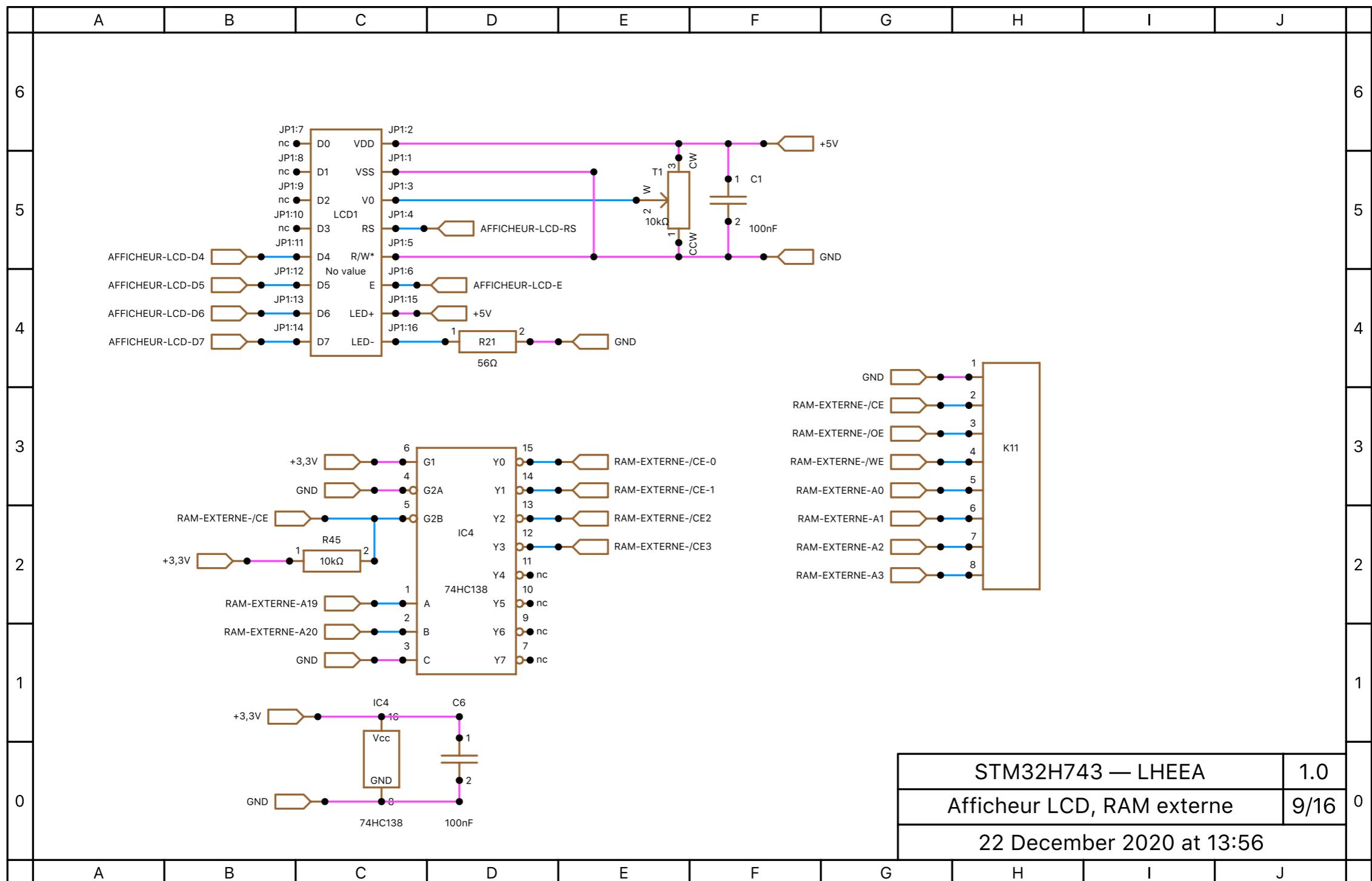
Leds, poussoirs, encodeur

22 December 2020 at 13:56

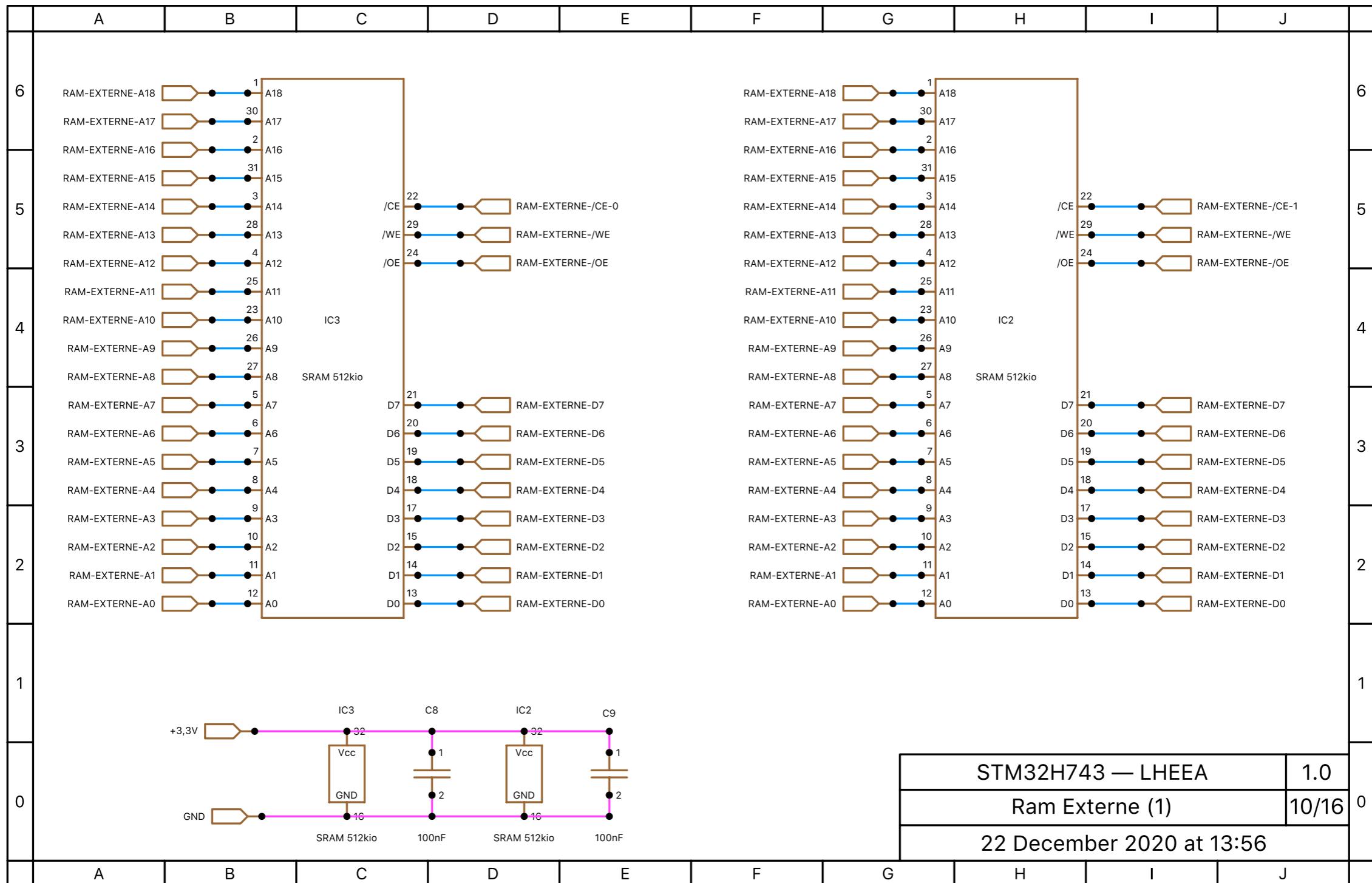
1.0

8/16

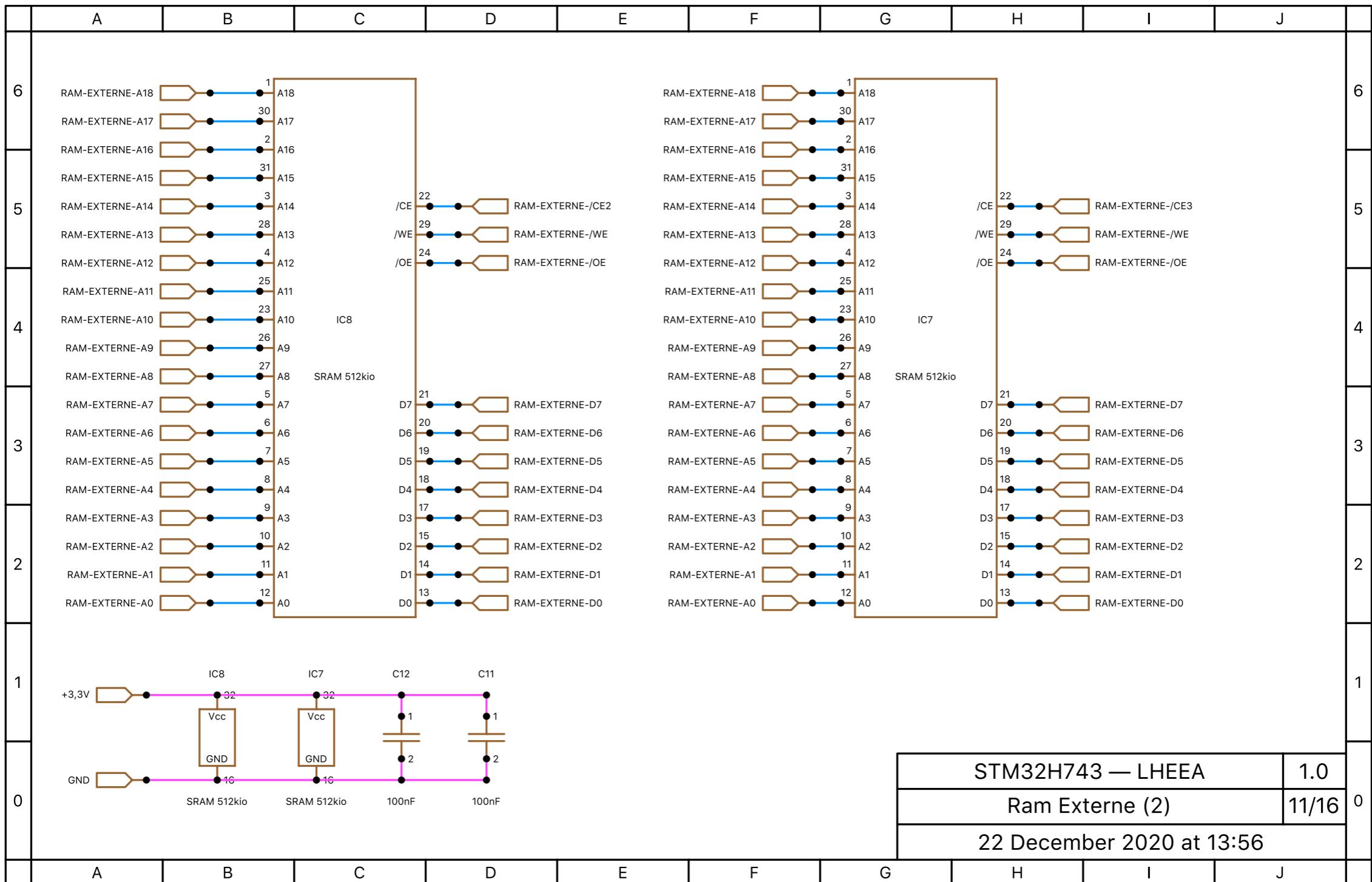
# Plan de la carte (9/16)



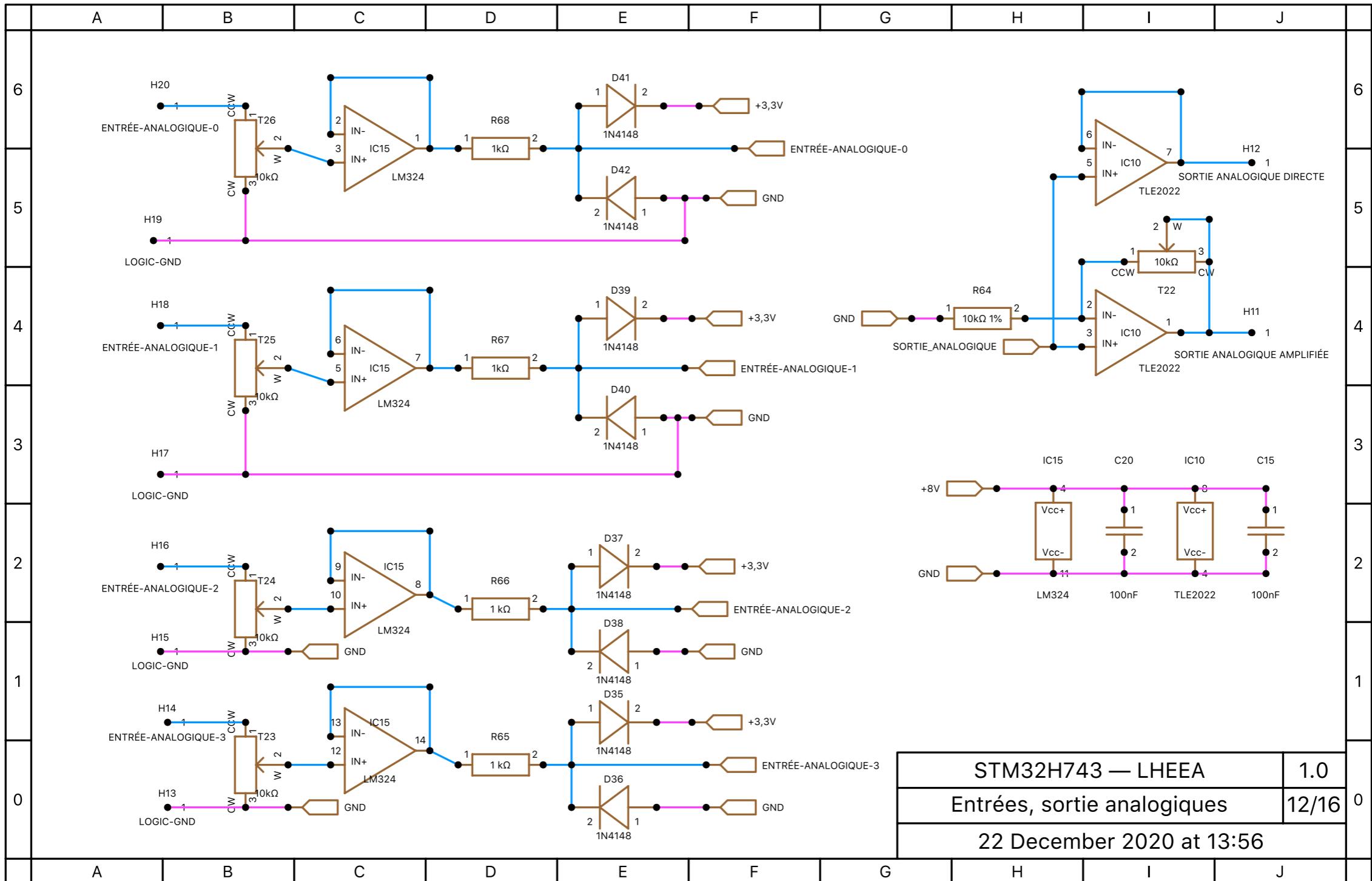
# Plan de la carte (10/16)



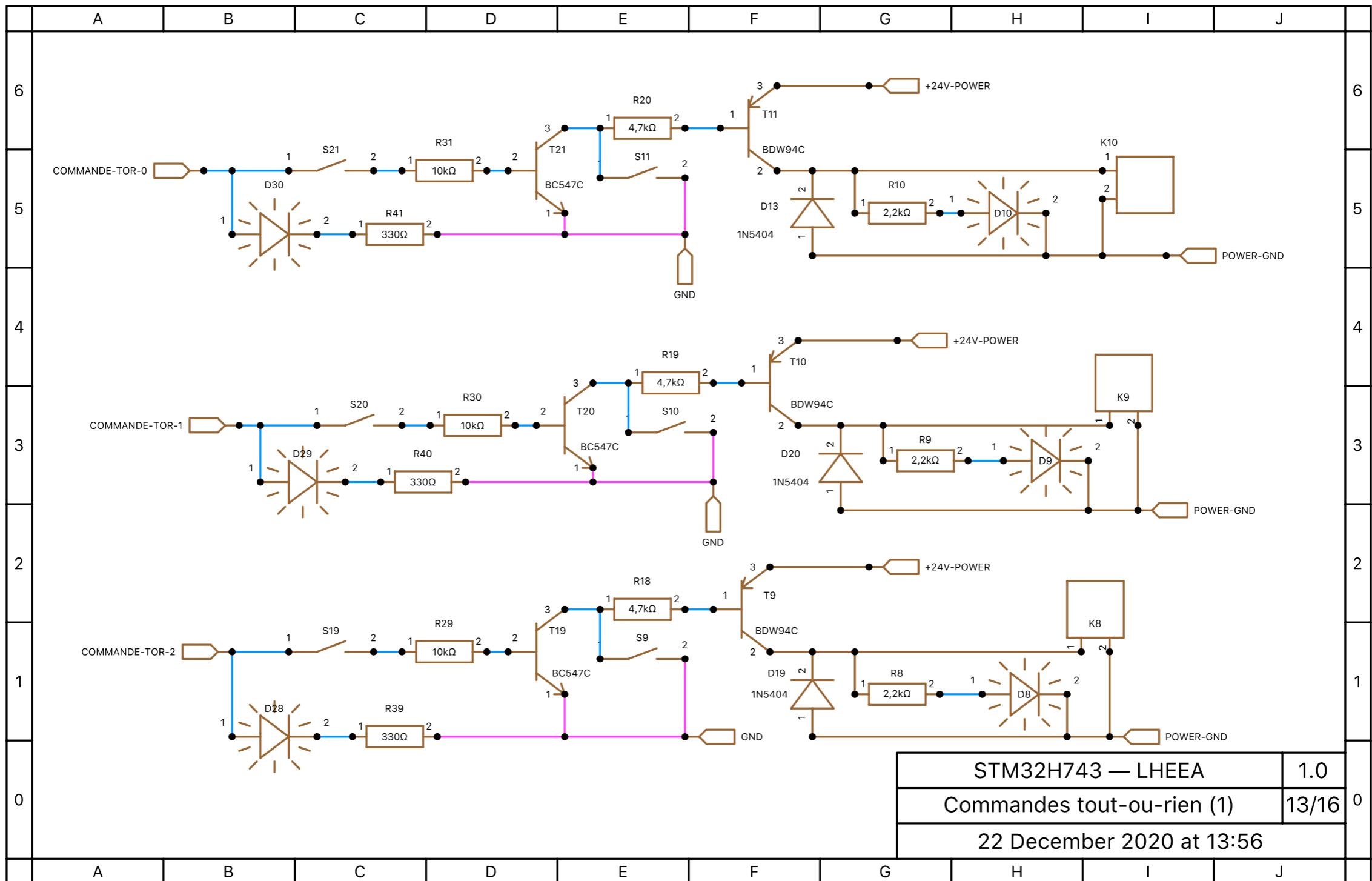
# Plan de la carte (11/16)



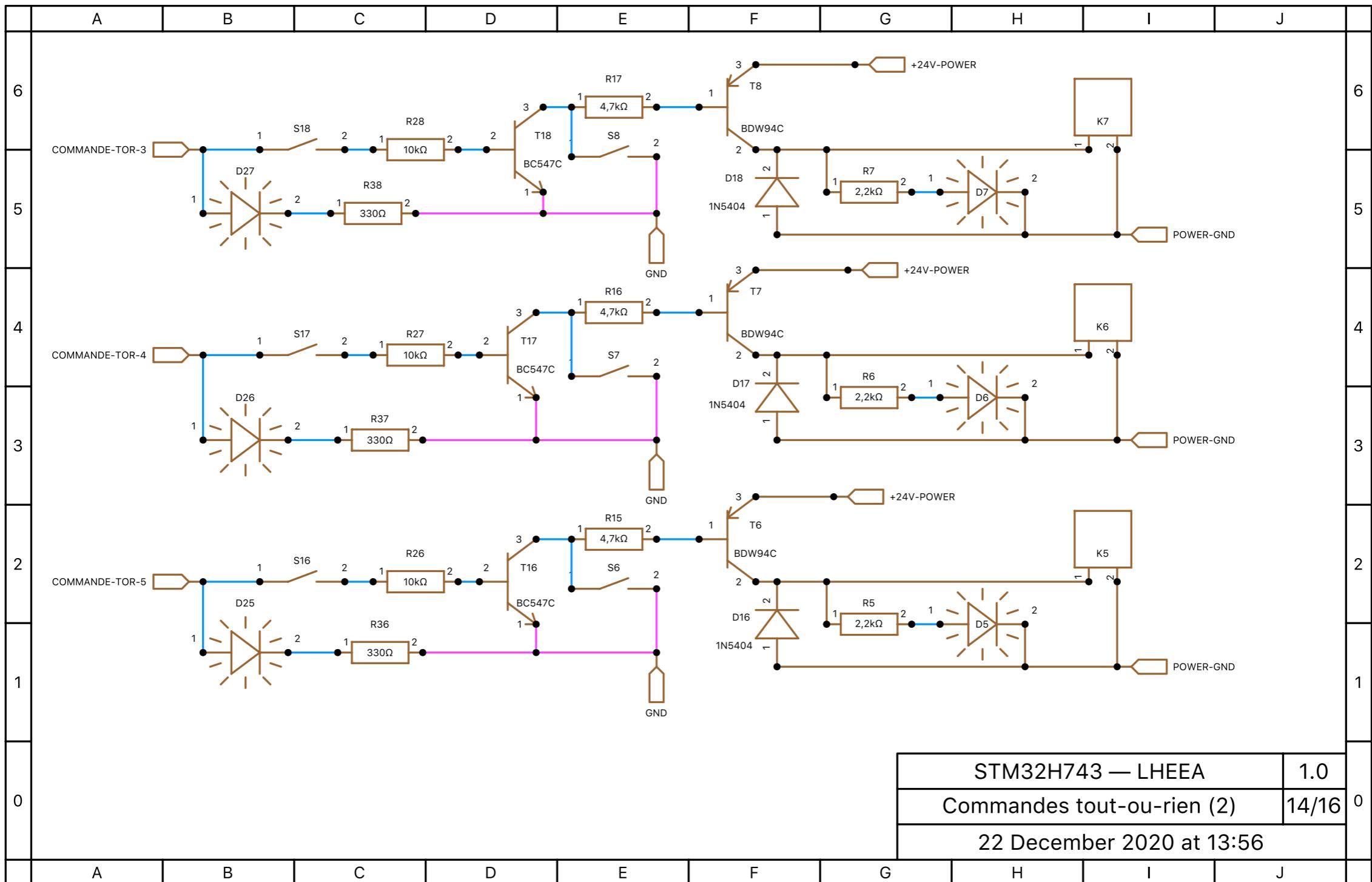
# Plan de la carte (12/16)



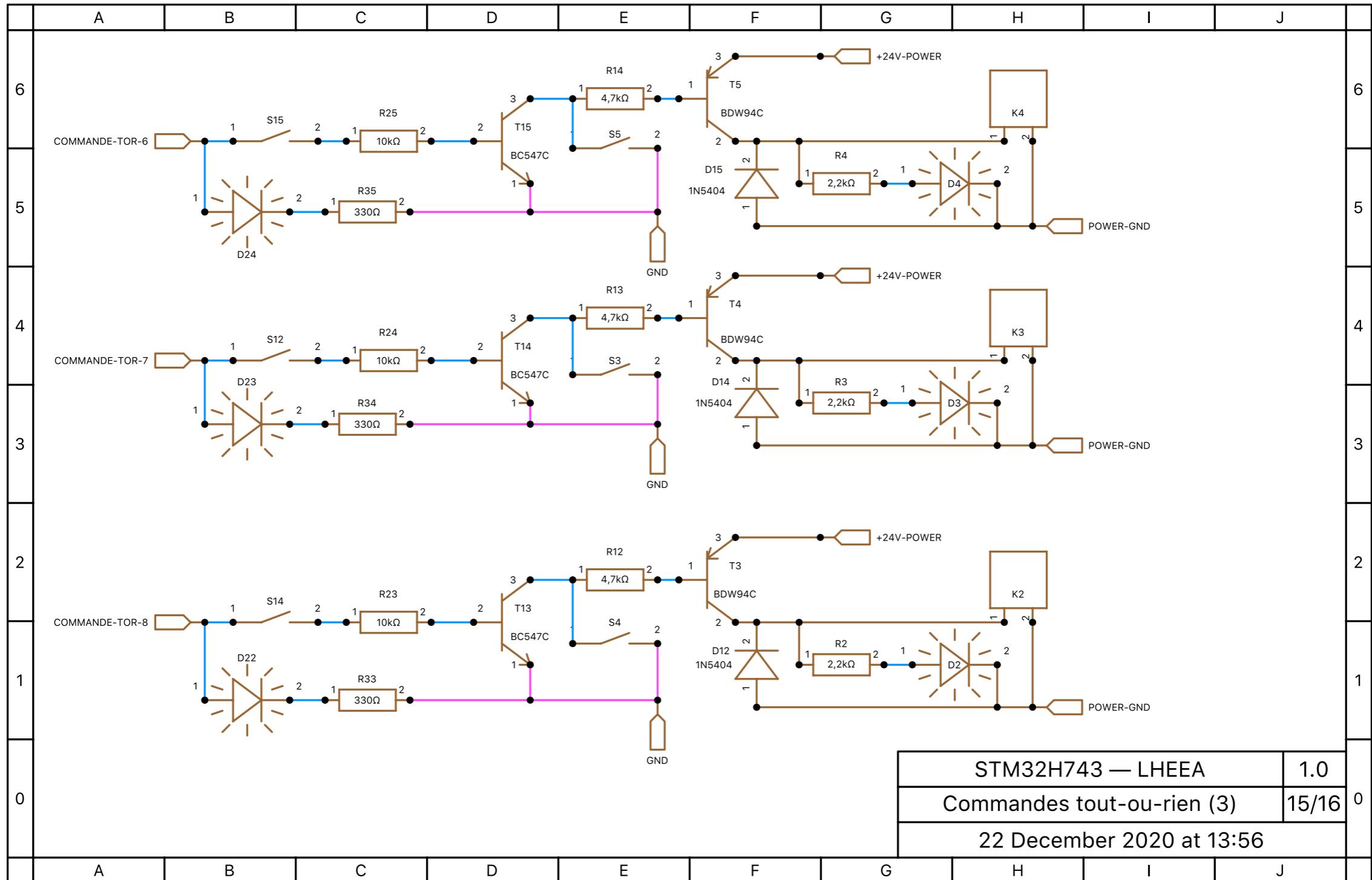
# Plan de la carte (13/16)



# Plan de la carte (14/16)



# Plan de la carte (15/16)



# Plan de la carte (16/16)

