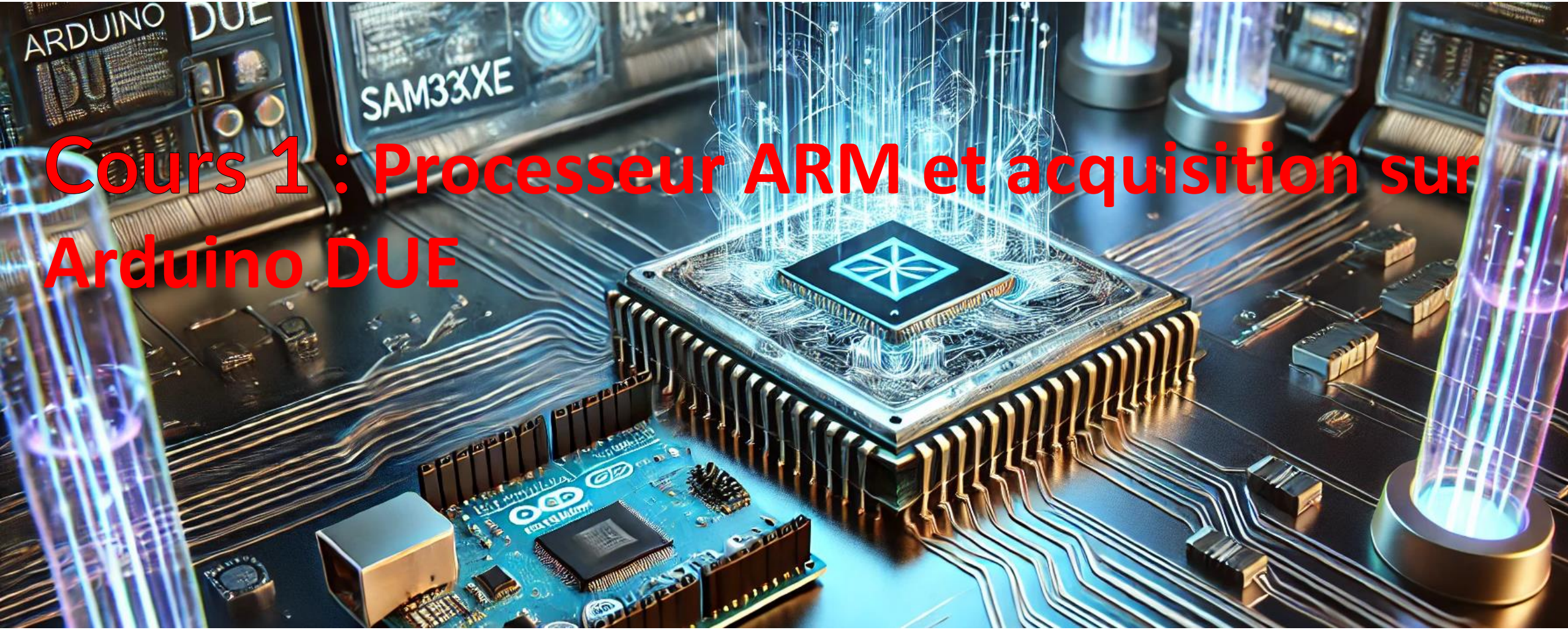
ECE
ÉCOLE D'INGÉNIEURS
ENGINEERING SCHOOL

ING3 – **Calcul embarqué**



Cours 1 : Processeur ARM et acquisition sur Arduino DUE

PÔLE ÉLECTRONIQUE

Alain Le Gall
alain.legall@ece.fr

Cours 1 : Processeur ARM et acquisition sur Arduino DUE

I. **Le processeur ARM**
   A. Introduction
   B. CISC vs RISC
   C. Avantages et inconvénients de RISC
   D. Exemple

II. **L'Arduino DUE**
   A. Présentation
   B. Acquisition de données par ADC (succinct)
   C. Utilisation du DAC

III. **Acquisition sur Arduino DUE (approfondi)**
   A. **Utilisation du ADC et DAC**
   B. **Utilisation du ADC et DAC : expérience**

I Le processeur ARM

Introduction

■ L'utilisation d'un **processeur ARM** en association avec une **Arduino Due** permet de tirer parti des performances élevées offertes par le microcontrôleur **SAM3X8E**, basé sur l'architecture **ARM Cortex-M3**. Ce type de configuration est idéal pour des applications nécessitant une acquisition de données rapide et précise, grâce à ses capacités avancées et ses nombreux périphériques.

■ Les **processeurs ARM** (*Advanced RISC Machine*) sont une famille de processeurs qui reposent sur une architecture **RISC** (*Reduced Instruction Set Computer*). Les architectures ARM représentent une approche différente de la conception du matériel d'un système par rapport aux architectures plus familières comme x86 (basée sur une architecture **CISC** : Complex Instruction Set computer).

RISC five est un autre exemple d'architecture RISC, libre celle-là.

ARM repose sur une architecture RISC, non libre.

CISC vs RISC

■ Pendant longtemps, on n'avait sur le marché que des ordinateurs avec une seule famille de processeurs : les processeurs x86 (puis x64 aussi), produits principalement produits par Intel et AMD. Avec l'avènement des appareils mobiles, c'est une autre famille qui a largement pris les devants en nombre d'appareils vendus : les processeurs ARM. Voyons les différences entre architectures ARM et x86 !

■ **CISC vs RISC** : la principale différence entre les deux architectures tient dans le choix de conception de leur jeu d'instruction : CISC (Complex Instruction Set computer) pour x86 et RISC (Reduced ISC) pour ARM. Cela signifie que les puces ARM ne supportent que des instructions simples et taille fixe (4 octets en général), s'exécutant en un nombre constant de cycles, à l'inverse des puces x86 qui proposent des instructions nécessitant plus de cycles que d'autres, pour réaliser certaines taches complexes. Les puces ARM supportent aussi moins de modes d'adressage, la plupart des instructions ne pouvant travailler qu'avec des données présentes dans un registre (petites zones de mémoire intégrées au processeur), alors que la majorité des instructions x86 peuvent aller chercher des données directement en mémoire.

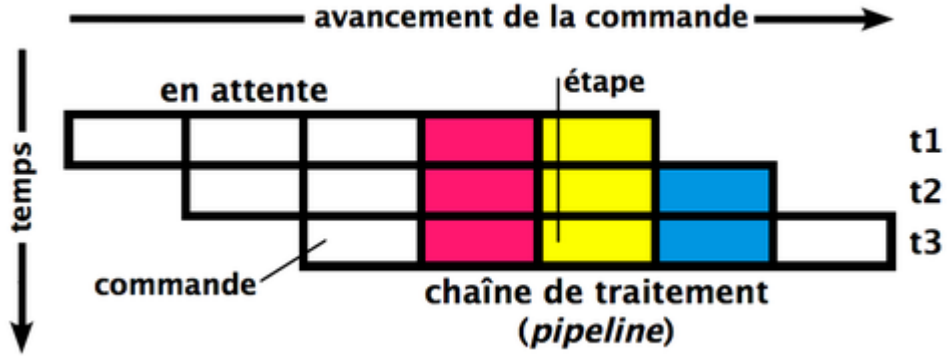
■ L'intérêt d'une architecture RISC se situe dans la moindre complexité du schéma électronique de la puce : moins de transistors (ça chauffe et consomme aussi moins ainsi) ! Ils ont plus de registres, diminuant la fréquence des accès mémoire. Il y a surtout aussi de l'optimisation au niveau du compilateur : on utilise moins d'instructions !

CISC vs RISC

■ L'architecture ARM a initialement été développée par la société Acon Computeurs (GB) qui l'utilisa sur sa gamme d'ordinateurs 32 bits Archimedes dès 1987.

■ Une particularité des processeurs ARM est leur mode de vente : ARM Ltd. ne fabrique ni ne vend ses processeurs sous forme de circuits intégrés. La société vend des licences de ses processeurs de manière qu'ils soient gravés dans le silicium par d'autres fabricants. Aujourd'hui, la plupart des grands fondeurs de puces proposent de l'architecture ARM.

■ Une notion importante : PIPELINE. Un pipeline, ou chaîne de traitement, est l'élément d'un processeur dans lequel l'exécution des instructions est découpée en plusieurs étapes. Avec un pipeline, le processeur peut commencer à exécuter une nouvelle instruction sans attendre que la précédente soit terminée. Chacune des étapes d'un pipeline est appelé étage. Le nombre d'étages d'un pipeline est appelée sa profondeur.



Avantages et inconvénients de RISC

Avantages

- L'architecture RISC possède un ensemble d'instructions, ainsi les compilateurs de langage de haut niveau peuvent produire un code plus efficace (10 instructions sont utilisées 70% du temps)
- Permet une liberté d'utilisation de l'espace sur les microprocesseurs du fait de sa simplicité. Architecture simplifiée = consommation réduite.
- De nombreux processeurs RISC utilisent les registres pour passer les arguments et contenir les variables locales (avoir plusieurs registres permet d'éviter les accès mémoires qui sont plus lents).
- Les fonctions RISC n'utilisent que quelques paramètres, et les processeurs RISC ne peuvent pas utiliser les instructions d'appel, et donc, utilisent une instruction de longueur fixe qui est facile à mettre en pipeline.
- La vitesse de l'opération peut être maximisée et le temps d'exécution peut être minimisé. Très peu de formats d'instruction, quelques nombres d'instructions et quelques modes d'adressage sont nécessaires.

Avantages et inconvénients de RISC

Inconvénients

- Les performances principales des processeurs RISC dépendent du programmeur ou du compilateur car les connaissances du compilateur jouent un rôle essentiel lors de la transformation du code CISC en code RISC, processus aussi (expansion) où le volume du code augmente.
- Le cache de premier niveau des processeurs RISC est aussi un inconvénient du RISC, ces processeurs ont de grands caches mémoire sur la puce elle-même. Pour alimenter les instructions, ils nécessitent des systèmes de mémoire très rapides. (Un cache de processeur est une mémoire matérielle utilisée par le CPU d'un ordinateur pour réduire le coût moyen (temps ou énergie) de l'accès aux données de la mémoire principale).

Exemple

■ Prenons un exemple d'un code C de calcul de la suite de Fibonacci et son équivalent en code machine pour RISC.

```
#include <stdio.h>

int fib (int i)
{
  if (i<=1) return(1);
  else return(fib(i-1)+fib(i-2));
}

int main (int argc, char *argv[])
{
  fib(2);
}
```

Exemple


```
fib:
        addi    sp,sp,-48   #set stack for fib (48 Bytes = 12 words)
        sw      ra,40(sp)   #store return adress in stack
        sw      s0,32(sp)   #store s0 in stack (callee saved)
        sw      s1,24(sp)   #store s1 in stack (callee saved)
        addi    s0,sp,48    #set frame pointer to old SP
        mv      a5,a0       #get function argument
        sw      a5,-36(s0)  #store argument (i) in stack
        lw      a5,-36(s0)  #? it seems that prev. instr. set a5 to 0
        sext.w  a4,a5       #a4 <- i
        li      a5,1        #a5 <- 1
        bgt     a4,a5,L2    # branch to L1 if i > 1
        li      a5,1        # (else branch) set R5 to 1 (result)
        j       L3          # branch L3
L2:
        lw      a5,-36(s0)  #get i (argument of fib)
        addiw   a5,a5,-1    #compute i-1
        sext.w  a5,a5       #useless ?
        mv      a0,a5       #set i-1 in argument register (s0)
        call    fib         #recursive call to fib(i-1)
        mv      a5,a0       #get result from recursive call fib(i-1)
        mv      s1,a5       #put result in s1
        lw      a5,-36(s0)  #get i (argument of fib)
        addiw   a5,a5,-2    #compute i-2
        sext.w  a5,a5       #useless ?
        mv      a0,a5       #set i-2 in argument register (s0)
        call    fib         #recursive call to fib(i-2)
        mv      a5,a0       #get result from recursive call fib(i-2)
        addw    a5,s1,a5    #a5 <- fib(i-1) + fib(i-2)
        sext.w  a5,a5       #useless?
L3:
        mv      a0,a5       #put result (1) in a0
        lw      ra,40(sp)   #restore return adresse
        lw      s0,32(sp)   #restore frame pointer
        lw      s1,24(sp)   #restore s1
        addi    sp,sp,48    #remove fib stack
        jr      ra          #return to caller code
```

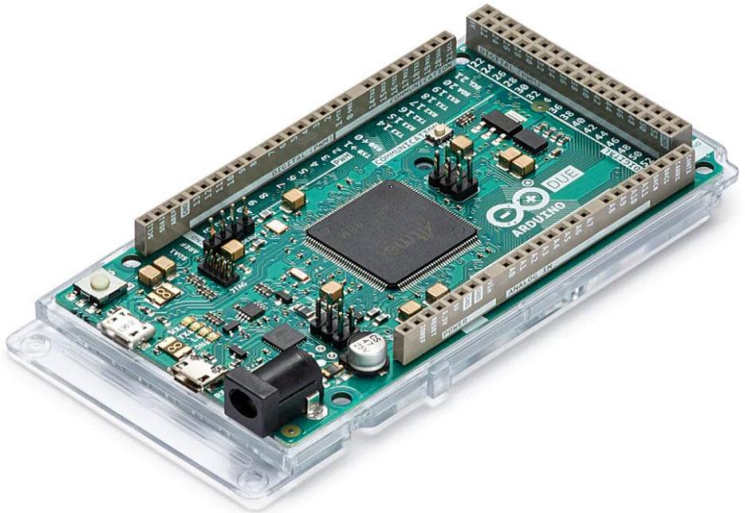
II L’Arduino DUE

Présentation

■ L'utilisation d'un processeur ARM en association avec une **Arduino Due** permet de tirer parti des performances élevées offertes par le microcontrôleur **SAM3X8E**, basé sur l'architecture ARM Cortex-M3. Ce type de configuration est idéal pour des applications nécessitant une acquisition de données rapide et précise, grâce à ses capacités avancées et ses nombreux périphériques.

■ Les caractéristiques principales du processeur ARM sur l'Arduino DUE : le SAM3X8E inclut :

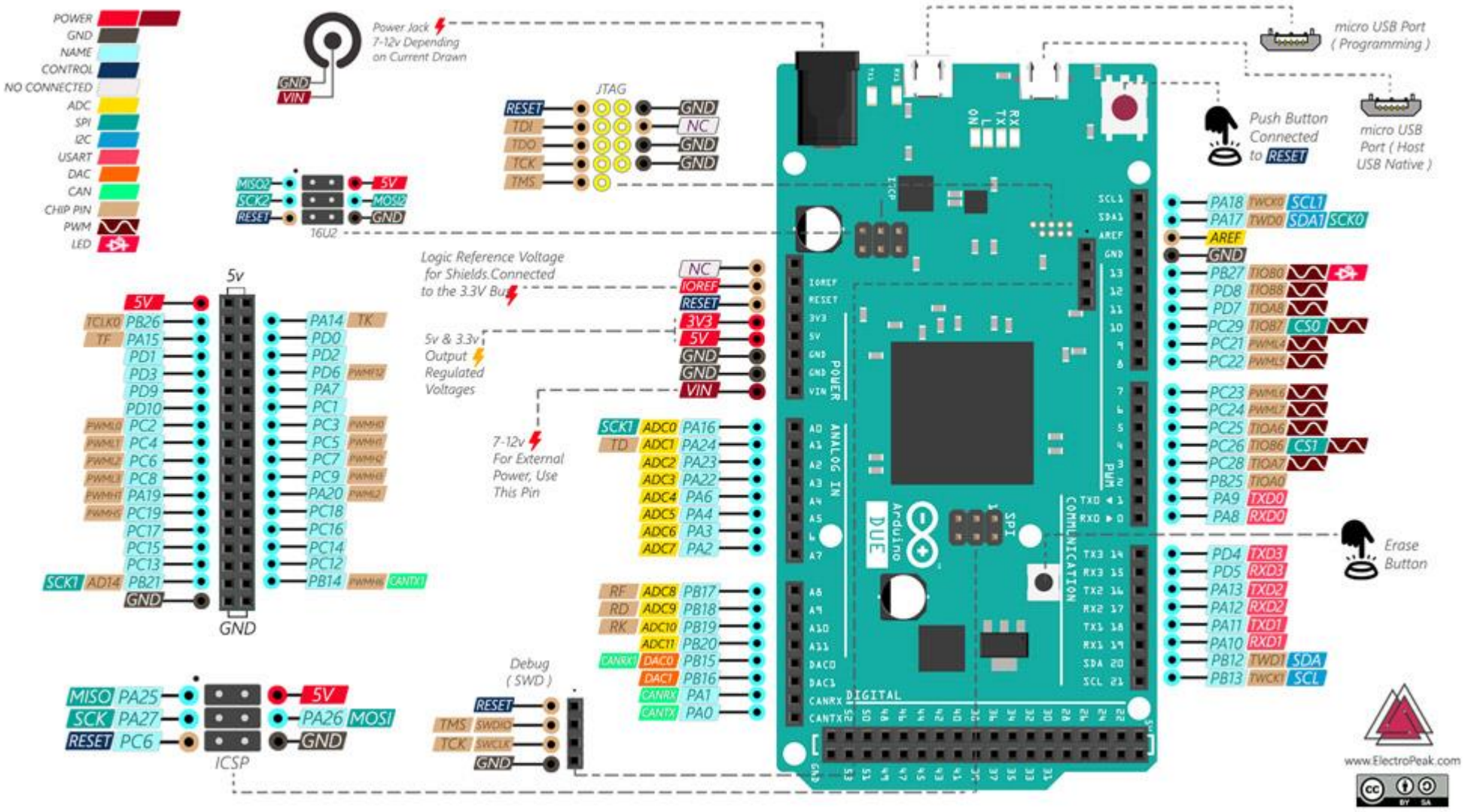
- **Horloge** : fréquence jusqu'à 84 MHz, bien plus rapide que la plupart des microcontrôleurs classiques
- **ADC** : Convertisseur Analogique Numérique :
  - Résolution 10 ou 12 bits
  - Jusqu'à 1 million d'échantillons par seconde (1 MSPS)
- **DAC** : Convertisseur Numérique Analogique :
  - 2 canaux 12 bits
- **Périphériques intégrés** :
  - Multiples UART, I2C, SPI, PWM, CAN
  - Mémoires : 96 ko de SRAM et 512 ko de Flash



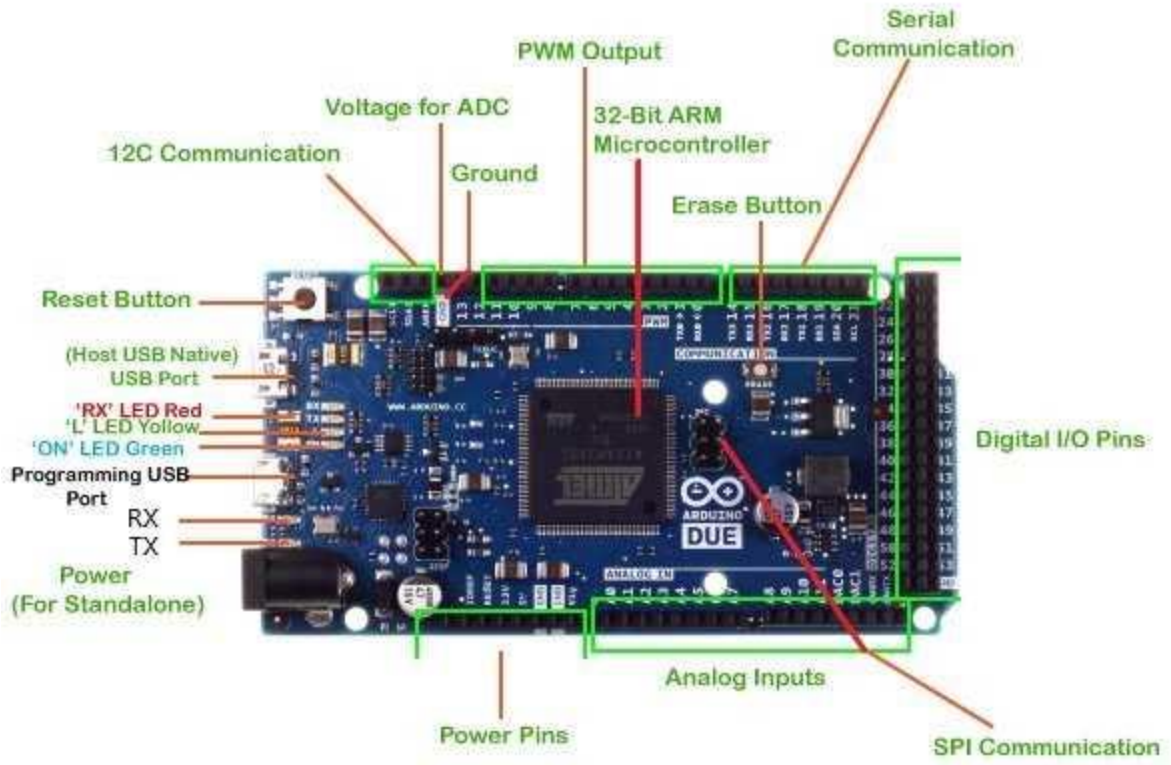
ATTENTION :les PIN ADC acceptent en entrée au maximum 3,3 V !

(il y a aussi une sortie 5V permettant d'alimenter un AOP ou autre…)

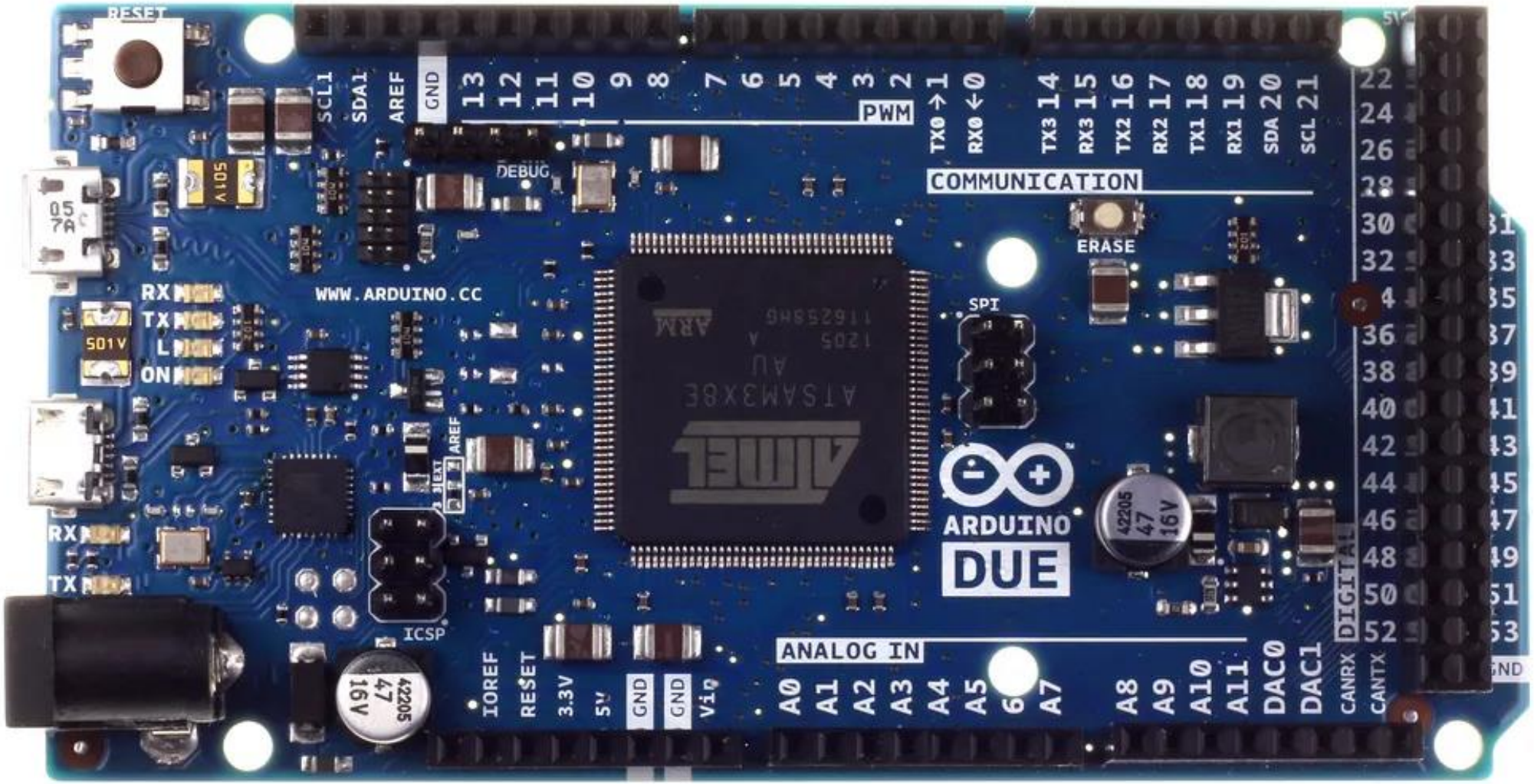
Présentation



Présentation



Présentation



Présentation

Feature	Description
Microcontroller	**Atmel SAM3X8E ARM Cortex-M3** 32-bit ARM Cortex-M3 / 84 MHz Clock speed
Memory	**SAM3X** 512 KB Flash / 96 KB SRAM (divided into two banks: 64 KB and 32 KB)
USB-to-serial	**ATmega16U2** connected to the SAM3X hardware UART
Digital Inputs	Digital Inputs not 5 V compatible (x54)
Analog Inputs	The Due's analog inputs pins measure from ground to a maximum value of 3.3 V (x12)
PWM Pins	PWM Pins with 8 bits resolution (x12)
Communication	UART (x4), I2C (x2), SPI (x1 SPI header), Native USB port (x1), Programming USB port (x1)
Power	Input voltage (VIN): 7-12 VDC / DC Current per I/O Pin: 8 mA
Dimensions	101.6 mm x 53.34 mm
Weight	36 g
Operating Temperature	-40 °C to +85 °C
Certifications	CE/RED, UKCA, FCC, IC, RCM, RoHS, REACH, WEEE

Component	Details
Atmel SAM3X8E	32-bit ARM Cortex-M3 at 84 MHz
Flash Memory	512 KB
Programming Memory	96 KB SRAM (divided into two banks: 64 KB and 32 KB)

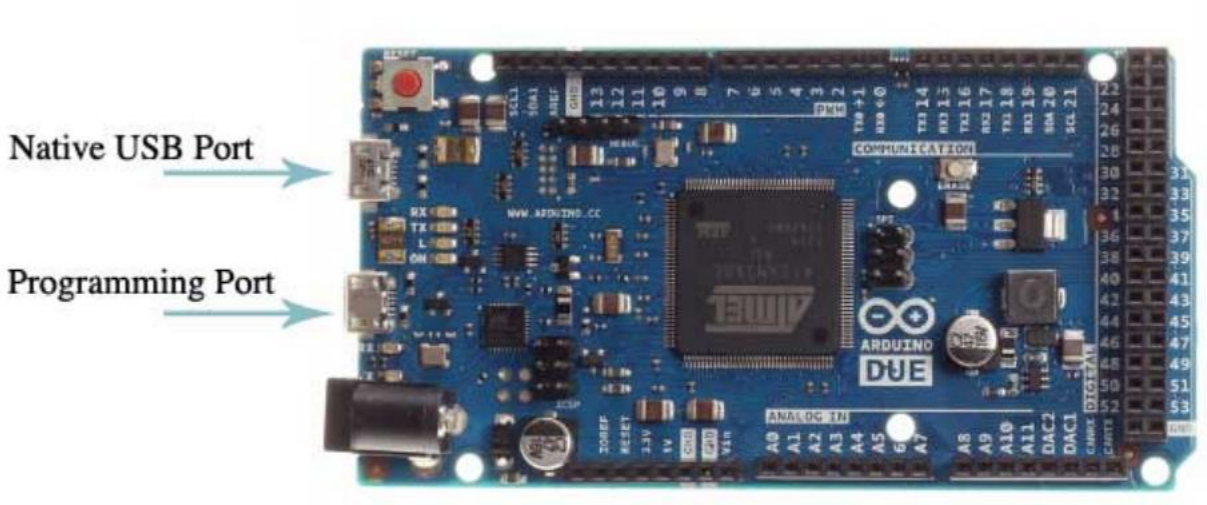
Inputs

Characteristics	Details
Number of inputs	54x digital inputs, 12x analog inputs
Inputs overvoltage protection	Yes
Antipolarity protection	Yes

Outputs

Characteristics	Details
DAC1 and DAC2	True analog output 12-bits resolution (4096 levels)
PWM outputs	12x PWM outputs

Présentation



Pour la liaison série par câble avec l'Arduino DUE, on utilisera le « **Programming Port** » (transfert de code Arduino, liaison série etc…)

Acquisition de données par ADC (succinct)

■ Pour exploiter le potentiel du processeur ARM pour des tâches comme l'acquisition rapide de signaux ou le traitement en temps réel, voici les étapes et recommandations importantes.

■ On peut bien sûr utiliser le classique « **analogRead(PINA)** » mais l'acquisition par cette méthode est en général lente.

■ On préfère utiliser les registres du microcontrôleur pour contrôler directement l'ADC, ce qui permet d'atteindre de grandes vitesses.

Exemple : on peut ajuster la fréquence d'échantillonnage et activer le mode **DMA** (Direct Memory Access) pour envoyer les données directement à la mémoire sans intervention du processeur.

Bibliothèque recommandée : **ADC Library for Arduino DUE** (elle offre des outils pour configurer facilement l'ADC et gérer des conversions rapides).

Mode DMA : le DMA est essentiel pour transférer rapidement les données issues des périphériques (comme l'ADC) vers la mémoire sans surcharger le processeur. Cela permet d'atteindre des performances maximales pour :

- Des systèmes de mesure rapides
- Des enregistrements de signaux (oscilloscope, enregistreur de données, etc…)

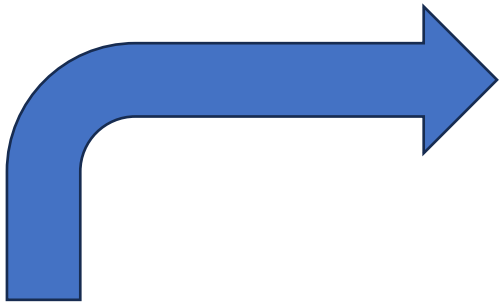
Acquisition de données par ADC (succinct)

Optimisation du code : voici quelques bonnes pratiques :

- Éviter les fonctions Arduino de haut niveau (souvent moins efficaces) pour des projets exigeants
- Préférer l'utilisation de registres et les bibliothèques bas-niveau
- Exploiter les interruptions pour déclencher des acquisitions sans bloquer le CPU

■ Application pratique :

acquisition de signaux



Voici un exemple simple d'acquisition d'un signal analogique

```
#include <ADC.h>

ADC adc; // Objet ADC

void setup() {
  Serial.begin(115200);
  adc.setResolution(12);          // Réglage de la résolution à 12 bits
  adc.setSamplingRate(ADC_FREQ_MAX); // Taux d'échantillonnage maximal
  adc.enableInterrupts(ADC_ISR_EOC); // Interruption en fin de conversion
}

void loop() {
  uint16_t value = adc.analogRead(A0); // Lecture rapide sur A0
  Serial.println(value);               // Affichage du résultat
  delayMicroseconds(100);              // Ajustez pour la fréquence souhaitée
}
```

Acquisition de données par ADC (succinct)

- Cas d'utilisation avancés :

  - **Traitement en temps réel** : le processeur ARM Cortext-M3 est assez puissant pour exécuter des algorithmes de traitement de signal (comme des filtres numériques) directement après l'acquisition.
  - **Enregistrement rapide de données** : les données acquises peuvent être stockées sur une carte SD (via SPI), ou transmises via USB ou un port série pour une analyse sur un PC par exemple.
  - **Interfaces avancées** : grâce aux périphériques CAN ou I2C rapides, vous pouvez intégrer la DUE dans des systèmes industriels ou de contrôle embarqué.

- Limitations et points de vigilance:

  - **Tension maximale d'entrée ADC** : 3,3 V (attention à ne pas dépasser cette valeur, risque de destruction)
  - **Température et consommation** : assurer une ventilation adéquate si le microcontrôleur est utilisé à pleine charge pour des périodes prolongées.

Utilisation du DAC

■ L'Arduino DUE dispose de deux sorties DAC (DAC0 et DAC1) qui permettent de convertir des signaux numériques en signaux analogiques. Ces sorties sont particulièrement utiles pour générer des formes d'ondes (sinusoïdales, triangulaires, etc…), contrôle des systèmes analogiques, ou créer des signaux de commande.

■ Caractéristiques des DAC de l'Arduino DUE :

- **Résolution** : 12 bits (valeurs de 0 à 4095)
- **Tension de sortie** : de 0 à 3,3 V (proportionnelle à la valeur numérique fournie)
- **Sorties physiques** : des broches DAC0 (pin A12) et DAC1 (pin A13) sur la carte

■ Configuration des DAC dans un programme : l'utilisation des DAC est simple avec les fonctions Arduino standard.

Exemple basique

```
void setup() {
  // Rien à configurer
}

void loop() {
  int value = 2048; // Valeur entre 0 et 4095
  analogWrite(DAC0, value); // Sortir 1,65V (moitié de 3,3V)
  delay(1000);
  analogWrite(DAC1, 4095); // Sortir 3,3V
  delay(1000);
}
```

Utilisation du DAC

■ Génération de formes d'onde avec les DAC: on peut utiliser les DAC pour générer des signaux analogiques tels que des sinusoïdes, des triangles, ou des signaux PWM lissés (pour éviter vibrations ou saccades par exemple si on commande un moteur, offrant un mouvement plus fluide).

■ **Génération d'une sinusoïde :**

```
#include <math.h> // Pour utiliser la fonction sin()

#define DAC_RESOLUTION 4095
#define PI 3.14159

void setup() {
  analogWriteResolution(12); // Réglage de la résolution à 12 bits
}

void loop() {
  for (int i = 0; i < 360; i++) {
    float angle = i * PI / 180; // Conversion en radians
    int value = (sin(angle) + 1) * (DAC_RESOLUTION / 2); // Normalisation à 0-4095
    analogWrite(DAC0, value);
    delayMicroseconds(100); // Ajuster pour la fréquence
  }
}
```

Utilisation du DAC

Génération d'un signal triangulaire :

```
#define DAC_RESOLUTION 4095

void setup() {
  analogWriteResolution(12); // Réglage à 12 bits
}

void loop() {
  // Montée du signal
  for (int value = 0; value <= DAC_RESOLUTION; value++) {
    analogWrite(DAC0, value);
    delayMicroseconds(50); // Ajuster pour la fréquence
  }

  // Descente du signal
  for (int value = DAC_RESOLUTION; value >= 0; value--) {
    analogWrite(DAC0, value);
    delayMicroseconds(50); // Ajuster pour la fréquence
  }
}
```

Utilisation du DAC

■ Optimisation avec des registres : pour des performances maximales, on peut configurer directement les registres du microcontrôleur SAM3X pour piloter les DAC avec précision, par exemple en utilisant le mode **Waveform Mode** ou en intégrant un **DMA** pour réduire la charge processeur.

■ **Activation des DAC via registres** : pour activer des fonctionnalités avancées, comme l'auto-génération de signaux, on doit manipuler de registres tels que :

- DACC_MR : configuration des modes des DAC
- DACC_CHER : activation des canaux
- DACC_CDR : registre pour écrire les valeurs numériques

Voici un exemple de configuration simplifiée des registres pour émettre une tension constante :

```
void setup() {
  pmc_enable_periph_clk(ID_DACC); // Activer l'horloge du DACC
  DACC->DACC_MR = 0x80000000;    // Mode simple
  DACC->DACC_CHER = DACC_CHER_CH0; // Activer le canal DAC0
}

void loop() {
  DACC->DACC_CDR = 2048; // Sortir une tension correspondant à 1,65 V
  delay(1000);
}
```

DACC_CHER_CH1 pour DAC1

Utilisation du DAC

■ Applications pratiques :

1. **Sortie audio** : utiliser les DAC pour générer des signaux audio (sons ou musiques) et les combiner avec un filtre passe-bas pour une qualité audio plus lisse.
2. **Commande de circuits analogiques** : commander des amplificateurs, des moteurs ou d'autres dispositifs nécessitant une entrée analogique.
3. **Test de systèmes** : simuler des signaux analogiques pour tester des circuits ou des capteurs.

■ **Limitations et précautions** :

- Les sorties DAC peuvent fournir un courant limité; si on doit alimenter une charge importante, il faut utiliser un AOP par exemple.
- Les sorties sont limitées à 3,3 V; il faut donc éviter de les connecter directement à des dispositifs nécessitant 5 V.
- Pour des signaux précis, il faut s'assurer de filtrer les bruits dans le système.

Utilisation du DAC

■ **Exemple de code de signal PWM lissé** :

On peut utiliser le code suivant pour générer un signal lissé avec un DAC sur l'Arduino DUE, sans les fluctuations rapides du PWM brut.

```
#define DAC_PIN DAC0           // Utiliser DAC0 (ou DAC1 pour changer de sortie)
#define PWM_FREQUENCY 50       // Fréquence du signal PWM désirée (en Hz)
#define SMOOTHING_STEPS 100    // Nombre d'étapes pour lisser la transition

void setup() {
  analogWriteResolution(12);  // Configurer la résolution à 12 bits pour le DAC
}
```

Utilisation du DAC


```
void loop() {
  // Génération d'un signal PWM lissé :

  // 1. Montée progressive
  for (int value = 0; value <= 4095; value += (4095 / SMOOTHING_STEPS)) {
    analogWrite(DAC_PIN, value);  // Convertir la valeur numérique en signal analogique
    delayMicroseconds(1000000 / (PWM_FREQUENCY * SMOOTHING_STEPS)); // Ajuste le temps
  }

  // 2. Descente progressive
  for (int value = 4095; value >= 0; value -= (4095 / SMOOTHING_STEPS)) {
    analogWrite(DAC_PIN, value);  // Convertir la valeur numérique en signal analogique
    delayMicroseconds(1000000 / (PWM_FREQUENCY * SMOOTHING_STEPS)); // Ajuste le temps
  }
}
```

delayMicroseconds(1000000 / (PWM_FREQUENCY * SMOOTHING_STEPS));

Cette instruction ajuste le temps entre chaque pas

III Acquisition sur Arduino DUE

Utilisation du ADC et DAC

- Si on souhaite manipuler directement les registres pour contrôler l'ADC de l'Arduino DUE, voici un exemple de code qui utilise les registres pour acquérir un signal audio. Cela permet d'optimiser les performances et de mieux comprendre le fonctionnement interne de l'ADC.

- Voici un exemple de code :

```
1  void setup() {
2    Serial.begin(115200);  // Initialisation de la communication série
3
4    // Configuration de l'ADC
5    PMC->PMC_PCER1 |= PMC_PCER1_PID37;  // Active l'horloge de l'ADC
6    ADC->ADC_MR = ADC_MR_PRESCAL(1) | ADC_MR_STARTUP_SUT0 | ADC_MR_TRACKTIM(15) | ADC_MR_SETTLING_AST3;  // Configuration du mode ADC
7    ADC->ADC_CHER = ADC_CHER_CH0;  // Active le canal 0 (A0)
8  }
9
```

Utilisation du ADC et DAC


```
10  void loop() {
11    static unsigned long lastSampleTime = 0;
12    unsigned long currentTime = micros();
13
14    // Vérifiez si le temps écoulé correspond au taux d'échantillonnage
15    if (currentTime - lastSampleTime >= 1000000UL / 44100) {
16      lastSampleTime = currentTime;
17
18      // Démarre la conversion ADC
19      ADC->ADC_CR = ADC_CR_START;
20
21      // Attend la fin de la conversion
22      while (!(ADC->ADC_ISR & ADC_ISR_DRDY));
23
24      // Lit la valeur ADC
25      uint16_t audioValue = ADC->ADC_CDR[0];
26
27      // Envoi de la valeur via le port série
28      Serial.println(audioValue);
29    }
30  }
```

Utilisation du ADC et DAC

- Expliquons le code :

INITIALISATION :

On active l'horloge de l'ADC en activant le périphérique correspondant dans le Power Management Controller (PMC) : PMC->PMC_PCER1 |= PMC_PCER1_PID37;

On configure le mode de l'ADC. Ici, on utilise un prescaler de 1, un temps de démarrage court, un temps de suivi de 15 cycles d'horloge, et un temps de stabilisation de 3 cycles d'horloge :
ADC->ADC_MR = ADC_MR_PRESCAL(1) | ADC_MR_STARTUP_SUT0 | ADC_MR_TRACKTIM(15) | ADC_MR_SETTLING_AST3;

On active le canal 0 (A0) pour la conversion ADC : ADC->ADC_CHER = ADC_CHER_CH0;

Utilisation du ADC et DAC

- Expliquons le code :

BOUCLE PRINCIPALE :

La condition vérifie si le temps écoulé depuis le dernier échantillon correspond au taux d'échantillonnage souhaité (44.1 kHz) : if (currentTime - lastSampleTime >= 1000000UL / 44100)

On démarre la conversion ADC : ADC->ADC_CR = ADC_CR_START;

On attend que la conversion soit terminée en vérifiant le bit DRDY (Data Ready) dans le registre d'état de l'ADC (ADC_ISR) : while (!(ADC->ADC_ISR & ADC_ISR_DRDY));

On lit la valeur convertie depuis le registre de données de l'ADC (ADC_CDR) pour le canal 0 : uint16_t audioValue = ADC->ADC_CDR[0];

Puis, on envoie la valeur lue via le port série.

Utilisation du ADC et DAC

- **Remarques :**

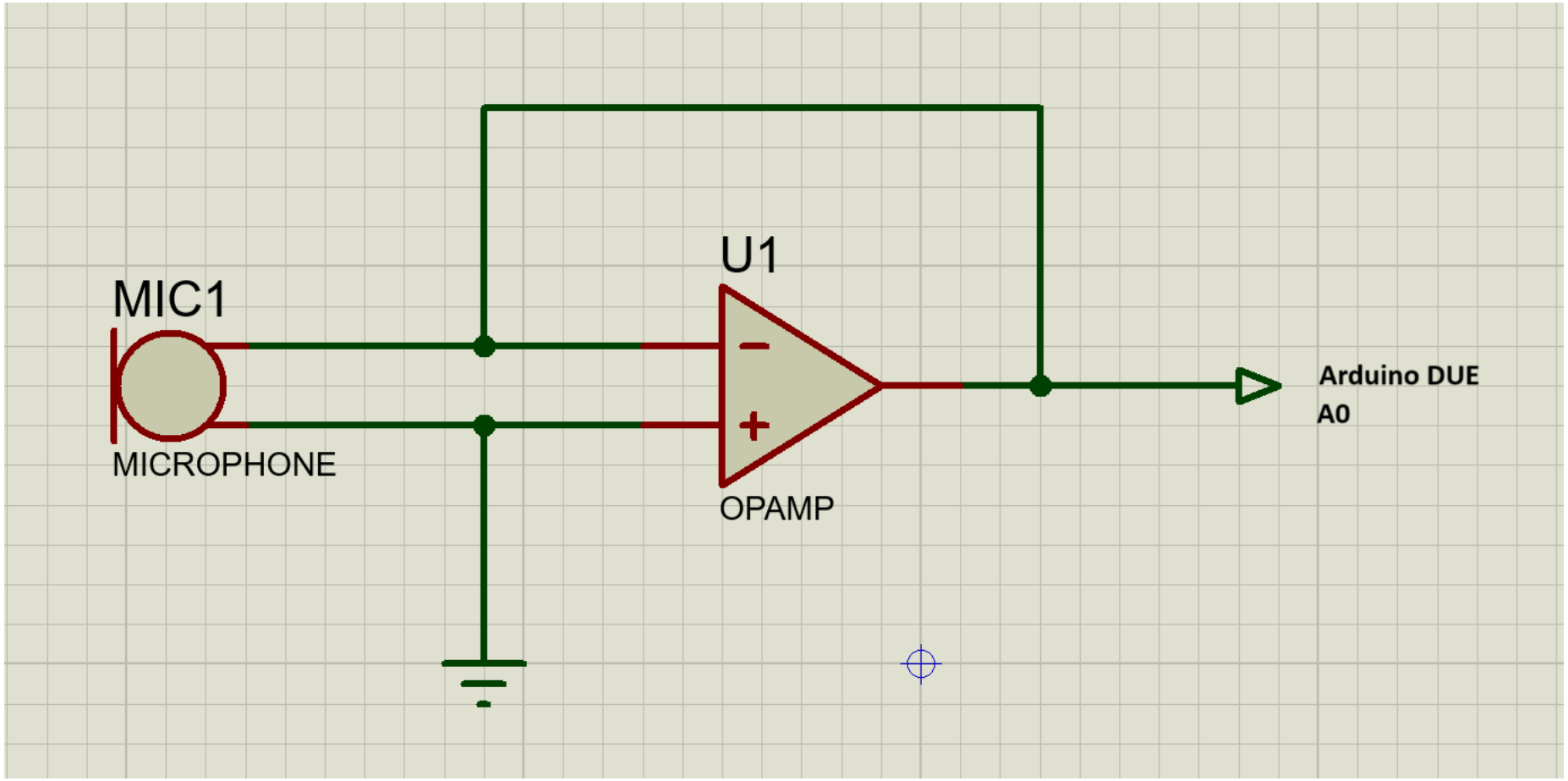
Optimisation : Ce code est optimisé pour une acquisition rapide en manipulant directement les registres, ce qui permet d'éviter les surcharges liées aux fonctions de haut niveau.

Taux d'échantillonnage : Le taux d'échantillonnage est fixé à 44.1 kHz, mais vous pouvez l'ajuster en fonction de vos besoins.

Visualisation des données : Comme précédemment, vous pouvez utiliser un logiciel de visualisation pour afficher et analyser les données audio reçues via le port série.
Ce code est un exemple avancé qui montre comment manipuler directement les registres pour contrôler l'ADC de l'Arduino DUE. Cela peut être utile pour des applications nécessitant une performance maximale ou une compréhension approfondie du matériel.

Utilisation du ADC et DAC : expérience

■ **Utilisons le ADC puis le DAC pour restituer un signal**: on utilise un microphone (celui du kit) pour saisir l'entrée d'un signal sonore, selon le schéma suivant :



Utilisation du ADC et DAC : expérience

- L'amplificateur opérationnel est branché de façon à saturer à +Vsta=3,3 V et à –Vsat=0 V

Cela permet de ne pas endommager la PIN A0 de l'Arduino DUE qui ne tolère en entrée qu'une tension entre 0 et 3,3 V.

- Le code employé est le suivant :

```
1   const int micPin = A0;  // Broche analogique où le microphone est connecté
2   const int sampleRate = 44100;  // Taux d'échantillonnage en Hz (44.1 kHz pour l'audio)
3
4   void setup() {
5     Serial.begin(115200);  // Initialisation de la communication série
6     analogReadResolution(12);  // Configuration de l'ADC en 12 bits
7     pmc_enable_periph_clk(ID_DACC); // Activer l'horloge pour le DAC
8     DACC->DACC_MR = 0x80000000;    // Mode de fonctionnement simple (par défaut 12 bits)
9     DACC->DACC_CHER = DACC_CHER_CH0; // Activer DAC0
10  }
11
```

- sampleRate nous indique la fréquence d'échantillonnage Fe . Il faut d'après le théorème de Shannon que Fe>2Fmax où Fmax est la fréquence maximale du signal d'entrée (ici 20 kHz car signal sonore).

Utilisation du ADC et DAC : expérience

- La suite du code :

```
12  void loop() {
13    static unsigned long lastSampleTime = 0;
14    unsigned long currentTime = micros();
15
16    // Vérifiez si le temps écoulé correspond au taux d'échantillonnage
17    if (currentTime - lastSampleTime >= 1000000UL / sampleRate) {
18      lastSampleTime = currentTime;
19
20      // Lecture de la valeur analogique
21      int audioValue = analogRead(micPin);
22
23      // Envoi de la valeur via le port série
24      Serial.println(audioValue);
25      DACC->DACC_CDR = audioValue; // Écrire dans le DAC
26    }
27  }
```

Utilisation du ADC et DAC : expérience

■ On a contrôlé la saisie des échantillons sonores. Cela permet de respecter la fréquence d'échantillonnage. Aussi, si on branche un Haut Parleur en sortie du DAC0, on pourra entendre à l'oreille le son entré au micro ! On peut ajouter un amplificateur non inverseur pour amplifier le son avant de l'envoyer vers le HP aussi.

■ On a aussi envoyé les échantillons du son numérisé dans la console série afin de pouvoir les récupérer et analyser à l'aide d'un logiciel libre comme COOLTERM : Roger Meier's Freeware

https://freeware.the-meiers.org

■ Nous ferons ces expériences lors du TP1.

■ On remarque ici qu'on a utilisé l'instruction analogRead ! Et l'expérience fonctionne bien. Les performances auraient été encore meilleures si on avait utilisé aussi les registres pour l'ADC… !

IV Conclusion

Conclusion

- L'Arduino DUE est équipé d'une technologie ARM, rapide et efficace. Couplée à ses entrées ADC et sorties DAC aussi, elle permet, lorsqu'on manipule ses registres, d'obtenir des saisies rapides et précises, ainsi qu'une possibilité de sortie analogique aussi.

Tout cela permettra de mener à bien des opérations de traitement du signal grâce à l'Arduino DUE, sur des signaux sonores (même des signaux plus rapides peuvent être aussi traités).