ESE30 - Module électif mécatronique

F. Pessaux, T. Bernard et S. Finet

Énoncé

Tout au long de ces exercices, vous devrez écrire des programmes. Dans la majorité des cas, ce sont de petites modifications des programmes précédents (des fois 2 ou 3 lignes).

Donc, pensez à dupliquer le programme précédent qui convient et modifiez cette copie pour obtenir le programme souhaité. Vous gagnerez ainsi beaucoup de temps.

1 Prise en main de l'environnement de développement

L'environnement de développement pour Arduino est normalement installé dans le répertoire :

`/auto/appy/arduino/arduino-1.6.0/`

des machines d'enseignement. Il est sinon disponible librement en téléchargement sur le site officiel :

`http://arduino.cc/en/Main/Software`

Il contient une version du compilateur gcc dédiée à l'architecture de la carte ainsi que des en-têtes (*includes*, *headers*), des bibliothèques bas-niveau et des scripts d'édition de liens. Afin de ne pas rajouter à la difficulté d'adaptation, vous ne compilerez pas en utilisant un `Makefile`. Vous utiliserez l'environnement de compilation Arduino qui met à votre disposition un éditeur de source sommaire, mais suffisant pour les besoins du projet (vous pouvez toujours utiliser un autre éditeur de texte en allant dans les Préférences). Cet environnement se lance en invoquant la commande :

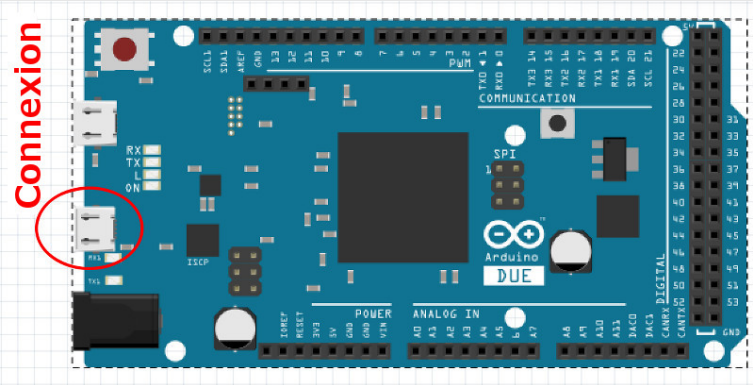
`/auto/appy/arduino/arduino-1.6.0/arduino`

Un éditeur de texte apparaît avec quelques boutons à gauche dont 2 sont capitaux : ≪Vérifier≫ (compiler) et ≪Téléverser≫ (charger sur la carte). Deux options sont à configurer :

— `Outils > Type de carte > Arduino Due (Programming Port)`.

— `Outils > Port > /dev/ACM0`.

La connexion est faite entre un port USB de l'ordinateur et le port le plus proche de la prise d'alimentation externe de la carte (c.f. image ci-dessous). L'alimentation de la carte par le port USB est suffisante dans notre cas.



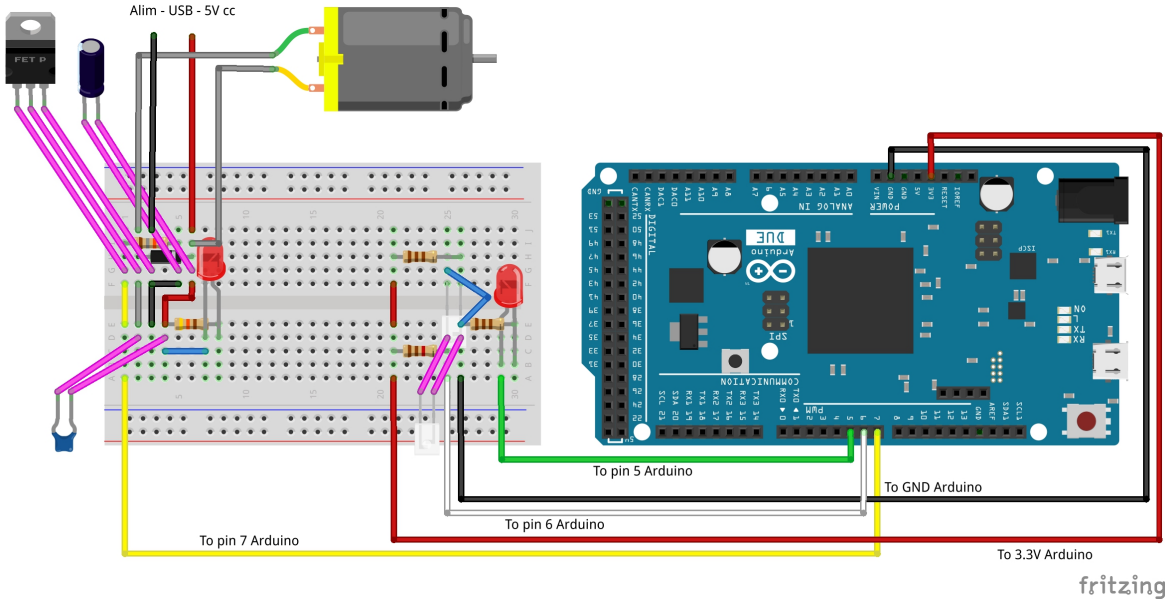
Q1 : Connectez la carte à un port USB de votre ordinateur. Lancez l'environnement de compilation Arduino et réglez les paramètres ≪type de carte≫ et ≪port≫.

L'environnement de compilation Arduino contient déjà quelques exemples directement téléchargeables sur la carte. Ces exemples se trouvent dans le menu `Fichier` > `Exemples`.

Q2 : Afin de vérifier le bon fonctionnement de votre installation, sélectionnez l'exemple se trouvant dans `Fichier` > `Exemples` > `Basics` > `Blink`. Cet exemple se contente de faire clignoter toutes les secondes la LED témoin se trouvant directement sur la carte (également reliée à la broche 13 de la carte). Compilez ce programme, téléchargez-le et vérifiez que tout fonctionne.

2 Présentation générale du montage

Le montage électronique qui vous est proposé est constitué de la carte Arduino elle-même et d'un certain nombre de composants électroniques *discrets* agencés sur une *breadboard*. Ce montage permet de piloter un moteur à courant continu et de détecter sa vitesse de rotation depuis la carte Arduino.



2.1 Le micro-contrôleur : `SAM3x8E`

Au cœur de la carte Arduino mise à votre disposition se trouve un micro-contrôleur (≪MCU ≫) `SAM3x8E` (`SAM3X ARM Cortex-M3`) à 84 MHz de chez Atmel Corporation. Ce micro-contrôleur intègre une architecture ARM 32 bits et de nombreuses interfaces d'entrée/sortie (numériques, analogiques, Ethernet, CAN, USB…). Les processeurs RISC d'architecture ARM sont particulièrement répandus le domaine de l'informatique embarquée (téléphones, tablettes, consoles de jeux…).

Q1 : À votre avis, quel va être le rôle du MCU ?

2.2 Les composants discrets

Le dispositif électronique se compose de 2 parties distinctes en terme de fonctionnalité …et d'alimentation électrique.

— La partie alimentation du moteur.

— La partie contrôle du moteur : contrôle en vitesse et acquisition de la vitesse.

Q2 : À votre avis, quelle est la raison de cette séparation ?

3 Détails de l'électronique sous-jacente

Un moteur électrique met en jeu des niveaux de tension et de courant bien plus élevés que ceux rencontrés dans le monde des microprocesseurs. Pour le piloter, il faut des composants, des montages et des techniques dédiés : on parle d'électronique de puissance. Ici, on va mettre en œuvre un moteur très modeste, fonctionnant dans une plage de tension typique de 1 à 5V et avec un courant inférieur à 1A.

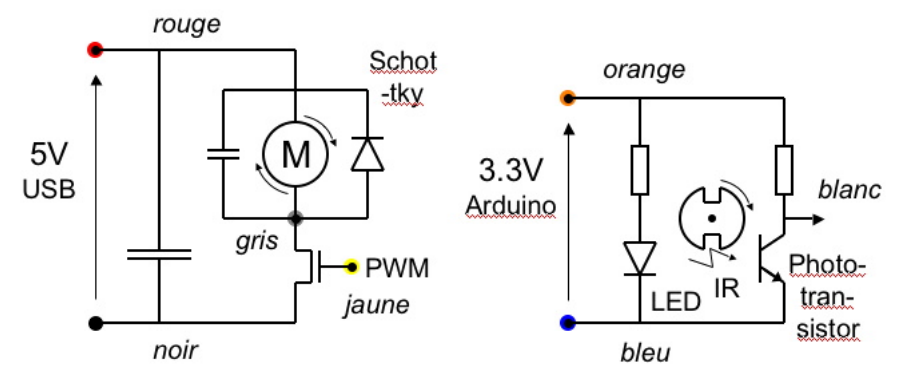
Une source de tension commandable dynamiquement semble être la solution la plus naturelle pour piloter un moteur à courant continu. Mais une telle fonction analogique est difficile à réaliser simplement avec un bon rendement énergétique. Une alternative consiste à utiliser une source de tension constante et à l'imposer en tout ou rien aux bornes du moteur. Il faut alors alterner le mode *tout* (i.e. connecté) et le mode *rien* (i.e. déconnecté) en proportions adéquates et suffisamment vite par rapport aux constantes de temps mécaniques dudit moteur. En pratique, on adopte une alternance périodique, la période étant appelée *temps de cycle* et fixée typiquement à 1ms. Cette façon de procéder est appelée « modulation par largeur d'impulsion ». Chaque impulsion correspond au mode *tout*. Cela se traduit en anglais par PWM, pour *Pulse Width Modulation*. La proportion du temps passé en mode *tout* fixe la tension moyenne vue par le moteur. Celle-ci est gérable/pilotable facilement grâce aux technologies numériques dont les fréquences de fonctionnement sont bien plus grandes que les fréquences propres ou caractéristiques du monde mécanique. Nous reviendrons sur ce point et l'exploiterons en section 8.

Ce procédé exploite la source de tension tantôt pleinement, tantôt pas du tout, ce qui est très intéressant en terme de rendement énergétique. En revanche, il faut pouvoir commuter rapidement entre les deux modes. Des composants électroniques dédiés sont utilisés pour cela : des IGBT pour les moteurs de forte puissance, des transistors MOS dits « de puissance » (à faible résistance de fonctionnement notamment) lorsque la puissance est plus modérée, comme c'est le cas ici.

Notre parti pris pour ce travail expérimental est d'exposer la réalité électronicienne. Ce sont donc des composants discrets qui sont utilisés et installés sur une platine d'expérimentation, appelée *breadboard* en anglais. Celle-ci présente une double série de rangées à 5 trous connectés électriquement entre eux. Les autres connexions y sont réalisées par des fils de couleur. Ces couleurs respectent certaines conventions. Ici la masse analogique (0V) est en noir et l'alimentation analogique (5V) en rouge. Elles servent pour le moteur et son électronique de puissance. Il y a aussi une masse numérique en bleu et une alimentation numérique (3,3V) en orange, communes avec la carte Arduino. Cela prend vite de la place d'installer quelques composants et leur connexions sur une telle breadboard. En conséquence, on s'est limité aux montages électroniques les plus simples. C'est ainsi qu'un unique transistor MOS de puissance est utilisé : c'est le composant le plus encombrant que l'on voit sur cette breadboard. C'est un trident adossé à une plaque métallique permettant de diffuser l'énergie thermique dissipée par sa résistance équivalente. Ce choix d'un unique MOS fait que le moteur ne pourra tourner que dans un sens (situation dite « monoquadrant »). Il aurait fallu un montage dit « pont en H » pour obtenir les deux sens de rotation (situation dite « 4 quadrants ». Pour cela, on aurait été contraint d'utiliser un composant intégré, cachant le détail du montage, mais qui aurait été certes plus proche d'une solution industrielle.

Ce transistor MOS est de type **n** pour présenter la plus faible résistance possible (les électrons sont plus mobiles que les trous). Il est donc connecté, en série avec le moteur, entre masse

(noir) et alimentation (rouge) analogiques. Par commodité, ces derniers sont issus du port USB d'un ordinateur, port qui comporte une source de tension de 5V pouvant fournir jusqu'à 0,5A selon la norme, parfois plus. Pour rendre passant le transistor MOS, dont la source est connectée à la masse, sa grille (fil jaune) doit recevoir au moins 2,5V. Celle-ci recevra en fait 3,3V, correspondant à un 1 logique provenant de la carte à microcontrôleur Arduino dont les entrées/sorties fonctionnent sous 3,3V. Le drain du MOS est connecté à une borne du moteur, via un fil gris. L'autre borne du moteur est connectée à l'alimentation (rouge).



Lorsque le transistor MOS est passant, c'est-à-dire en mode *tout*, le moteur voit une tension de 5V à ses bornes (moins la petite chute ohmique sur le MOS). Durant cette phase (qui correspond à *une* impulsion d'une certaine longueur en temps), le courant qui circule dans les bobines du moteur augmente, suivant la loi de Lenz. Le transistor MOS est ensuite rendu non passant, par application d'une tension nulle sur sa grille : c'est ainsi que l'on commute du *tout* au *rien*. Ceci coupe le chemin électrique entre alimentation et masse sur lequel se trouve le moteur. Or l'intensité circulant dans le moteur ne peut chuter instantanément - en raison de la même loi de Lenz - sauf à laisser apparaître une tension inverse extrêmement élevée – donc destructrice – à ses bornes. Idéalement, pour traiter cette difficulté, il faudrait pouvoir reboucler le moteur sur lui-même. Un interrupteur reliant les deux bornes du moteur pourrait remplir cette fonction, à condition d'être commandé en *parfaite* opposition de phase avec le MOS.

Implanter un tel interrupteur de rebouclage et le piloter correctement est délicat. Une alternative consiste à installer une diode aux bornes du moteur, usuellement appelée ≪diode de roue libre ≫. Celle-ci doit permettre au courant installé dans les bobines du moteur de circuler, à la faveur d'un retournement de la tension à ses bornes au début du mode *rien*. Cette diode est donc installée en sens *inverse* pour permettre au courant de circuler du fil gris ou fil rouge. Elle est donc bloquée pendant le mode *tout*, ce qui est bien sûr indispensable. En mode *rien*, elle va dissiper une certaine énergie, car sa tension de seuil est non nulle, donc freiner un peu le moteur. Afin d'optimiser les performances, c'est une diode de type Schottky que l'on utilise : elle présente une tension de seuil plus faible qu'une diode **pn** classique, ce qui minimise la dissipation thermique d'énergie et améliore la réactivité. Sur la platine, cette diode – petit cylindre noir horizontal – est placée juste au dessous et à l'arrière du transistor MOS, docn un peu difficile à voir.

Avec les dispositions prises ci-dessus, la tension s'inverse brutalement aux bornes du moteur lors de la commutation de *tout* à *rien*, mais la diode ouvre les ≪vannes ≫ du rebouclage dès que cette inversion dépasse quelques dixièmes de volts. Le courant, quant à lui, diminue en mode *rien*, en raison de la dissipation mécanique et thermique de l'énergie. Avec la hausse qui

intervient en mode *tout*, il s'instaure un régime périodique (sous des conditions d'utilisation constantes en régime et en charge).

La forte agitation en tension, due au principe même de la modulation PWM, perturbe l'environnement électromagnétique, par exemple la masse analogique, dont le potentiel peut fluctuer sensiblement autour de 0V. Pour réduire ces perturbations, on utilise des condensateurs. Sur la platine, le plus volumineux est un condensateur cylindrique vertical, installé juste à la droite du MOS et placé entre alimentation (rouge) et masse (noire) analogiques, issues du port USB. Sa capacité, 100μF, est assez importante, grâce à sa technologie de fabrication dite ≪électrolytique ≫ (qui lui impose un sens de montage). En revanche, ce condensateur présente une mauvaise réactivité et ne peut modérer les transitoires les plus rapides. Pour lutter contre ceux-ci, il faut utiliser des technologies de condensateur plus rapides, mais qui n'atteignent pas des valeurs de capacités aussi élevées. Heureusement, une valeur de capacité bien plus faible est suffisante. Au point qu'un tel condensateur rapide peut être installé directement aux bornes du moteur, c'est-à-dire à la racine du problème, sans altérer significativement le fonctionnement du triplet MOS-diode-moteur. Sur la platine, c'est un condensateur de capacité 10nF, de forme rectangulaire et couleur grise, qui est placé, à l'avant, entre fils gris et rouge. Technologiquement, il est à base de film isolant en polyester.

Finalement ce sont 4 composants essentiels (un transistor MOS, une diode et deux condensateurs) qui constituent l'électronique de puissance utilisée dans le cadre de ce microprojet. Leurs rôles respectifs sont expliqués suivant une démarche constructive dans les paragraphes précédents. Il reste quelques détails à mentionner. Bien que connectée à une sortie de la carte Arduino, la grille du MOS (fil jaune) sera parfois non pilotée, donc à une tension indéterminée, pouvant faire fonctionner le moteur de façon intempestive. Pour éviter cela, on la connecte à la masse via une résistance de forte valeur (dizaines de kilo-ohms) : on parle de résistance de *pull-down*. Sur la platine, celle-ci est placée à l'arrière de la diode. Ne pouvant fournir qu'un courant inférieur au milliampère, elle s'effacera devant toute tension imposée depuis l'Arduino, capable de délivrer environ 20mA. Le montage comporte aussi une LED (diode électro-luminescente) jaune avertissant de la mise sous tension depuis le port USB. Une résistance de valeur adéquate est installée en série avec cette LED afin d'y fixer le courant à une valeur de quelques milliampères, ce qui fournit un éclairage adéquat tout en évitant de la ≪griller ≫.

Le moteur et son électronique de puissance constituent la partie *actuateur* du système. Mais il faut aussi observer ce que fait le moteur pour pouvoir le piloter judicieusement, d'où la nécessité d'un capteur. Nombreuses sont les applications où il est nécessaire de connaître la position angulaire du rotor avec des précisions de l'ordre du degré, ou encore supérieures. Dans le cadre du présent projet, on se contente d'une précision au demi-tour près.

Sur l'arbre du moteur est installée une roue coupée de deux fentes diamétralement opposées servant à laisser passer la lumière. En positionnant une LED d'un côté et un photorécepteur de l'autre, le passage d'une fente peut-être détecté. C'est généralement une lumière infrarouge (IR) qui est utilisée et c'est l'option adoptée ici. Certains capteurs d'image de smartphone permettent de l'apercevoir, moyennant une obscurité suffisante faite autour du montage. Dans le photorécepteur, le photocourant issu de la séparation des paires électrons-trous sous l'effet des photons reçus est extrêmement faible. Il se compte en nano-ampères. Inexploitable directement, vu l'ordre de grandeur des capacités environnantes, ce courant doit être amplifié. Ceci peut-être fait *in situ* par exploitation de *l'effet transistor* (bipolaire), qui permet en gros de gagner un facteur proche de 1000. Un tel composant, appelé ≪phototransistor ≫, est utilisé sur le montage. On le voit, translucide, à l'avant de la roue. La LED IR, située à l'arrière, est, elle, de couleur

légèrement rosée. Elle est mise en série avec une résistance adéquate pour y établir le courant à 20mA, courant nominal maximum, afin d'obtenir le flux lumineux IR le plus élevé possible. Le phototransistor est quant à lui mis en série avec une résistance de l'ordre du kilo-ohm, valeur assez élevée pour obtenir une chute ohmique de plusieurs volts en pleine lumière, donc détectable numériquement, mais pas trop élevée afin d'avoir une réactivité suffisante face à la brièveté du passage d'une fente. Ces deux montages sont alimentés sous 3,3V entre alimentation (orange) et masse (bleue) numériques issues de la carte Arduino. On n'utilise pas les masse et alimentation analogiques, car elles sont trop agitées. Moyennant ces dimensionnements et précautions, la tension en sortie du phototransistor peut être exploitée directement par la carte Arduino. Cette tension est malgré tout sujette à de petites perturbations et, comme elle évolue lentement à l'échelle des temps numériques, il est délicat de décider quand elle effectue une commutation du 0 au 1 logique, et vice versa. L'ensemble du montage constitué par LED, phototransistor, résistances et autre amplificateur éventuel est usuellement appelé ≪interrupteur optique ≫.

4 Les fonctionnalités de base du SAM3x8E et de la carte Arduino

Nous allons maintenant nous intéresser à la programmation du MCU. Nous allons exploiter ses différentes unités en écrivant quelques programmes simples dont les principes serviront pour l'accomplissement global du projet.

4.1 Architecture d'un programme

Un programme sur Arduino est au moins constitué de 2 parties obligatoires :

- Une fonction `void setup (void)` appelée **1 fois** à l'initialisation du système (*reset*). Lorsque le transfert d'un programme sur la carte est terminé, le MCU effectue automatiquement un reset.
- Une fonction `void loop (void)` appelée en boucle sans fin après l'initialisation par `setup`.

Q1 : Débutez un nouveau programme (menu `Fichier` > `Nouveau`). Vous devez y trouver par défaut ces 2 fonctions vides. Compilez-le pour voir que ça vous donne un programme ... vide mais qui fonctionne correctement ... à ne rien faire.

4.2 Le port série

La carte Arduino intègre une communication bidirectionnelle via le port USB en mode série. Nous n'utiliserons cette communication qu'en mode «sortie» (i.e. pour envoyer des informations vers le PC). Ceci est particulièrement utile pour débugger (ou émettre des informations à une station de traitement plus puissante). Trois fonctions nous seront particulièrement utiles :

- `void Serial.begin (speed)` qui permet d'initialiser la communication avec l'ordinateur pour échanger des messages via un terminal ou la console de l'environnement de programmation à un vitesse donnée. On utilisera `speed` à 9600 (bits/s).
- `uint32_t Serial.println (val)` qui permet «d'afficher» une valeur (chaîne, entier, flottant, ...) sur le terminal ou la console de l'environnement de programmation, en terminant par un retour à la ligne.
- `uint32_t Serial.print (val)` qui fait comme `println` sans retour à la ligne final.

Q2 : Complétez votre programme vide de la question précédente pour lui faire initialiser la communication série et lui faire afficher un quelconque message. Pour ouvrir la console où s'affichent les messages, utilisez le menu `Outils` > `Moniteur Série` (ou son raccourci).

Q3 : Modifiez votre programme pour qu'il affiche le contenu d'une variable entière incrémentée à chaque appel à `loop ()`.

Attention : L'envoi sur le port série est une opération facile à faire, et qui semble bien pratique pour débugger. Toutefois, elle **très coûteuse** en **temps** et peut introduire un ralentissement de vos programmes, modifiant leur comportement «sans debug» !

Q4 : Imaginez une solution de debug lorsque la communication série s'avère trop lente.

5 Les IOs

Sans entrées/sorties un MCU n'a aucune utilité car il ne peut pas discuter avec ≪l'extérieur ≫. Un certain nombre de broches (*ports*) du MCU sont physiquement connectées aux broches de la carte Arduino. C'est par leur intermédiaire en **lecture** et **écriture** qu'il sera possible d'interagir avec ≪l'extérieur ≫. Nous nous intéressons ici aux ports *numériques*, c'est à dire capables de lire ou d'écrire des signaux *binaires* : `HIGH` ou `LOW`. Avant d'utiliser un port, il faut le positionner en **entrée** ou en **sortie**.

- `void pinMode (uint32_t pin, uint32_t mode)` qui configure une broche `pin` en `mode` pouvant être `OUTPUT` ou `INPUT`.
- `int digitalRead (uint32_t pin)` qui retourne la valeur du signal pouvant être `HIGH` ou `LOW` sur la broche `pin`.
- `void digitalWrite (uint32_t pin, uint32_t value)` qui positionne le signal `value` pouvant être `HIGH` ou `LOW` sur la broche `pin`.

Q1 : Le montage présente une LED de debug branchée sur la broche 5, un capteur infrarouge sur la broche 6 et le contrôle du moteur sur la broche 7. Quels les sont les ≪sens ≫ de ces broches ?

Q2 : Écrivez un programme qui configure les ports susdits et, toutes les secondes, effectue les actions suivantes :

1. Lit et affiche l'état du capteur infrarouge.
2. Bascule l'état allumé/éteint de la LED.
3. Bascule l'état allumé/éteint du moteur.

Vous avez à disposition la fonction `void delay (unsigned long ms)` qui suspend l'exécution pendant `ms` millisecondes.

En réalité, la fonction `digitalWrite` ne fait que des vérifications avant d'aller écrire dans un **registre** du MCU pour configurer le niveau électrique à positionner sur ce port. Pour rappel, un port correspond à un ensemble de broches physiques, encore appelées pins, qui permettent d'établir une connexion physique entre le MCU et le monde extérieur.

Les registres du MCU sont *mappés* en mémoire, c'est à dire que l'on peut y accéder comme une variable, à partir de leur adresse. On dit aussi qu'ils sont *adressables*. En particulier, les registres qui permettent d'écrire sur la broche 5 (la LED debug), sont les `PIO_SODR` (PIO Controller Set Output Data Register) et `PIO_CODR` (PIO Controller Clear Output Data Register) associés au port `C` du MCU. Leurs adresses sont respectivement 0x400E1230 et 0x400E1234 (écrit dans le *datasheet* du `SAM3x8E` ... attention, 1467 pages qui piquent les yeux). À chaque bit de ces registres est associée une broche. Le bit de la broche 5 est le 25. Mettre ce bit à 1 dans le `PIO_SODR` envoie `HIGH` sur la ligne. Mettre ce bit à 1 dans le `PIO_CODR` envoie `LOW` sur la ligne.

Q3 : Modifiez votre programme pour allumer et éteindre la LED en utilisant directement les registres.

6 Les interruptions

Un processeur vit dans un ≪monde clôt ≫ et passe son temps à y exécuter une boucle : récupérer l'instruction à exécuter, la décoder, l'exécuter. *A priori*, tout le futur du programme qu'il exécute est influencé uniquement par le contenu de la mémoire.

Pour ≪dialoguer ≫ avec ≪l'extérieur ≫, il lui faut être **interrompu** pour sortir de cet ≪isolement ≫. Le mécanisme d'*interruption* sert à cet effet. Une interruption est un évènement/signal **asynchrone** qui interrompt le processeur dans le code qu'il exécute. Le contexte du processeur est alors sauvegardé et le processeur saute à adresses fixes dépendantes de la nature de l'interruption pour y exécuter un gestionnaire d'interruption (*handler*) qui n'est autre qu'une fonction.

Q1 : À votre avis, quel est l'intérêt des interruptions ? Imaginez quelques exemple d'utilisation.

Pour déterminer quel handler exécuter lors de l'arrivée d'une interruption, il faut installer l'adresse de la fonction handler à exécuter à l'adresse fixe correspondant à l'interruption. Vous avez à votre disposition la fonction :

`void attachInterrupt(uint32_t pin, void (*callback)(void), uint32_t mode)`

qui permet de lier le handler `callback` à l'interruption liée à la broche `pin` lorsque l'évènement `mode` survient.

La fonction `callback` est de type `void` → `void`. Le `mode` détermine quel type d'évènement déclenche l'interruption. Il peut être :

- `CHANGE` : lorsque l'état électrique de la broche change (`LOW` ↔ `HIGH`).
- `RISING` : lors d'un front **montant** (`LOW` → `HIGH`).
- `FALLING` : lors d'un front **descendant** (`HIGH` → `LOW`).
- `LOW` ou `HIGH` que nous ne détaillons pas.

Sur le montage se trouve un phototransistor infrarouge positionné face à une LED infrarouge. Ce couple servira à détecter les passages des fentes de la roue. Le phototransistor envoie un signal `LOW` ou `HIGH` en fonction de son éclairage.

Q2 : Écrivez un programme qui installe un handler d'interruption sur la broche 6 pour gérer à votre choix `RISING` ou `FALLING` en faisant incrémenter le nombre d'interruptions détectées (dans une variable globale). Votre fonction `loop ()` devra afficher ce compteur toutes les secondes.

Attention : Votre variable globale mémorisant le nombre d'interruptions doit être déclarée comme `volatile long ir_changes_counter = 0;`. Le qualificateur `volatile` en C indique au compilateur ≪attention, cette variable peut être modifiée n'importe quand, même si tu ne vois pas où ni comment ! ≫. En effet, elle va être modifiée par votre handler qui n'est jamais appelé directement par votre code C (ce sont les interruptions qui le déclenchent). La déclaration `volatile` va alors empêcher le compilateur d'optimiser le réarrangement du jeu d'instructions concernant la variable déclarée comme `volatile` et plus globalement interdire toute optimisation dessus (mise en cache par exemple). Ne pas prendre cette précaution peut en effet mener à un comportement non désiré, étant donné que le compilateur suppose que la variable concernée est constante, alors qu'elle ne l'est pas dans les faits.

Q3 : Testez votre programme en occultant/libérant le récepteur infrarouge. Quel est le signal de sortie du récepteur en fonction de son éclairage ? Que remarquez-vous quant au nombre d'interruptions comptées, à quoi cela peut-il être dû ?

Q4 : Sachant que l'on va se servir du couple LED-détecteur pour déterminer le nombre de tours de roue pendant un laps de temps afin d'en déduite la vitesse, quel front (montant ou descendant) allez-vous surveiller sous interruption ?

Q5 : Proposez une solution pour éviter le problème détecté en **Q3**.

Indice Il existe une fonction **unsigned long** millis (**void**); qui retourne le nombre de millisecondes écoulées depuis que le Arduino a (re-)démarré. Ce ≪compteur ≫ n'est associé à aucune interruption et repasse à 0 (débordement) environ au bout de 50 jours.

7 Les timers

Jusqu'à présent, lorsque nous avons dû attendre un certain temps, nous avons utilisé la fonction `delay`. Pendant ce temps, le MCU était occupé ... à ne rien faire, ce qui est fâcheux.

Un MCU dispose de compteurs particuliers décrémentés tous les ≪un certain temps≫ : les *timers*. Le `SAM3x8E` dispose de 3 timers matériels avec chacun 3 canaux (on a donc 9 timers). Un timer est décrémenté à une fréquence diviseur de l'horloge de base du MCU (2, 8, 32, 128). Le choix du facteur de division se fait en écrivant dans un registre.

Un timer doit être initialisé à une certaine valeur. Lorsqu'il a été décrémenté et est arrivé à 0, une **interruption** est levée. On est donc averti du temps écoulé par interruption, pendant ce temps, le MCU peut faire autre chose.

Q1 : Nous souhaitons configurer un timer pour **1s**. Le facteur de division que nous choisissons est de **128**. Déterminez la valeur avec laquelle initialiser en fonction de : la vitesse du CPU (**#define**-ée par `VARIANT_MCK`), de la période de déclenchement voulue (`SAMPLING_FREQ`) et du facteur de division 128.

Deux fichiers source vous sont donnés (`Timer/timer1.*`) avec une fonction `start_timer_TC1` permettant d'initialiser et de démarrer le timer **1** du `SAM3x8E`.

Q2 : Dupliquez votre programme de la question **Q2** ou **Q3** de la section 5. Au lieu de la faire attendre 1s avec `delay`, utilisez le timer 1. Pour installer le handler d'interruption timer, **on n'utilise pas** `attachInterrupt`. Ce handler a un nom et un prototype fixe :

`void TC3_Handler ().`

Note : Retirez la ligne qui écrivait sur le port série l'état du receveur infrarouge (`Serial.println (analogRead (IR_PIN)) ;`). En effet, utiliser le port série dans une interruption ne fonctionne pas.

Attention : Le piège est qu'il ne faut pas oublier de réarmer le timer à la fin du traitement de l'interruption ! En effet, une fois qu'il a ≪sonné≫, si on ne le relance pas, il ne le fait pas tout seul. Pour ce faire, il faut appeler `TC_GetStatus (TC1, 0) ;` qui a pour effet d'effacer l'interruption et donc de réarmer le timer.

8 La PWM : moduler le moteur à courant continu

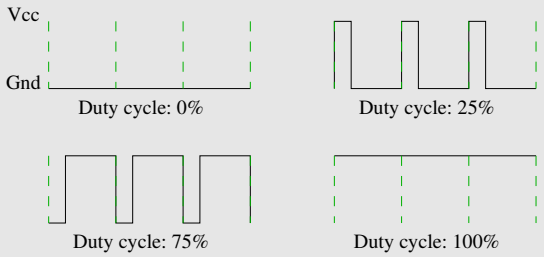
Dans la question **Q2** de la section 5, vous avez fait tourner le moteur à plein régime en positionnant `HIGH` sur la sortie reliée à sa commande.

Ceci a eu pour effet de commander en permanence au MOSFET d'être « passant ». Ainsi, le courant en provenance de l'alimentation moteur circulait en continu à sa tension maximale.

Q1 : Imaginez une idée pour que le moteur tourne moins vite ?

Comme introduit en section 3, La PWM (**P**ulse **W**idth **M**odulation est une technique qui permet de simuler des effets de courant analogique à partir d'impulsions en « tout-ou-rien ». La partie numérique est utilisée pour générer un signal carré (`HIGH` / `LOW`) en changeant la proportion de temps où celui-ci est à `HIGH` par rapport à `LOW` sur une période.

On appelle *duty cycle* le rapport entre le temps `HIGH` et `LOW` sur une période. Plus le duty cycle est important, plus la tension moyenne résultant sera élevée.



Sur Arduino, certains ports peuvent générer un signal PWM. C'est le cas de la broche 7 sur laquelle est branché le MOSFET. Pour générer un signal PWM, une fonction est directement disponible, qui se charge (grâce à un timer) de générer le signal carré sur une broche, avec le duty cycle demandé :

```
void analogWrite (uint32_t pin, uint32_t value);
```

Le paramètre `pin` est la broche, `value` est une valeur entre 0 et 255, 0 correspondant à un duty cycle de 0% et 255 à un duty cycle de 100%.

Q2 : Écrivez un programme qui fait varier la vitesse de rotation du moteur en jouant sur le duty cycle de la broche commandant le MOSFET. Vous pourrez par exemple le faire accélérer puis ralentir.

9 Assemblage : déterminer la vitesse

À ce point, vous savez :
- Contrôler des sorties **numériques**.
- Contrôler des sorties **analogiques**.
- Installer un **handler d'interruption** sur une broche.
- Mettre en route un **timer**.
- Envoyer des informations (éventuellement de debug) sur le port **série**.

Vous avez désormais toutes les connaissances pour surveiller la vitesse de rotation du moteur.

Attention : Pour la suite, il vous est très conseillé de découper votre programme en plusieurs fichiers, tout comme `timer1.*` l'a été. L'environnement de développement Arduino ouvre tous les fichiers du répertoire où se trouve votre `.ino` dans des onglets séparés et compile tout d'un coup.

Q1 : Qu'allez-vous surveiller sous interruption ?

Q2 : Qui va afficher (toutes les 1/2s) la vitesse que vous aurez calculée ?

Q3 : Comment allez-vous calculer la vitesse ?

Q4 : En reprenant les différentes parties que vous avez réalisées, écrivez le programme qui va afficher toutes les 1/2s la vitesse angulaire du moteur en **tours par seconde**.

10 Cirage de pompes

On peut voir dans certaines gares ou établissements fréquentés par des personnes soucieuses de la propreté de leurs chaussures, des cireuses à chaussures automatiques. Ce sont des brosses rotatives dont le but est de cirer ... les pompes.

Ces appareils sont usuellement équipés d'un bouton-poussoir pour déclencher la rotation. Afin de donner un coup de jeune à ces machines d'un autre temps nous souhaitons supprimer le bouton-poussoir et adopter le principe de fonctionnement suivant.

En absence de client, la brosse tourne à un régime faible pour montrer que l'appareil est fonctionnel et disponible.

Lorsqu'un client appose sa chaussure, un frottement apparaît sur la brosse, qui permet de détecter la présence de la chaussure. Il faut alors augmenter la vitesse pour compenser et obtenir pour un brossage efficace.

Si pour une raison quelconque le frottement est tel qu'il empêche la brosse de tourner, alors l'appareil doit se mettre en sécurité et couper le moteur pendant un certain temps (pour éviter qu'il ne grille).

La durée d'un brossage normal est considérée comme une constante et est signalée au client (par une LED).

Q1 : Transformez votre montage ... en cireuse automatique suivant le principe énoncé ci-dessus. Pour tester, vous pourrez agir avec les doigts sur la roue du moteur. Utilisez la LED pour signaler une fin de séance de cirage de pompes.

11 Pour aller plus loin...

Un PID en temps continu a pour expression :

$$u(t) = K_p e(t) + K_d \frac{de}{dt} + K_I \int_0^t e(s)ds,$$

avec $e(t)$ l'erreur mesurée à l'instant t. Dans le cas discret, qui nous concerne, en notant T la période d'échantillonnage, $e(n)$ l'erreur mesurée, et $u(n)$ la commande envoyée à l'instant $t = nT$ ($n \in \mathbb{N}$), la commande PID discrétisée devient :

$$u(n) = K_p e(n) + \frac{K_d}{T}\big(e(n) - e(n-1)\big) + K_I T \sum_{k=0}^{n} e(k)$$

C'est cette commande qu'il faut envoyer au moteur. On voit que la durée T entre 2 mesures doit être prise en compte. Pour éviter d'avoir de mauvaises surprises, en pratique il convient de s'assurer que les mesures des capteurs et l'envoi des commandes se font à une période d'échantilonnage fixe T. Pour cela, on utilisera bien sûr un timer.

Par ailleurs, notre système n'ayant que 2 points de signalement par tour, la mesure de la vitesse angulaire ne sera pas extrêmement précise. Cette imprécision va bien sûr limiter les performances du contrôleur.

Q1 : Commençons alors les choses simplement : implémentez un simple correcteur proportionnel.

L'utilisation d'un terme intégral peut amener à de mauvaises surprises. En effet, si les erreurs s'accumulent trop en fonction du temps, le terme $K_I \int_0^t e(s)ds$ devient beaucoup plus grand que le terme $K_p e(t) + K_d \frac{de}{dt}$, et c'est la catastrophe. Pour éviter ce désagrément, on utilise en pratique un correcteur PID avec «anti-emballement» («anti wind up») qui peut par exemple prendre l'expression suivante :

$$u(t) = S^{at}\big(K_p e(t) + K_d \frac{de}{dt}(t) + I(t)\big),$$

$$\frac{dI}{dt}(t) = K_I e(t) + K_s\big(u(t) - K_p e(t) - K_d \frac{de}{dt}(t) - I(t)\big)$$

avec :
$S^{sat}(u) = u_{min}$ si $u < u_{min}$.
$S^{sat}(u) = u$ si $u_{min} < u < u_{max}$.
$S^{sat}(u) = u_{max}$ si $u > u_{max}$.

On remarque bien que lorsque la commande sature à son maximum, pour des gains bien choisis (par exemple, pour un PI, il faut $K_s K_p > K_I$), la croissance du terme intégral est ralentie, voire diminue si la commande calculée est trop grande par rapport à u_{max}.