TAlone

CC BY

Zeste de savoir

Arduino : les secrets de l'analogique

12 août 2019

Table des matières

1. L'AtMega, processeur d'Arduino . . . 2
   1.1. AtMega et AVR . . . 2
   1.2. Registres et introduction à l'analogique avec AVR . . . 2
   1.3. Un peu plus loin avec les registres . . . 4
   1.4. Brochage sur Arduino . . . 4
2. Base de l'analogique et référence de tension . . . 4
   2.1. Architecture de l'analogique . . . 5
   2.2. Les références internes . . . 6
   2.3. Créez votre référence . . . 7
3. ADC et fast-sampling . . . 10
   3.1. Prescaler et ADC . . . 10
   3.2. Code et résultats . . . 11
   3.3. Free-running mode . . . 12
4. Comparaison directe avec Arduino . . . 14
   4.1. Qu'est-ce qu'un comparateur . . . 14
   4.2. Un comparateur inconnu, mais rapide . . . 15
   4.3. Utiliser une entrée de l'ADC . . . 17
5. Quelques bonus de la datasheet . . . 19
   5.1. La fonction analogRead au complet (ou presque) . . . 19
   5.2. Le capteur de température intégré . . . 20
   5.3. Un peu d'analogique : L'étage d'entrée de l'ADC . . . 22

Cet article n'est pas destiné aux débutants à Arduino ; pour un tutoriel, allez voir le cours d'Eskimon et olyte .
Une **connaissance minimale** du langage est présumée pour suivre cet article, vous aurez aussi plus de facilité si vous savez travailler avec les registres ; toutefois, ces notions sont abordées rapidement si ce n'est pas le cas.

Nous allons aujourd'hui nous intéresser aux concepts que le langage Arduino cache volontairement afin de simplifier le développement, particulièrement au niveau des mesures analogiques, ces fonctions sont avancées mais très utiles dans certains cas particuliers.

Dans un premier temps, nous étudierons rapidement l'architecture d'un Arduino, pour voir notamment comment fonctionne son processeur. Une fois ces bases posées, nous pourrons voir comment « améliorer » votre Arduino avec quelques bouts de code.

1. L'AtMega, processeur d'Arduino

Cette partie est particulièrement théorique et complexe, lisez-la tranquillement, et si certaines choses ne sont pas très claires, n'hésitez pas à passer à la suite tout de même, vous devriez comprendre les codes.

1.1. AtMega et AVR

Vous le saviez peut-être, ou vous vous en doutiez, l'Arduino est basé sur un microcontrôleur (le gros bloc noir au centre sur la Uno). Celui-ci diffère en fonction du type de votre Arduino (Uno, Mega, M0, Yún, Micro, …) ; tout ce qui est décrit dans cet article ne fonctionne avec certitude que pour les Arduino basés sur un processeur AtMega, avec une architecture AVR, nous allons revenir sur ces termes dans un instant ; pour cette raison, il existe trois cartes pour lesquelles il est probable que le code de cet article ne fonctionne pas : il s'agit des Arduino Due, Zero ou 101. Les codes ont été testés sur les cartes Nano, Uno et Mega, donc ils fonctionnent avec certitude sur ces cartes, mais il n'y a aucune raison qu'ils ne fonctionnent pas sur les autres cartes – hormis celle indiquées.

Bien, maintenant que vous avez la bonne carte, nous pouvons nous intéresser au contenu de celle-ci : le processeur d'Arduino s'appelle AtMega et est basé sur une architecture nommée AVR. La programmation directe d'un AVR peut s'avérer compliqué, c'est pourquoi le projet Arduino a été créé ; l'objectif de ce projet est de fournir une carte embarquant tous les composants nécessaires au bon fonctionnement de l'AtMega, ainsi qu'une base de code pour un développement simplifié. Cette base de code est utilisable pour la plupart des projets, mais il y a quelques fois où il nous faudrait aller plus loin, pour cela, nous pouvons envoyer des instructions directement au processeur AVR, tout en gardant une partie de la couche d'abstraction que met à notre disposition le projet Arduino.

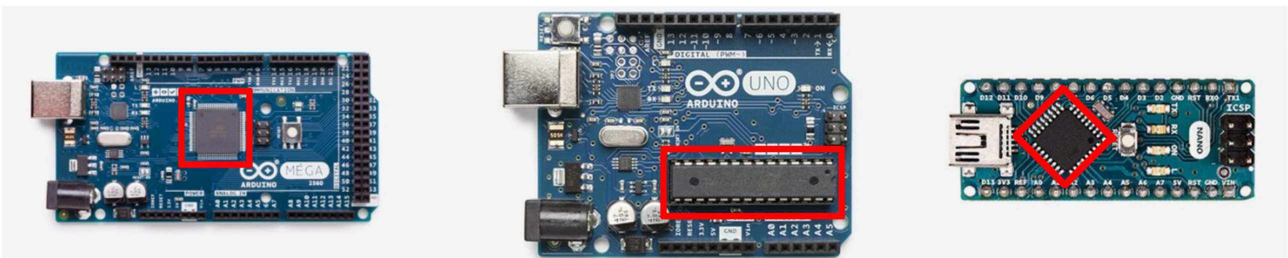

Figure 1. – Repérage des processeurs sur trois cartes

1.2. Registres et introduction à l'analogique avec AVR

Afin de communiquer avec le processeur, nous allons devoir utiliser ce que l'on appelle des registres : il s'agit en quelque sorte de variables, souvent sur un octet, dans lesquelles nous allons donner au processeur des instructions et des entrées, puis il va écrire sur un autre registre des sorties. Cette façon de fonctionner est dite très bas niveau – c'est-à-dire très proche du matériel, c'est pourquoi on s'en passe le plus possible lors de développement Arduino. Ce n'est

pas très clair dit comme ça, alors voici un exemple simple de registre dont nous allons nous servir plus tard, le registre ADCSRA :

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

C'est peut-être encore plus obscur maintenant, mais voici ce qu'il faut savoir : le bit 6, lorsqu'il est passé à 1, déclenche ici une fonction semblable à `analogRead`, c'est d'ailleurs en passant par ce registre qu'Arduino peut appeler cette fonction. Deux fonctions nous seront particulièrement utiles pour travailler avec les registres : les fonctions `sbi` et `cbi`, qui vont passer, respectivement, un bit d'un registre à 1 ou à 0. Par exemple, en reprenant notre exemple ci-dessus, pour faire au niveau du processeur l'équivalent d'un appel à `analogRead`, il faut utiliser :

```
// Ne prenez pas ça en compte pour le moment, c'est expliqué plus bas
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

void analogRead() {
  sbi(ADCSRA, ADSC);
}
```

Comme vous pouvez le voir, la fonction prend d'abord le nom du registre, puis le nom du bit du registre. Il y a deux choses à remarquer ici :

- déjà, un bloc de code bizarre en haut ; il s'agit d'un bout de code permettant, si les fonctions ne sont pas définies sur la carte, de les définir nous-même. Ce n'est donc pas important et je ne vais pas le mettre dans les exemples partiels, vous pouvez essayer sans, et si votre code ne fonctionne pas, tentez de les ajouter, cela pourrait corriger le problème ;
- ensuite, plus intéressant, notre fonction `analogRead` est de type `void`, donc elle ne retourne rien ; de plus, elle ne prend aucun argument, cette fonction est donc parfaitement inutile.

Rassurez-vous, c'est normal, nous avons simplement dit au processeur de lire un pin, sans lui préciser lequel (par défaut, c'est le A0), et de mettre cette valeur dans un registre, ce qu'il a fait. La véritable fonction `analogRead` contient, en plus, du code pour sélectionner le pin à lire et pour lire la valeur de sortie, mais le code complet ne nous intéresse pas ici. Cet exemple n'est ici que, d'une part, pour vous expliquer le principe des registres, d'autre part, pour montrer la complexité du développement AVR.

1.3. Un peu plus loin avec les registres

En plus des fonctions `sbi` et `cbi`, nous utiliserons dans cet article quelques autres fonctions de base, que je vais rappeler rapidement ici. Pour manipuler les registres, nous aurons souvent recours au masquage, qui en langage Arduino se traduit comme suit :

```
ADCSRA |= 1 & 0b01000000;
```

Ce code est le même que celui ci-dessus, ou tout au moins il fait la même chose : on applique un 1 sur le 7ème bit du registre (bit n°6). Plus souvent, on préfère traduire la partie binaire, assez longue, en hexadécimal, ce qui donne :

```
ADCSRA |= 1 & 0x80;
```

Un autre opérateur qu'il me semble important de rappeler (ou plutôt deux), est le double-chevron (`<` ou `>`), qui décale la valeur en binaire de n crans vers la gauche ou vers la droite.

1.4. Brochage sur Arduino

Certains pins de notre AVR n'ont pas le même nom sur l'Arduino, ainsi, je vais tenter de vous montrer ici la correspondance au niveau des pins analogiques entre AVR et Arduino. Nous allons mentionner ici les pins analogiques A0 à A5, équivalent à d'autres pins sur l'AVR ; ces pins sont différents en fonction de votre type d'Arduino, je vous propose donc un petit tableau :

Arduino	Proc. Uno	Proc. Mega	Proc. Nano
A0-5	23-28	97-92	23-28
A6 et A7	/	91 et 90	19 et 22
A8-15	/	89-82	/

Si vous avez un autre Arduino, pas de panique, il est probable que l'une des cases ci-dessus corresponde, vous pouvez aller voir la page Wikipédia , toutes les cartes ayant le même processeur ont les mêmes pins, en général. Pour les autres pins, nous les verrons au cas par cas lorsque nous en aurons besoin.

2. Base de l'analogique et référence de tension

Dans cette partie, nous allons voir l'architecture de fonctionnement de l'analogique sur AtMega, ainsi qu'un de ses principes de base : la référence de tension.

2.1. Architecture de l'analogique

Le système permettant à Arduino de lire une valeur analogique est appelé ADC (ou CAN en français), pour Analog-to-Digital Converter (ou Convertisseur Analogique-Numérique), ce système permet à l'AtMega, qui est processeur numérique, de lire des valeurs analogiques, pour cela, il les convertit, justement, en valeurs numériques. Un ADC coûte plutôt cher à produire, c'est pourquoi chaque processeur AtMega n'en contient qu'un seul ; alors pour permettre d'utiliser 6 à 15 pins analogiques différents, il est nécessaire de recourir à un multiplexeur, qui est une sorte de sélecteur, permettant de choisir quel pin de l'Arduino va aller vers l'ADC, et fournir une valeur numérique. Prenons comme exemple les processeurs contenant 8 pins analogiques – ce qui est le cas de la plupart, mais ils ne sont pas accessibles sur certains packages, comme sur celui utilisé dans la carte Uno ; le registre ADMUX est agencé comme suit :

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
REFS1	REFS0	ADLAR	/	MUX3	MUX2	MUX1	MUX0

Les bits intéressants ici sont les quatre derniers, MUX[0-3], ce sont ces bits qui sélectionnent quel pin analogique sera utilisé, voici un petit tableau pour la correspondance :

MUX[3-0]	**Pin AVR**	**Pin Arduino**
0000	ADC0	A0
0001	ADC1	A1
0010	ADC2	A2
0011	ADC3	A3
0100	ADC4	A4
0101	ADC5	A5
0110	ADC6	(A6)
0111	ADC7	(A7)
1000	Spécial[1]	/

Ces pins sont les mêmes sur la carte Mega, avec un petit changement : le multiplexeur dispose de 6 bits (et non 4), on retrouve ainsi les mêmes valeurs binaires que sur la Uno, le pin « Spécial » en moins et avec les pins A[8...15] en plus sur les adresses [100000...100111]. Encore une petite subtilité ? Le bit MUX[5] ne se trouve pas sur ADMUX, mais sur ADCSRB[3]. Si ce n'est pas très clair, reportez-vous à la fonction analogRead donnée en dernière partie.

Nous pouvons ainsi compléter un petit peu notre fonction analogRead de tout à l'heure :

```
void analogRead(uint8_t pin) {
  // On sélectionne notre pin
  ADMUX |= pin & 0x07;
  // On lance la conversion
  sbi(ADCSRA, ADSC);
}
```

À noter que cette fonction permet d'utiliser uniquement des nombres pour les pins (0, 1, 2, …), mais pas les raccourcis qu'Arduino met à notre disposition (A0, A1, A2, …). Vous pouvez tester, rien ne va exploser, mais comme ces pins correspondent en interne à des nombres de 14 à 21, et que nous avons ajouté un masque (le 0x07, 0000 0111 en binaire), le pin lu sera celui correspondant en décimal aux trois derniers bits du nombre correspondant à l'entrée analogique – c'est à ça que sert le masque, il n'écrit que sur les trois derniers bits du registre.

Un petit aparté, lorsque l'on touche directement aux registres, aucune vérification n'est effectuée par le système, donc, comme on dit, « *never trust user input* », il faut vérifier chaque chose, et veiller à ce que les masques soient exacts, notamment. Notons enfin que le code réel d'`analogRead`, en plus de lire la sortie du processeur, contient des instructions spécifiques pour chaque processeur, chacun ayant une façon spécifique de gérer son registre (mais le code ci-dessus devrait être bon pour toutes les cartes).

2.2. Les références internes

Deux autres bits du registre ADMUX ci-dessus sont intéressants, les bits REFS[0-1], qui permettent de sélectionner la référence de tension utilisée par l'AtMega. Je pense que vous ne voulez plus voir de tableau, alors rassurez-vous, cette partie est très bien gérée par l'Arduino, et il nous sera donc bien inutile de toucher directement à ces bits. Je vous propose alors de voir comment utiliser ces références avec l'Arduino : le principe est que la tension de référence utilisée doit être inférieure ou égale à la tension maximale mesurée, et l'ADC du processeur va « créer » 1024 niveaux de tension entre 0V et la tension de référence, pour ensuite vérifier à quel niveau correspond notre entrée analogique.

Bien, l'Arduino contient donc déjà quelques références internes, pour ça je vais simplement citer le tutoriel d'Eskimon mentionné en début d'article, pour rappel :

- DEFAULT: la référence de 5V par défaut (ou 3,3V pour les cartes Arduino fonctionnant sous cette tension, telle la Due) ;
- INTERNAL: une référence interne de 1.1V (pour la Arduino Uno) ;
- INTERNAL1V1: comme ci-dessus mais pour la Arduino Mega ;
- INTERNAL2V56: une référence de 2.56V (uniquement pour la Mega).

Pour les utiliser, rien de plus simple, il suffit de passer un de ces noms à la fonction `analogReference()`. Globalement, on a donc accès à trois (ou quatre sur la Mega) références simples d'utilisation, 5V, 3.3V (expliqué juste après, en connectant un seul fil), éventuellement 2.56V, et 1.1V ; c'est suffisant pour la plupart des applications, mais imaginons que nous souhaitions mesure entre 0 et 2 Volts, ou entre 12 et 15V, ce n'est pas optimal dans un cas, et dangereux dans l'autre. Dans la partie suivante, nous allons voir comment éviter ce problème en créant nos

propres références de tension ; mais avant cela, je vous propose un schéma récapitulatif de ce que nous avons vu dans ces deux premières parties.

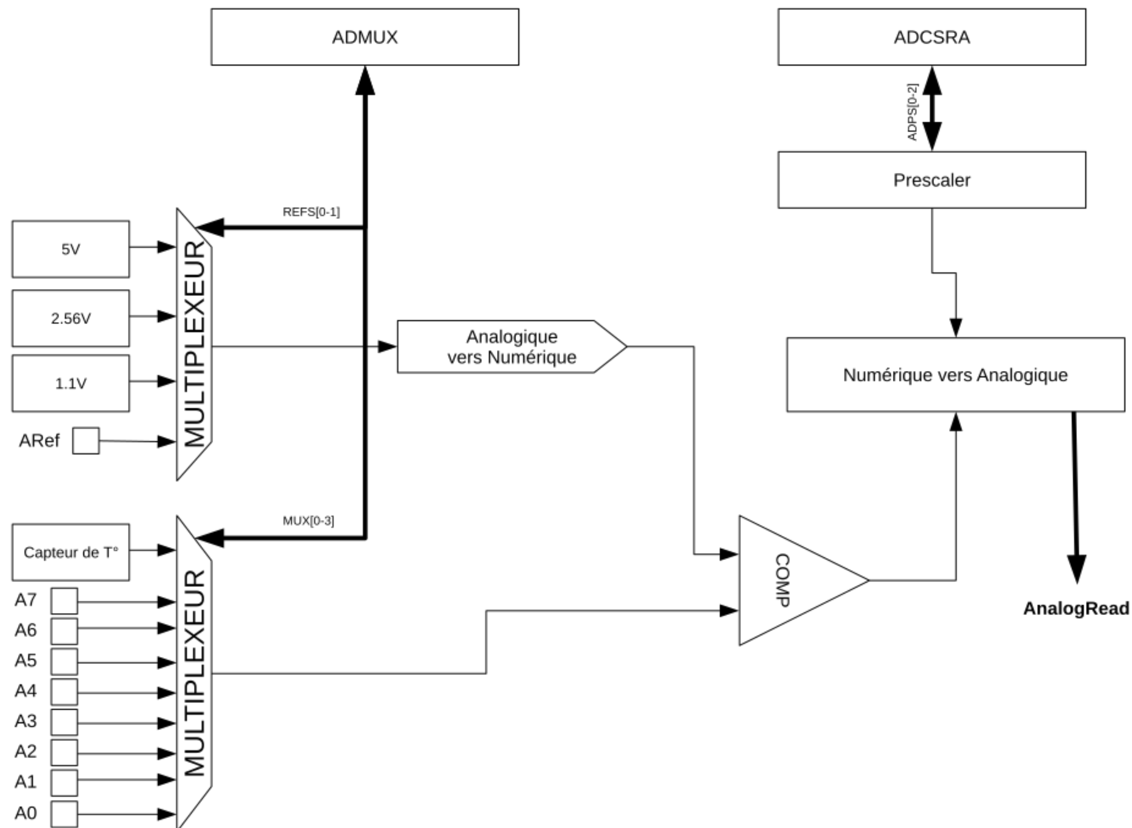

FIGURE 2. – Schéma bilan

Je pense que la majorité du schéma se passe de commentaire, sauf peut-être la partie Prescaler : pour le moment, sachez juste qu'elle existe et qu'elle sert globalement à « configurer » l'ADC, nous en reparlerons dans la partie suivante.

2.3. Créez votre référence

> *i* Cette partie est plutôt orientée électronique, contrairement au reste du tutoriel, si ce n'est pas votre fort, vous pouvez la zapper et utiliser simplement les références internes.

Pour cette partie, il faudra utiliser une valeur d'`analogReference` non-mentionnée dans le paragraphe ci-dessus, il s'agit de EXTERNAL ; le principe de ce pin est de prendre comme référence haute une tension qu'on lui donne, et il faut placer cette tension sur la broche ARef.

2.3.1. Une référence supplémentaire en 3.3V

Cette sous-partie sera très courte, puisqu’elle mentionne seulement une possibilité offerte par l’Arduino, qui dispose d’un régulateur 3.3V intégré. Le principe est simplement de relier la broche 3.3V de l’Arduino directement au pin ARef, ce qui permet, en plus des références internes d’avoir, pour le prix d’un câble, une référence sympa, proche d’1/2 Vcc (moitié de la tension d’alimentation, 5V).

2.3.2. Votre propre référence

Entrons maintenant dans le vif du sujet, nous allons réaliser notre propre référence de tension pour Arduino, pour cela, voyons d’abord, afin d’ajuster la valeur de nos composants, quelle est l’impédance d’entrée du pin ARef que nous allons utiliser ; la datasheet nous la donne à 32kOhm en moyenne, il faudra donc faire attention à ce que l’impédance de sortie de notre référence de tension ne soit pas supérieure à 3.3kOhm (valeur normalisée la plus proche d’un dixième de $R_{A_{Ref}}$).

En tout cas, veillez à ce que votre référence de tension ne puisse pas dépasser la tension maximale supportée par l’Arduino (bien souvent 5 Volts).

Voyons donc trois moyens de faire cette référence de tension :

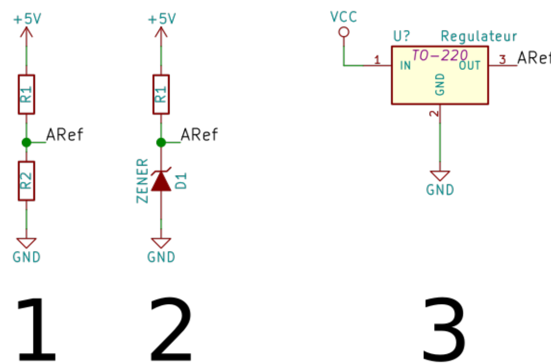

Figure 2. – Diverses références de tension

2.3.2.1. Avec un pont diviseur La première manière de faire est d’utiliser un pont diviseur de tension, c’est ce qui est montré sur la figure 1 ; le principe, sans rentrer dans le fonctionnement, est que la valeur de tension en sortie (entre les deux résistances) représente une fraction de la valeur de tension en entrée, si vous cherchez, la formule est la suivante :

$$V_{out} = V_{in} \times \frac{R2}{R1 + R2}$$

Et par le MET (si vous ne connaissez pas, ce n’est pas grave), la résistance de sortie est de :

$$R_{out} = R1//R2 = \frac{R1R2}{R1 + R2}$$

Par conséquent, utiliser des valeurs de l’ordre du kilo-ohm est fréquent, afin de ne pas charger le montage par la résistance 32k de l’Arduino. Notons que pour diviser par deux une valeur de tension en utilisant ces ponts il faut que R1 soit de la même valeur que R2. Ce montage est fonctionnel, mais il présente deux grands inconvénients :

— il est imprécis, la tolérance des résistances communes étant de 5 %, l’imprécision peut être grande, des résistances de 1 % ou moins sont donc conseillées ;
— il est peu pratique au niveau des calculs, il faut calculer sa tension et s’assurer que l’impédance de sortie est assez faible.

Ce montage est très intéressant, mais moins pour la broche ARef, le principe d’utilisation courant avec l’Arduino est qu’il permet de mesurer des tensions supérieures à 5V, en divisant la tension d’entrée, ainsi, une entrée entre 0 et 10V, un pont diviseur avec R1=R2, et le tour est joué, vous avez une plage de mesure de 0 à 5V que vous pouvez brancher directement sur vos pins analogiques.

2.3.2.2. Avec une diode zener Un composant permet une meilleure précision, avec un calcul bien plus simple : il s’agit de la diode zener ; ce composant donne une valeur bien fixe de tension, pour laquelle il a été conçu, pourvu qu’on y fasse passer un courant fixe (il peut varier un peu en pratique). Dans notre cas, ce ne sera pas bien difficile, il suffit d’y rajouter une résistance en série et vous obtenez le montage 2, à noter, le seul calcul ici est celui de la valeur de la résistance, qui vaut :

$$R1 = \frac{V_{cc} - V_z}{I_z}$$

Dans la plupart des cas, $I_z = 5mA$, donc l’inconvénient de ce montage est de consommer un peu de courant ; aussi, Vcc vaut 5V si vous alimentez le montage avec la tension d’Arduino. On ne se préoccupe pas ici de l’impédance de sortie du montage car, traversée par 5mA, une diode zener de moins de 5V présente une résistance dynamique de moins d’un kilo-ohm, et la résistance de limitation de courant sera généralement de l’ordre du kilo-ohm également. Ce montage est un grand classique pour faire une référence de tension et je le recommande grandement, au vu de sa fiabilité, de son très faible coût, et de sa simplicité d’utilisation.

2.3.2.3. Un régulateur spécifique ? Certains circuits sont proposés en tant que référence de tension, ils sont conçus pour ce type d’utilisation, et leur impédance de sortie est généralement suffisamment basse pour permettre de les brancher directement à un microcontrôleur.

Ces circuits intégrés ont une utilisation de base simple (cf. montage 3), mais il faut en règle générale mettre un ou deux condensateurs sur leurs entrées et sorties, ce qui rend encore plus chère l’utilisation de ces composants déjà onéreux.

Je ne recommande pas ces circuits, spécifiquement parce que leur rapport efficacité / prix est bien inférieur à celui d’une diode zener, sachez donc que cela existe si vous avez besoin d’une précision extrême, mais je doute qu’il y ait beaucoup d’applications en amateur.

1. Mesure spécifique que nous verrons en bonus, spoiler : c’est un capteur de température.

3. ADC et fast-sampling

La fonction `analogRead`, sur un Arduino, s'exécute en 115 µs environ, nous allons voir dans cette partie comment réduire ce temps d'exécution, en passant directement par les fonctions de l'AtMega.

3.1. Prescaler et ADC

Vous vous rappelez peut-être du bloc « prescaler » de la partie précédente, je vous avais dit qu'elle servait à « configurer » l'ADC ; plus précisément, le prescaler va diviser la fréquence de fonctionnement de l'AtMega (16 MHz sur un Arduino), par un facteur de division donné, afin de laisser le temps à `analogRead` d'obtenir une mesure fiable, ainsi que pour avoir une bonne précision au niveau de la valeur lue. Ce prescaler est configuré au niveau du code d'Arduino afin d'obtenir un rapport de division de 128, ce qui est un rapport très important, la raison de ce choix est que les Arduino tournent aujourd'hui pour la plupart à 16MHz, mais le code d'Arduino doit être portable, et afin de tourner sur des fréquences aussi faibles de 1MHz en conservant la même fréquence pour l'ADC (afin d'en conserver la précision), il est impératif de mettre un rapport de division bien plus important pour les grandes fréquences ; voici un morceau du code de configuration d'`analogRead` sur Arduino :

```
// Si la fréquence du processeur est de 16MHz
#if F_CPU >= 16000000 // 16 MHz / 128 = 125 KHz
  sbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);
```

Comme vous pouvez le voir dans ce morceau de code, au niveau de l'AtMega, le rapport de division du prescaler est configuré par trois bits (ADPS[0-2]) du registre ADCSRA (nous l'avions mentionné sur notre schéma récapitulatif), ces trois bits fonctionnent de la façon suivante :

ADPS2	ADPS1	ADPS0	Facteur de division
1	1	1	128
1	1	0	64
1	0	1	32
1	0	0	16
0	1	1	8
0	1	0	4
0	0	1	2
0	0	0	2

3.2. Code et résultats

Ces trois bits se trouvent à la fin du registre ADCSRA, dont nous avons déjà parlé – rappelez-vous, c'est le registre contenant ADSC, le bit permettant de lancer une conversion analogique-numérique. Nous allons pouvoir nous servir de ces trois bits ainsi que de nos fonctions favorites (`cbi` et `sbi`), afin de régler le prescaler sur une fréquence supérieure (et donc un ratio de division inférieur), par exemple, nous pourrions passer le rapport de division à 16, permettant une lecture 8 fois plus rapide, en utilisant ce morceau de code – à mettre dans le `setup` :

```
sbi(ADCSRA, ADPS2);
cbi(ADCSRA, ADPS1);
cbi(ADCSRA, ADPS0);
```

Ensuite, il est possible de faire une mesure simplement en appelant de façon normale la fonction `analogRead`. Voici donc que notre prescaler tourne à une fréquence d'1 MHz, nous devrions donc avoir une mesure toutes les :

$$\begin{aligned} t_{mesure} &= \frac{1}{F_{mesure}} \\ &= \frac{r_{div} \times nb_{cycles}}{F_{arduino}} \\ &= \frac{16 \times 13}{16 \cdot 10^6} \\ &= 13 \cdot 10^{-6} \qquad (13s) \end{aligned}$$

En pratique, le temps entre deux mesures est compris entre 16 et 20 µs, soit un peu plus que la théorie ; afin de ne pas vous encombrer avec la formule ou des mesures expérimentales, je vous propose de tableau résumant les temps et fréquences théoriques et pratiques obtenus sur un Arduino Uno :

Rapport de division	Temps théorique	Temps min.	Temps max.	Fréquence moyenne
128	104 µs	112 µs	116 µs	8.7 kHz
64	52 µs	54 µs	64 µs	17.2 kHz
32	26 µs	32 µs	36 µs	30.3 kHz
16	13 µs	16 µs	20 µs	55.5 kHz
8	6.5 µs	12 µs	20 µs	71.4 kHz
4	3 µs	8 µs	12 µs	100 kHz
2	1.5 µs	4 µs	12 µs	125 kHz

Il faut toutefois mettre l'accent sur quelque chose d'important : plus la vitesse de l'ADC est élevée, plus la précision est faible, la datasheet garantit une vitesse maximale de 76 kHz (cf.

§24.1), mais, pour une précision maximale, la vitesse maximale est de 15 kHz, un rapport de 64 fait donc **déjà perdre en précision**.

Afin d'approfondir, notez que cette partie – en particulier les morceaux sur le prescaler, ont fait l'objet d'un billet , n'hésitez pas à le lire pour vous entraîner à ces nouvelles notions ; vous y trouverez en particulier des codes pratiques pour mesurer la vitesse d'`analogRead`.

3.3. Free-running mode

Si vous avez besoin d'effectuer plusieurs mesures analogiques à la suite, il pourrait sembler bon de laisser l'ADC effectuer des mesures en continu, et de les récupérer dès que possible, pour cela, il existe le « free-running mode » de l'ADC. Le principe est que le convertisseur analogique-numérique se déclenchera automatiquement à nouveau dès qu'il aura fini de lire la valeur précédente. Ce n'est pas le comportement par défaut d'Arduino, puisque le convertisseur est normalement déclenché dès l'appel à `analogRead`, voyez le schéma suivant afin de comprendre un peu mieux comment est déclenché le convertisseur :

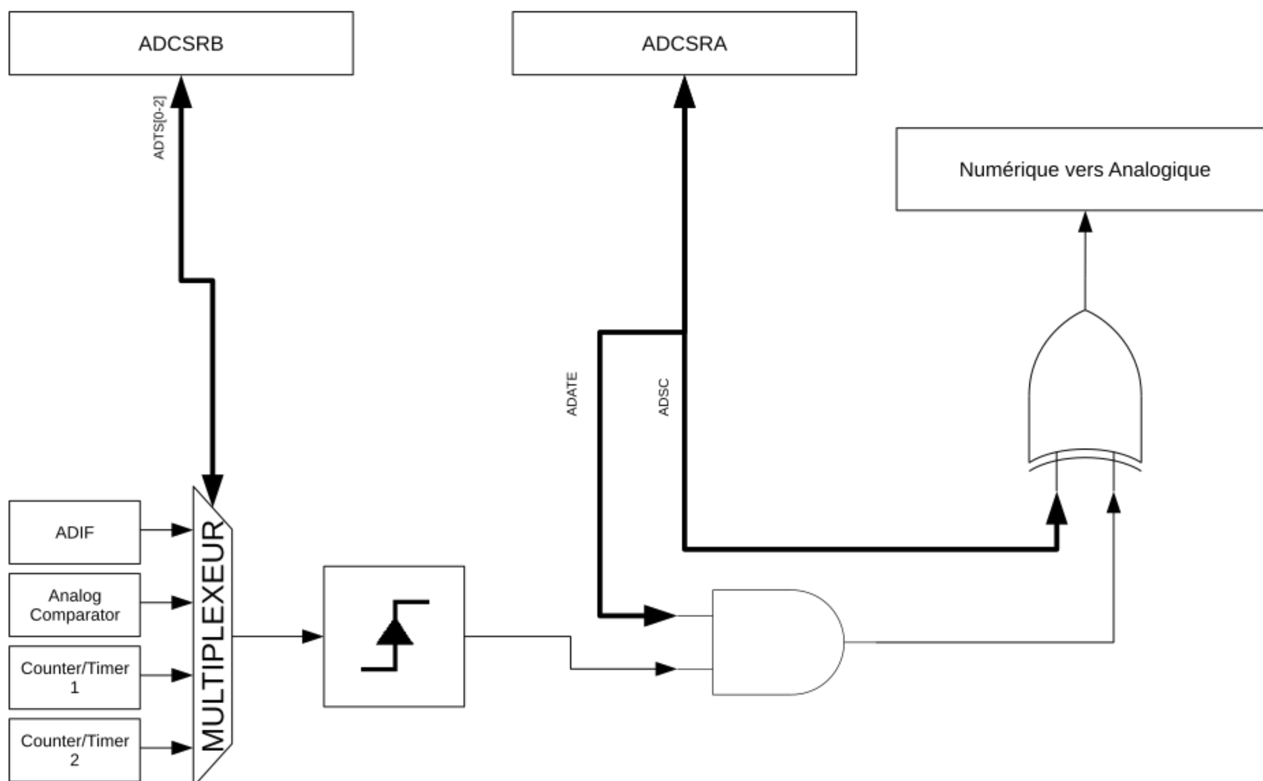

Figure 3. – Schéma activation du convertisseur

Cette image, en plus du registre ADCSRA, introduit un autre registre que nous n'avons pas encore vu, il s'agit de ADCSRB, qui lui est complémentaire, ce registre est grandement inutilisé :

7	6	5	4	3	2	1	0
/	ACME	/	/	MUX5[2]	ADTS2	ADTS1	ADTS0

Nous ne verrons pas le bit ACME ici, pour la simple raison qu'il n'est pas utile pour l'ADC – nous le traiterons dans la partie sur le comparateur. Mais les trois bits ADTS[2-0] vont grandement nous intéresser, puisqu'ils permettent de sélectionner par quoi sera déclenché le convertisseur ; nous verrons ici simplement le cas où ADTS = 0b000, puisque les autres se réfèrent à des notions trop éloignées du sujet de l'article. Ce cas est celui où le convertisseur sera déclenché par ADIF, c'est-à-dire le registre activé en fin de conversion, permettant à notre ADC de boucler pour toujours, lisant continuellement la valeur de notre pin. Bien, il est l'heure de passer à la pratique, voici un morceau de code permettant de lire la valeur d'un pin en utilisant le free-running mode :

```
void setup() {
  // Désactivons l'ADC pour l'instant
  cbi(ADCSRA, ADEN);

  // Activation du free-running mode
  ADCSRB = 0x00;
  // On sélectionne notre pin (A0)
  ADMUX |= 0 & 0x07;
  // Important : préciser la référence de tension (ici, équivalent
      de DEFAULT)
  ADMUX |= (1 << REFS0);

  // Choix de la division du prescaler (ici, facteur 8)
  cbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);

  // Ce bit doit être passé à 1 pour prendre en compte le
      free-running
  sbi(ADCSRA, ADATE);

  // Demande d'une interruption à la fin de la conversion
  sbi(ADCSRA, ADIE);

  // Réactivons l'ADC
  sbi(ADCSRA, ADEN);
  // On lance la première conversion
  sbi(ADCSRA, ADSC);

  // Ne pas oublier d'activer les interruptions matérielles
  sei();
}

void loop() {
  // Code à exécuter lors d'une mesure analogique
  int value = (ADCH << 8) | ADCL;
}
```

Voilà, rien de bien compliqué hormis que la valeur n'est pas lue en passant par `analogRead`,

mais directement depuis les registres de sortie. Ce code fonctionne bien, mais il est possible de l'améliorer en réalisant un code asynchrone, c'est-à-dire que le code de votre application va s'exécuter normalement, et vous aurez en parallèle un code qui s'exécutera dès qu'une nouvelle valeur analogique aura été lue ; pour faire ceci, ce n'est pas compliqué, il faut tout d'abord ajouter en fin de `setup` un `sei();`, c'est une fonction qui va activer les interruptions matérielles, et nous permettre d'exécuter du code asynchrone ; ensuite, utilisez votre `void loop` comme vous le souhaitez, et utilisez le code suivant pour détecter une nouvelle mesure de l'entrée analogique :

```
ISR(ADC_vect) {
  // Code à exécuter lors d'une mesure analogique
  int value = (ADCH << 8) | ADCL;
}
```

Cette syntaxe est probablement nouvelle pour vous, mais elle ne doit pas vous déstabiliser, voyez simplement ça comme une fonction qui serait déclenchée par l'AtMega ; notez qu'il ne faut pas la mettre dans votre `setup` ou votre `loop`, mais bien comme si vous déclariez une nouvelle fonction globale.

Faites bien attention à deux choses lors de vos tests, en particulier avec `Serial.print()`, il sera nécessaire d'initialiser la sortie série sur plus des 9600 bauds courants, à cause de la rapidité de mesure ; aussi, notez qu'il n'est pas recommandé d'appeler une quelconque fonction Arduino dans l'interruption, il faut appeler que des fonctions AVR ; stockez donc votre valeur lors de l'interruption, pour l'utiliser dans `loop`.

4. Comparaison directe avec Arduino

Jusqu'ici, nous avons beaucoup parlé d'un convertisseur analogique-numérique, mais si nous avons besoin d'effectuer une comparaison entre deux valeurs, utiliser deux pins analogique avec un morceau de code est plutôt complexe, et surtout relativement lent ; pour effectuer ce genre de comparaisons, nous avons régulièrement recours à un comparateur, bien plus rapide et efficace.

4.1. Qu'est-ce qu'un comparateur

Un comparateur est un élément relativement simple, il dispose de trois broches, une broche + (on dit non-inverseuse), une broche - (dite inverseuse) et une broche de sortie ; le principe est que si

$$V > V \iff V_{out} = 1$$

2. Comme mentionné dans la partie précédente, MUX5 n'est accessible que sur la carte Mega ou équivalent.

Le comparateur va donc « comparer » V+ et V- et placer sa broche de sortie dans l'état logique correspondant. Les comparateurs sont un moyen très connu des électroniciens de passer d'un signal analogique à un signal numérique, en le comparant à un certain seuil, on pourrait par exemple imaginer le circuit suivant :

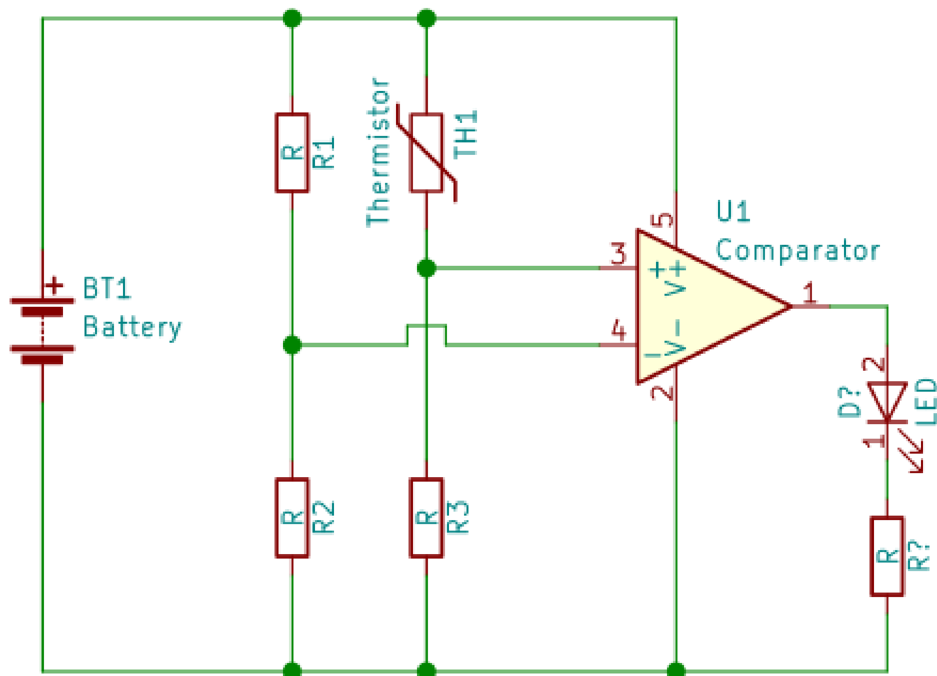

Figure 4. – Exemple d'utilisation d'un comparateur

Ici, le capteur marqué TH est un capteur de température ; il est monté, ainsi que les deux autres résistances de gauche, en pont diviseur – rappelez-vous, nous avons abordé cette notion plus haut, avec la création d'une référence de tension. Vous n'avez pas besoin de comprendre le principe exact de fonctionnement (d'ailleurs, les résistances n'ont même pas de valeurs), mais simplement de voir que lorsque la température augmente, la résistance de la thermistance (le composant qui mesure la température) diminue, ce qui fait augmenter la tension sur V_+, et va au bout d'un moment (quand la tension sera passée au-dessus de V_-), déclencher le comparateur, qui allumera la LED.

Cet exemple est important à appréhender, car des ponts diviseurs, avec les comparateurs, il y en a toujours, et ce même sur l'Arduino.

4.2. Un comparateur inconnu, mais rapide

Alors, me direz-vous, quel est l'intérêt de recourir à un tel comparateur alors que nous pourrions simplement lire la valeur analogique et la convertir en valeur numérique avec un ADC, comme dans la partie précédente. La réponse est simple : le comparateur est rapide, mieux que ça, il est instantané ; une fois configuré, la sortie du comparateur change d'état instantanément

lors du changement décisif de ses entrées (c'est vrai en théorie, et très proche en pratique, car les multiplexeurs, seuls composants pouvant délayer la sortie, ont un délai de propagation très rapide, de l'ordre de la nanoseconde, une fois réglés sur la bonne entrée).

Voyons maintenant comment utiliser ce magnifique comparateur, pour cela, il faut d'abord activer le comparateur, en passant par le registre ACSR, constitué comme suit :

7	6	5	4	3	2	1	0
ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Le premier bit, ACD, permets de désactiver le comparateur lorsqu'il est passé à 1 : il faudra nous assurer qu'il est à 0 avant d'utiliser le comparateur. Aussi, les entrées du comparateur sont déjà utilisées pour des entrées numériques de l'Arduino, il nous faudra donc désactiver ces entrées sur les pins 7 et 6, à travers un registre fait pour ça, DIDR1, que nous passerons intégralement à 0 sauf pour les deux derniers bits. Enfin, le bit d'activation des interruptions (oui, comme dans la partie précédente, avec ADC_vec), doit être passé à 1, il s'agit du bit ACIE du registre ACSR.

Enfin, deux bits nous intéressent particulièrement, et sont amenés à différer selon les usages, les bits ACIS0 et ACIS1, permettant de choisir comment obtenir l'interruption. Pour cela, un petit tableau de l'usage de ces bits :

ACIS1	ACIS0	Mode
0	0	Déclenchement au changement d'état
0	1	/
1	0	Déclenchement au front descendant
1	1	Déclenchement au front montant

Ici, nous utiliserons le déclenchement au front montant, ce qui déclenchera une interruption lorsque de V_+ deviendra supérieur à V_-. Justement, en parlant de V_+ et V_-, où se trouvent-ils sur l'Arduino ? C'est très simple, il s'agit des entrées numériques que nous avons désactivées juste avant, sur les pins 7 (-) et 6 (+).

Bien, écrivons maintenant notre code complet :

```
1 void setup() {
2   // Ne fonctionne pas sur certaines cartes sans ces lignes
3   pinMode(6, INPUT);
4   pinMode(7, INPUT);
5
6   // Désactivons les entrées numérique 6 et 7
7   DIDR1 = 0b11;
8
9   // Activons le comparateur et les interruptions qui lui sont
        liées
```

```
10   cbi(ACSR, ACD);
11   sbi(ACSR, ACIE);
12 
13   // Important : Dire au comparateur d'utiliser AIN0 plutôt que la
         référence interne (défaut)
14   cbi(ACSR, ACBG);
15 
16   // On sélectionne le mode "front montant"
17   ACSR |= 0b11 & 0x03;
18 
19   // Réactivons les interruptions matérielles
20   sei();
21 }
22 
23 // Interruption
24 ISR(ANALOG_COMP_vect) {
25   // Faire quelque chose lors du front montant
26 }
```

C'est terminé, il suffit de placer une référence sur V- et une valeur à mesurer sur V+ pour voir le résultat (n'oubliez pas de faire quelque chose dans l'interruption, et attention aux fonctions Arduino).

4.3. Utiliser une entrée de l'ADC

Dans certains cas, l'utilisation d'un pin numérique pour effectuer une comparaison peut sembler inutile, surtout quand des pins analogiques sont présents pour effectuer ces comparaisons en temps normal. Voyons donc comment utiliser nos chers pins analogiques pour l'entrée négative du comparateur – notons que l'entrée + ne peut quant à elle pas être remplacée, il s'agira **toujours** de AIN0, située broche 6 du Uno.

Commençons donc par activer le multiplexeur de la broche -, en passant le bit ACME du registre ADCSRB à 1 – rappelez-vous, nous avons déjà parlé de ce registre dans la partie sur l'ADC, pour rappel, il ressemble à ça :

7	6	5	4	3	2	1	0
/	ACME	/	/	/	ADTS2	ADTS1	ADTS0

Il vous faudra aussi choisir le pin à utiliser grâce au registre ADMUX, que nous avons lui aussi déjà croisé, je vous invite à vous référer à la partie « Base de l'analogique et référence de tension » pour un tableau descriptif.

Voici donc un petit bout de code qui vous permettra d'utiliser le pin analogique

```
void setup() {
  // Ne fonctionne pas sur certaines cartes sans cette ligne
  pinMode(6, INPUT);

  // Désactivons l'entrée numérique 6
  cbi(DIDR1, AIN0D);

  // Activons le comparateur et les interruptions qui lui sont
      liées
  cbi(ACSR, ACD);
  sbi(ACSR, ACIE);
  cbi(ACSR, ACBG);

  // Activation du multiplexeur analogique et désactivation de
      l'ADC (obligatoire)
  sbi(ADCSRB, ACME);
  cbi(ADCSRA, ADEN);

  // On sélectionne le mode "front montant"
  ACSR |= 0b11 & 0x03;

  // On sélectionne notre pin
  ADMUX |= 0b000 & 0x07;

  // Réactivons les interruptions matérielles
  sei();
}

// Interruption
ISR(ANALOG_COMP_vect) {
  // Faire quelque chose lors du front montant
}
```

Notons que cette méthode est pratique car elle ne gâche pas de pin numérique, et qu'elle permets d'effectuer des conversions depuis des sources différentes rapidement (comparaison sur ADC1, puis sur ADC2, etc) ; toutefois, elle prive l'Arduino de son ADC (il faut le désactiver impérativement), un élément essentiel, assurez-vous donc que vous n'aurez que des comparaisons à faire avec d'effectuer cette procédure.

Comme d'habitude, un petit schéma récapitulatif de cette partie :

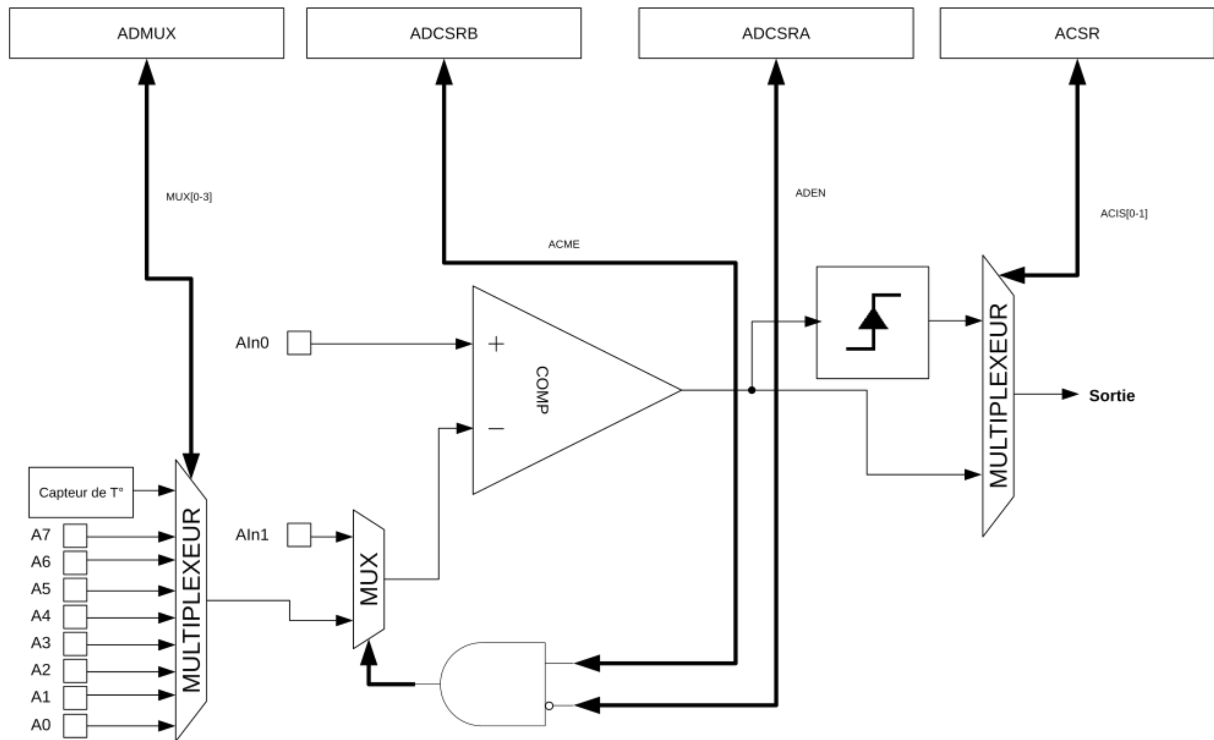

Figure 4. – Schéma comparateur

Encore un schéma qui se passe de commentaires, notez que je me suis permis une petite simplification : l'icône du front montant peut tout autant être le front descendant.

5. Quelques bonus de la datasheet

En lisant la datasheet des processeurs des cartes Uno et Mega pour réaliser ce tutoriel, j'ai noté quelques fonctions intéressantes ou amusantes qu'il me semble utile de mentionner pour faire une petite conclusion, sans ordre particulier.

5.1. La fonction analogRead au complet (ou presque)

Dans les première et deuxième parties de l'article, nous parlions de la fonction `analogRead`, en essayant de faire nos propres lectures analogiques avec les registres AVR. À ce moment, je ne vous avais pas livré le code complet de la fonction, car il ne présentait pas d'intérêt, mais si vous êtes curieux, voici un code fonctionnel et complet pour `analogRead`, un peu plus simple que celui utilisé sur Arduino, car il ne supporte que les trois cartes principales : Uno, Mega et Nano.

```
int analogReadNew(uint8_t pin) {
  // Pour les cartes Mega
  #if defined(__AVR_ATmega1280__)
```

```
    // Si le pin considéré est de type Ax, on utilise ça
    if (pin >= 54) pin -= 54;
  // Pour les Uno et Nano
  #else
    if (pin >= 14) pin -= 14;
  #endif

  // Définition de la référence de tension
  ADMUX |= (1 << REFS0);
  // On sélectionne notre pin
  ADMUX |= pin & 0x07;

  #if defined(ADCSRB) && defined(MUX5)
    // Utilisation de MUX5 sur la Mega pour les pins au-dessus de
        A7
    ADCSRB = (ADCSRB & ~(1 << MUX5)) | (((pin >> 3) & 0x01) <<
        MUX5);
  #endif

  // On lance la conversion
  sbi(ADCSRA, ADSC);

  // Le bit sera désactivé à la fin de la conversion
  while(bit_is_set(ADCSRA, ADSC));

  // Lire ADCL en premier est obligatoire, sinon l'ADC se bloque
  uint8_t low = ADCL;

  // Récupérer le résultat
  return (ADCH << 8) | low;
}
```

Constatons que la lecture du pin est traitée différemment en fonction de la carte, et ce par sélection du processeur. Les pins A0, A1, etc, définis par Arduino, ont été attribués dans la continuité des entrées / sorties numériques. Par exemple, sur un Arduino Mega – disposant de 53 E/S numériques – se cache derrière A0 le nombre 54. C'est pourquoi chaque carte fait l'objet d'une exception dans le code, ce qui est assez lourd, j'en conviens.

5.2. Le capteur de température intégré

Nous allons maintenant utiliser notre fonction `analogRead`, un peu modifiée, afin de découvrir une fonctionnalité intéressante du microcontrôleur : le capteur de température.

5.2.1. Simple d'utilisation...

> Le capteur de température n'est pas disponible sur les cartes Mega et équivalentes, seulement les Uno et diverses.

Ce capteur est *a priori* très simple d'utilisation : il suffit d'aller effectuer une lecture analogique sur le pin à l'adresse 1000 sur la Uno. Cela signifie que vous ne pourrez pas utiliser la fonction analogRead par défaut d'Arduino, puisqu'elle applique un masque 0x07 sur le pin d'entrée ; nous allons donc récupérer notre fonction `analogReadNew`, et en modifier une ligne comme suit :

```
// Ancienne ligne
ADMUX |= pin & 0x07;

// Nouvelle ligne
ADMUX |= pin & 0xF0;
```

Il faudra alors faire très attention à ce que vous faites, puisque quatre bits d'ADMUX pourront être écrits au lieu de trois auparavant ; une solution qui peut être privilégié est la comparaison directe, pour accéder au capteur sans être en mesure de faire une quelconque erreur :

```
if(pin > 8) pin = 0;

ADMUX |= pin & 0xF0;
```

Rien de plus simple maintenant, appelez votre fonction `analogReadNew(0b1000)` pour lire la valeur du capteur de température. Si vous tentez, vous ne trouverez probablement aucun lien concluant entre cette valeur et la température de la pièce dans laquelle vous vous trouvez, ceci car la valeur lue est binaire, comme toujours, or le capteur de température renvoie une tension en millivolts représentant la température en Kelvins, il est donc nécessaire de la convertir en utilisant la formule suivante :

$$
\begin{aligned}
T_{(^\circ C)} &= T_{(K)} - 273 \\
&= \frac{N_{ard} \times U_{alim}}{2^n} - 273 \\
&= \frac{1000 \times N_{ard}(mV) \times U_{alim}}{2^n} - 273 \\
&= \frac{N_{ard} \times 5000}{1024} - 273 \qquad (Uno)
\end{aligned}
$$

5.2.2. ...mais complexe à configurer

Comme vous pouvez le constater, ce capteur est très tentant, et on pourrait vouloir l'user à tout-va, mais comme vous vous en doutez sûrement, il dispose de nombreuses faiblesses. Tout d'abord, la datasheet mentionne lorsqu'elle en parle une précision de ± 10 °C, ce qui est énorme. Mais cette précision n'est même pas respectée en pratique, puisque je relève une valeur de 74, soit 88 °C, dans une pièce à 25 °C...

Nous aurait-on menti ? Le constructeur serait-il enfin démasqué ? Pas du tout, deux facteurs entrent en jeu ici : le capteur étant interne au microcontrôleur, on ne connaît pas sa température exacte, qui est toujours supérieure à la température de la pièce ; ensuite, la datasheet précise qu'il faut **calibrer** le capteur, ce que nous n'avons pas fait. La formule qu'ils proposent est plutôt simple, et part du principe que le capteur est linéaire (ce qui n'est que peu le cas, d'où la tolérance de 10 °C). Il faut donc retirer à la mesure un facteur « intrinsèque » au composant, T_{OS}, puis diviser le tout par un facteur k entier. Cette fois sous forme de code, je vous propose une fonction de conversion :

```
#define T_OS 5
#define K 3

const double ADC_GAIN = 4.8828; // = 5000 / 1024

int convertTemp(int analogVal) {
  // Valeur intermédiaire pour votre compréhension
  unsigned long tempWithoutCorrection = analogVal * ADC_GAIN - 273;

  // Valeur finale
  return (tempWithoutCorrection - T_OS) / K;
}
```

Comme vous pouvez le voir, ce capteur n'est pas des plus précis, mais peut toutefois être utile afin de détecter des températures très hautes ou très basses pouvant endommager le microcontrôleur, ou pour avoir des échelons de température à la dizaine près.

5.3. Un peu d'analogique : L'étage d'entrée de l'ADC

Cette partie aura pour objectif de vérifier le « Analog Input Resistance », mentionné à $100M\Omega$ pour l'AtMega, alors que Microchip (le fabricant) demande de ne pas dépasser $10k\Omega$.

Commençons par voir le schéma équivalent des entrées analogiques de l'AtMega, qui sont reliées telles quelles aux pins Ax de l'Arduino.

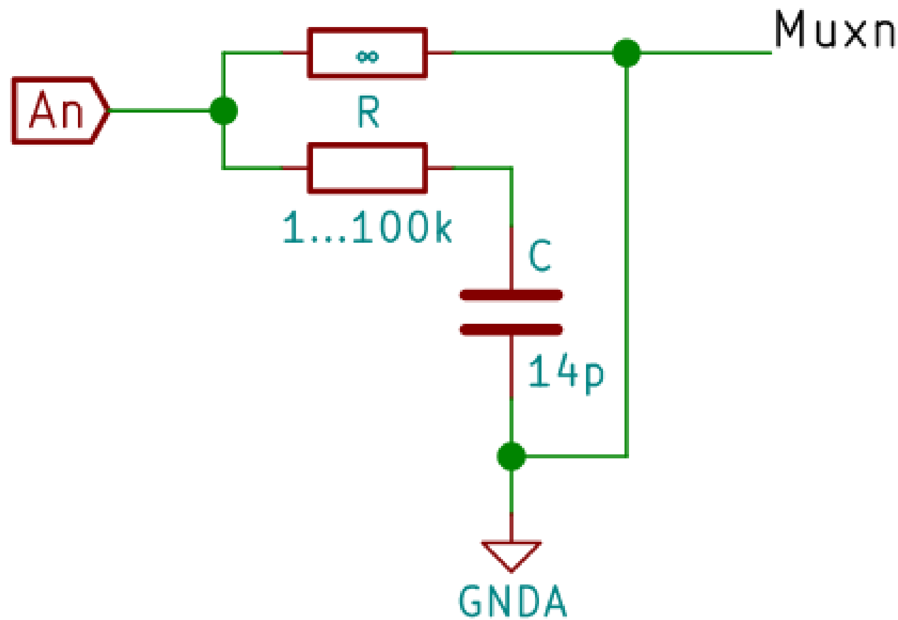

FIGURE 5. – Schéma de l'entrée analogique

La résistance infinie n'est pas présente en réalité, elle est ici pour indiquer qu'on la néglige lors des calculs, et que l'entrée est bien prélevée ici, et non entre le condensateur et la résistance, ce qui aurait formé un filtre passe-bas.

Le circuit d'entrée, relié à An, voit donc en parallèle une résistance infinie et un élément RC, qui va donc dominer, et même être seul à considérer lors des calculs. Une petite précision avant de commencer : la masse analogique GNDA, est à la moitié de la tension d'alimentation, permettant des entrées autant positives que négatives – note : cela ne changera rien à nos calculs. L'impédance en entrée est donc :

$$\begin{aligned} Z_{in} &= R + Z_C \\ &= R - \frac{j}{\omega C} \\ \Leftrightarrow \quad |Z_{in}| &= \sqrt{R^2 + \frac{1}{(\omega C)^2}} \\ |Z_{in}| &= \sqrt{\frac{(R\omega C)^2 + 1}{(\omega C)^2}} \end{aligned}$$

En calculant l'impédance pour $R = 1k\Omega$, $R = 100k\Omega$ et $f = 100kHz$, le condensateur domine de toute façon et l'ensemble présente une impédance d'environ $700k\Omega$. En ayant la règle du dixième en tête, il faudra donc avoir en entrée une impédance de $70k\Omega$. D'où viennent ces valeurs de 10k et 100M, alors ?

La valeur de 10k est une sécurité prise par le constructeur afin d'être sûr de ne pas être à la limite de 700k ; l'atténuation maximale est ainsi restreinte à un peu moins d'un centième de l'entrée, ce qui est très faible. La valeur de 100M, quant à elle, est la valeur réelle de l'impédance

infinie vue précédemment ; elle pourrait être prise en compte dans le calcul, mais, de la même façon, n'influerait que sur un centième du résultat.

Vous voici maintenant maitre de votre Arduino, et prêts à construire votre propre oscilloscope :D. Pour information, ce tutoriel est diffusé sous licence CC-BY, ce qui signifie que vous pouvez le diffuser sous réserve de me mentionner. Les codes qu'il contient sont quant à eux libérés de toute licence, n'hésitez donc pas à les utiliser partout.

L'icône de ce billet a été réalisée par Vect+ et publiée sous licence CC-BY (je respecte ce choix en publiant l'œuvre dérivée sous la même licence), j'y ai ensuite disposé les couleurs de ZdS ainsi que le logo d'Arduino ; elle est volontairement identique au billet précédant cet article.

Liste des abréviations

MET Modèle Équivalent de Thévenin. 8