

Informatique industrielle

Programmation bas niveau de l'ATmega328P

GPIO – Interruptions – Timers – PWM – ADC

A. RICCA, H. HADJ-ABDELKADER & S. Otmane

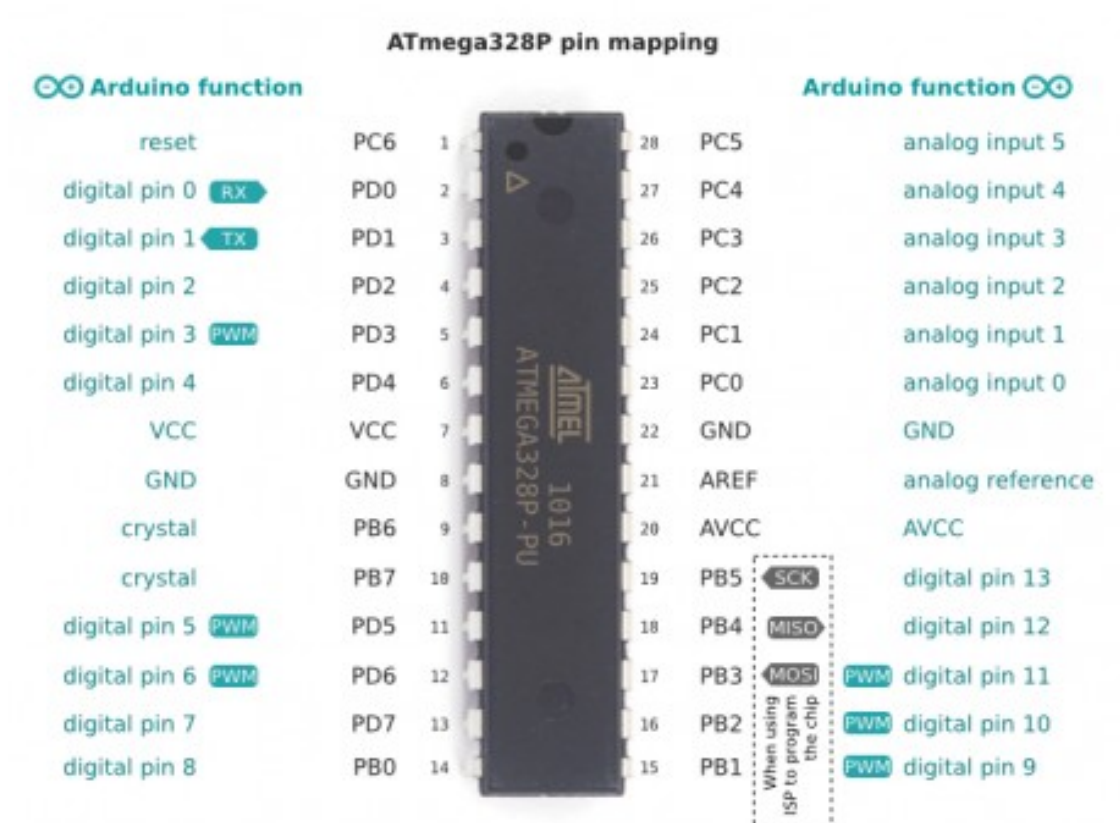


Table des matières

1	Objectifs et organisation du module	4
1.1	Progression proposée	4
2	Architecture de l'ATmega328P et correspondance Arduino Uno	5
2.1	Broche physique, port AVR et numéro Arduino	5
3	Entrées/sorties numériques – GPIO	7
3.1	DDRx : direction	7
3.2	PORTx : sortie ou résistance de pull-up	7
3.2.1	Pourquoi une résistance de pull-up ?	8
3.2.2	Principe de la pull-up	8
3.2.3	Activation de la pull-up interne de l'ATmega328P	9
3.3	PINx : lecture	9
3.4	Masques binaires	9
3.5	Exemple : logique combinatoire	10
4	Interruptions : passer du polling à l'événementiel	11
4.1	Polling ou interruption ?	12
4.2	Interruptions externes INT0 et INT1	12
4.3	Pourquoi utiliser <code>volatile</code> ?	12
4.3.1	Le compilateur ne connaît pas nécessairement les événements asynchrones	13
4.3.2	Le rôle de <code>volatile</code>	14
4.3.3	Exemple : compteur d'impulsions d'un encodeur	14
4.3.4	Quand utiliser <code>volatile</code> ?	15
4.3.5	<code>volatile</code> ne signifie pas "atomique"	15
4.3.6	Lecture atomique d'une variable multi-octets	16
4.4	Interruptions sur changement de broche – PCINT	17
4.4.1	Pourquoi les interruptions PCINT ?	17
4.4.2	INT0/INT1 et PCINT : quelle différence ?	17
4.4.3	Organisation des PCINT de l'ATmega328P	17
4.4.4	Les registres utilisés	18
4.4.5	Méthode générale de configuration	19
4.4.6	Exemple 1 – Bouton sur PB0	19
4.4.7	Attention : PCINT détecte les deux changements	20
4.4.8	Exemple 2 – Ne détecter que l'appui	20
4.4.9	Plusieurs broches dans un même groupe	21
4.4.10	Comment savoir quelle broche a changé ?	21
4.4.11	Pourquoi utiliser un XOR ?	22
4.4.12	Application possible : encodeur incrémental	22
4.4.13	PCINT et rebond des boutons	23
4.4.14	INT0/INT1 ou PCINT ?	23
4.5	Rebonds mécaniques	24

5	Timers : construire une base de temps	25
5.1	Timer0 et Timer1	25
5.2	Mode normal et overflow	25
5.3	CTC : Clear Timer on Compare Match	25
5.4	Génération matérielle d'un signal carré	26
6	PWM : agir sur la puissance	27
6.1	Fast PWM	27
6.2	Timer1 et TOP programmable	27
7	Convertisseur analogique-numérique – ADC	29
7.1	Registres essentiels	29
7.2	Conversion par interruption	30
8	Concevoir un système avec plusieurs interruptions	31
9	Encodeur incrémental	32
9.1	Une voie : compter des impulsions	32
9.2	Deux voies A/B en quadrature	32
9.3	Décodage x1, x2, x4	33
9.4	Décodage simple par interruption	33
10	Mesure de vitesse d'un moteur	34
10.1	Méthode A : compter pendant une fenêtre	34
10.2	Méthode B : mesurer la période entre deux fronts	34
10.3	Input Capture du Timer1	35
10.4	Choix de méthode	35
11	TD – Série d'exercices	36
12	TP – Progression expérimentale	38
12.1	TP1 – Révision GPIO	38
12.2	TP2 – Interruptions externes et événements	38
12.3	TP3 – Timers et base de temps sans delay	38
12.4	TP4 – CTC et PWM	39
12.5	TP5 – ADC avec polling puis interruption	39
12.6	TP6 – Encodeur : position et sens	39
13	Projet intégrateur – Mesure et commande d'un moteur	40
13.1	Cahier des charges minimal	40
13.2	Architecture proposée	40
13.3	Découpage en jalons	41
13.4	Critères de validation	41
14	Corrigés des TD	42
15	Conception robuste d'un système embarqué	44
15.1	Du cahier des charges aux ressources matérielles	44
15.2	Concurrence, atomicité et variables partagées	44
15.3	Architecture non bloquante	45
16	Encodeur incrémental - approfondissement	46
16.1	États de quadrature et décodage x4	46
16.2	Résolution mécanique	46
16.3	Limite de débit	47

17 Mesure de vitesse - précision et filtrage	48
17.1 Méthode par fenêtre : résolution	48
17.2 Méthode par période et quantification du timer	48
17.3 Détection d'arrêt	48
17.4 Filtrage numérique simple	48
18 Projet final - tachymètre et commande de moteur CC	49
18.1 Architecture matérielle	49
18.2 Cahier des charges proposé	49
18.3 Découpage en jalons	50
18.4 Régulation PI - extension	50
19 Série d'exercices supplémentaires	51
19.1 Exercices de synthèse	51
20 Méthode de mise au point et diagnostic	52
20.1 Observer avant de corriger	52
20.2 Checklist de diagnostic	52
21 Fiche de compétences et évaluation	53
22 Fiches registres – aide-mémoire	54
22.1 GPIO	54
22.2 Interruptions externes	54
22.3 Timer1	54
22.4 ADC	54
23 Glossaire des acronymes et termes techniques	55
23.1 Acronymes généraux	55
23.2 Registres et blocs matériels fréquemment utilisés	56
23.3 Termes de programmation embarquée	57
24 Références et conseils de lecture	58

Objectifs et organisation du module

Objectifs

À l'issue du module, l'étudiant doit être capable de lire un datasheet de microcontrôleur, de configurer les périphériques de l'ATmega328P directement par registres et de développer des traitements événementiels reposant sur les interruptions. Ces différentes compétences seront mobilisées dans le cadre d'un projet d'application consacré à la mesure de la vitesse de rotation d'un moteur à l'aide d'un encodeur.

Le fil directeur est volontairement progressif : **entrée/sortie numérique** → **événement** → **temps** → **conversion analogique** → **capteur incrémental** → **mesure de vitesse**. L'API Arduino peut être utilisée pour la communication série et la mise au point, mais la configuration des périphériques est réalisée avec les registres AVR.

À retenir

Convention de lecture. Tout acronyme important est développé à sa première apparition. Un glossaire en fin de document regroupe ensuite les sigles, registres et termes techniques essentiels. AVR désigne ici la famille de microcontrôleurs 8 bits utilisée par l'ATmega328P ; API signifie *Application Programming Interface* (interface de programmation).

1.1 Progression proposée

Séq.	Thème	Compétences	Activité
1	Architecture et GPIO	DDRx, PORTx, PINx, masques, pull-up	TD1 + TP1
2	Interruptions	INT0/INT1, PCINT, ISR, volatile, rebonds	TD2 + TP2
3	Timers	prescaler, overflow, compare match, CTC	TD3 + TP3
4	PWM	Fast PWM, phase correct, fréquence et rapport cyclique	TD4 + TP4
5	ADC	référence, canal, prescaler, conversion, ADIF/ADIE	TD5 + TP5
6	Encodeurs	A/B, quadrature, x1/x2/x4, sens, résolution	TD6 + TP6
7	Vitesse moteur	comptage sur fenêtre, mesure de période, Input Capture	Projet

Architecture de l'ATmega328P et correspondance Arduino Uno

Objectifs

Identifier les ressources du microcontrôleur et comprendre qu'une broche physique peut porter plusieurs fonctions alternatives.

L'ATmega328P est un microcontrôleur AVR 8 bits (famille de microcontrôleurs RISC 8 bits d'Atmel/Microchip). Le CPU (*Central Processing Unit*, unité centrale de traitement) exécute les instructions du programme. Dans le cadre de ce module, les ressources essentielles sont les ports B, C et D, les interruptions externes INT0/INT1, les interruptions sur changement de broche, les Timer/Counter0, 1 et 2, les unités Output Compare, l'unité Input Capture du Timer1 et le convertisseur analogique-numérique 10 bits.

2.1 Broche physique, port AVR et numéro Arduino

La carte Arduino Uno facilite le câblage, mais le programme bas niveau doit raisonner avec les noms AVR. Quelques correspondances fondamentales :

Arduino	ATmega328P	Fonction utile
D13	PB5	LED intégrée / SCK
D9	PB1	OC1A
D8	PB0	ICP1
D6	PD6	OC0A
D3	PD3	INT1 / OC2B
D2	PD2	INT0
A0	PC0	ADC0
A1	PC1	ADC1

ATMega328P and Arduino Uno Pin Mapping

Arduino function					Arduino function
reset	(PCINT14/RESET) PC6	1	28	PC5 (ADC5/SCL/PCINT13)	analog input 5
digital pin 0 (RX)	(PCINT16/RXD) PD0	2	27	PC4 (ADC4/SDA/PCINT12)	analog input 4
digital pin 1 (TX)	(PCINT17/TXD) PD1	3	26	PC3 (ADC3/PCINT11)	analog input 3
digital pin 2	(PCINT18/INT0) PD2	4	25	PC2 (ADC2/PCINT10)	analog input 2
digital pin 3 (PWM)	(PCINT19/OC2B/INT1) PD3	5	24	PC1 (ADC1/PCINT9)	analog input 1
digital pin 4	(PCINT20/XCK/T0) PD4	6	23	PC0 (ADC0/PCINT8)	analog input 0
VCC	VCC	7	22	GND	GND
GND	GND	8	21	AREF	analog reference
crystal	(PCINT6/XTAL1/TOSC1) PB6	9	20	AVCC	VCC
crystal	(PCINT7/XTAL2/TOSC2) PB7	10	19	PB5 (SCK/PCINT5)	digital pin 13
digital pin 5 (PWM)	(PCINT21/OC0B/T1) PD5	11	18	PB4 (MISO/PCINT4)	digital pin 12
digital pin 6 (PWM)	(PCINT22/OC0A/AIN0) PD6	12	17	PB3 (MOSI/OC2A/PCINT3)	digital pin 11(PWM)
digital pin 7	(PCINT23/AIN1) PD7	13	16	PB2 (SS/OC1B/PCINT2)	digital pin 10 (PWM)
digital pin 8	(PCINT0/CLKO/ICP1) PB0	14	15	PB1 (OC1A/PCINT1)	digital pin 9 (PWM)

Digital Pins 11, 12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17, 18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

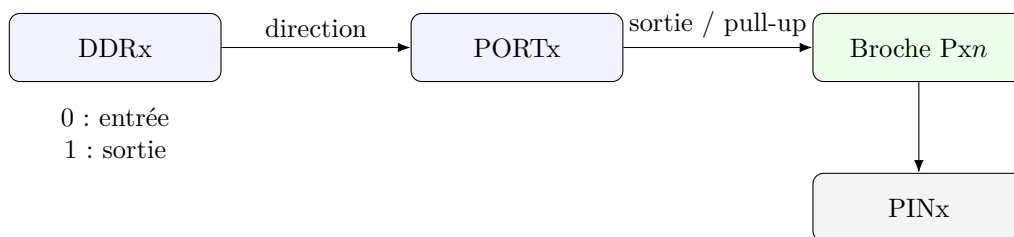
Attention

Toujours commencer par la table de brochage et la description des fonctions alternatives. Une même broche peut être GPIO, entrée d'interruption, sortie PWM ou entrée analogique selon la configuration des registres du périphérique.

Entrées/sorties numériques – GPIO

À retenir

GPIO signifie *General Purpose Input/Output*, soit **entrée/sortie à usage général**. Une GPIO est une broche numérique configurable par logiciel comme entrée ou comme sortie. Sur l'ATmega328P, les GPIO sont principalement regroupées dans les ports B, C et D.



À retenir

Une broche GPIO est pilotée par trois idées différentes : sa **direction** (DDR_x), la **valeur écrite ou la pull-up** (PORT_x), et la **valeur réellement lue** (PIN_x).

Objectifs

Configurer une broche, écrire une sortie, lire une entrée et modifier un bit sans altérer les autres bits d'un registre.

Pour chaque port $x \in \{B, C, D\}$, trois registres structurent le fonctionnement : DDR_x (*Data Direction Register*, registre de direction), PORT_x (registre de données/sortie du port) et PIN_x (registre de lecture des broches du port).

3.1 DDR_x : direction

Un bit à 1 dans DDR_x configure la broche en sortie ; un bit à 0 la configure en entrée.

```
DDRB |= (1 << PB5); // PB5 sortie
DDRB &= ~(1 << PB0); // PB0 entree
```

3.2 PORT_x : sortie ou résistance de pull-up

Le rôle du registre PORT_x dépend de la configuration de la broche définie dans DDR_x.

- si la broche est configurée en **sortie** ($DDxn=1$), le bit $PORTxn$ fixe son niveau logique : 0 pour un niveau bas et 1 pour un niveau haut ;
- si la broche est configurée en **entrée** ($DDxn=0$), le bit $PORTxn$ permet notamment d'activer la **résistance de pull-up interne**.

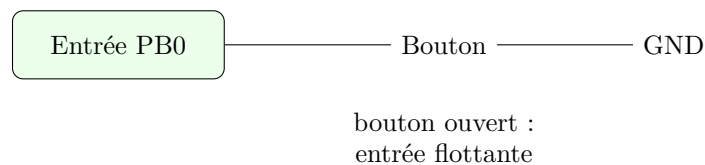
3.2.1 Pourquoi une résistance de pull-up ?

Lorsqu'une broche numérique est configurée en entrée, elle présente une impédance élevée. Si aucun circuit extérieur n'impose son potentiel, son niveau logique n'est donc pas nécessairement défini.

On parle alors d'une **entrée flottante**.

Une entrée flottante peut être sensible aux perturbations électriques environnantes et être interprétée tantôt comme un 0, tantôt comme un 1. Cela peut provoquer des changements d'état indésirables dans le programme.

Prenons par exemple un bouton poussoir connecté directement entre une entrée du microcontrôleur et la masse :



Lorsque le bouton est appuyé, l'entrée est reliée à la masse et son état est clairement égal à 0.

En revanche, lorsque le bouton est relâché, le circuit est ouvert : aucune source n'impose alors directement un niveau logique à l'entrée.

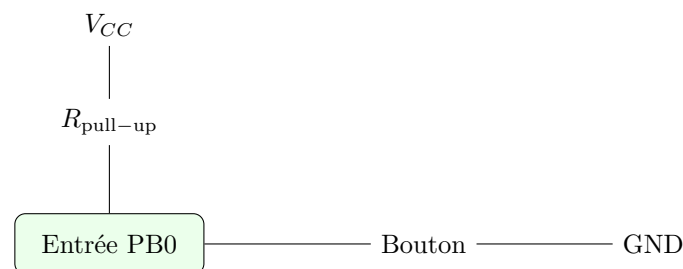
Attention

Une entrée numérique laissée flottante ne doit pas être considérée comme étant automatiquement à 0. Son état peut devenir indéterminé et sensible aux perturbations électriques.

3.2.2 Principe de la pull-up

Une solution consiste à relier l'entrée à l'alimentation V_{CC} par une résistance appelée **résistance de rappel au niveau haut**, ou **pull-up**.

Le principe est alors le suivant :



Lorsque le bouton est **relâché**, la résistance impose un niveau haut sur l'entrée :

$$\text{bouton relâché} \Rightarrow PINB.0 = 1$$

Lorsque le bouton est **appuyé**, l'entrée est directement reliée à la masse :

$$\text{bouton appuyé} \Rightarrow PINB.0 = 0$$

La résistance évite qu'un courant important circule entre V_{CC} et la masse lorsque le bouton est fermé.

3.2.3 Activation de la pull-up interne de l'ATmega328P

L'ATmega328P intègre des résistances de pull-up qui peuvent être activées par logiciel.

Pour une broche configurée en entrée, la combinaison des bits DDxn et PORTxn détermine le comportement de la broche :

DDxn	PORTxn	Configuration
0	0	Entrée sans pull-up (haute impédance)
0	1	Entrée avec pull-up interne activée
1	0	Sortie à l'état bas
1	1	Sortie à l'état haut

```
// PBO configurée en entree
DDRB &= ~(1 << PB0);

// Activation de la resistance de pull-up interne
PORTB |= (1 << PB0);

if (!(PINB & (1 << PB0))) {
    // Le bouton est appuye
}
else {
    // Le bouton est relache
}
```

3.3 PINx : lecture

```
if (PINB & (1 << PB0)) {
    // PBO lu a 1
}
```

À retenir

Avec une résistance de pull-up et un bouton relié à la masse, la logique est inversée par rapport à l'intuition :

bouton relâché → 1

bouton appuyé → 0

On parle de **logique active à l'état bas**.

3.4 Masques binaires

Les opérations |, &, ~, ^ et décalage « doivent être maîtrisées. Le motif (1 << PB5) fabrique un masque dont seul le bit PB5 vaut 1.

À retenir

Préférer `PORTB |= masque` et `PORTB &= ~masque` à une affectation globale lorsque les autres bits du port ne doivent pas être modifiés.

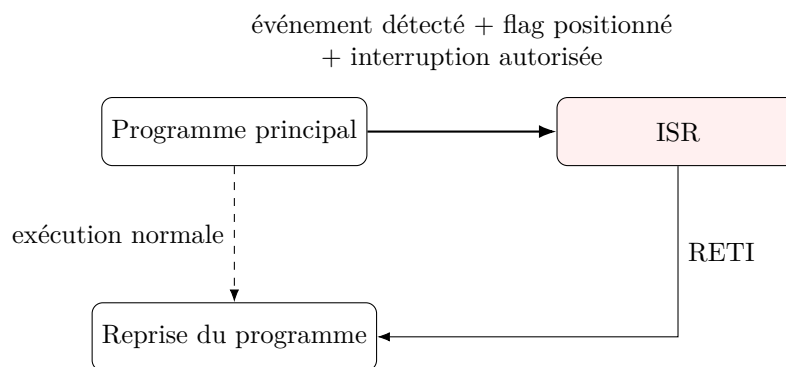
3.5 Exemple : logique combinatoire

Une sirène sur PD5 doit être active seulement si PB0 et PB1 sont actifs :

```
DDRB &= ~(1 << PB0) | (1 << PB1));  
DDRD |= (1 << PD5);  
  
if ((PINB & (1 << PB0)) && (PINB & (1 << PB1)))  
    PORTD |= (1 << PD5);  
else  
    PORTD &= ~(1 << PD5);
```

Cet exemple reprend l'esprit du TP1 fourni et constitue un bon point de départ avant l'introduction des interruptions.

Interruptions : passer du polling à l'événementiel



Attention

Une ISR doit rester courte : acquérir l'information urgente, mettre à jour un compteur ou un drapeau, puis rendre la main. Les calculs longs, affichages et communications sont de préférence traités dans la boucle principale.

Objectifs

Comprendre le mécanisme d'interruption, configurer INT0/INT1 et écrire une ISR courte et sûre.

À retenir

Une **ISR** (*Interrupt Service Routine*) est la routine de service exécutée lorsqu'une interruption est acceptée. **INT0** et **INT1** sont les deux entrées d'interruption externe dédiées de l'AT-mega328P. **PCINT** signifie *Pin Change Interrupt* : interruption déclenchée par un changement d'état d'une broche surveillée.

4.1 Polling ou interruption ?

En polling, le processeur interroge continuellement une entrée. Avec une interruption, le programme principal poursuit son travail et le matériel signale l'événement.

```
Programme principal -----+-----+----->
                           | interruption |
                           v               |
                           [ ISR ]-----+
```

Les vecteurs du datasheet placent notamment INT0, INT1, TIMER1 CAPT, TIMER1 COMPA, TIMER0 COMPA et ADC parmi les sources d'interruptions disponibles.

4.2 Interruptions externes INT0 et INT1

INT0 est associé à PD2, INT1 à PD3. Les bits ISC01:ISC00 et ISC11:ISC10 de EICRA définissent le type de déclenchement ; EIMSK autorise INT0/INT1 et EIFR contient les flags.

ISC01	ISC00	Déclenchement INT0
0	0	niveau bas
0	1	tout changement logique
1	0	front descendant
1	1	front montant

```
#include <avr/interrupt.h>
volatile uint32_t pulses = 0;

void int0_init(void) {
    DDRD &= ~(1 << PD2);
    PORTD |= (1 << PD2); // pull-up si besoin
    EICRA |= (1 << ISC01) | (1 << ISC00); // front montant
    EIMSK |= (1 << INT0);
    sei();
}

ISR(INT0_vect) {
    pulses++;
}
```

4.3 Pourquoi utiliser volatile ?

Lorsqu'une variable est utilisée à la fois dans le programme principal et dans une routine d'interruption (ISR), elle peut être modifiée à un moment que le programme principal ne peut pas prévoir.

Considérons l'exemple suivant :

```
uint8_t bouton_appuye = 0;

ISR(INT0_vect)
{
    bouton_appuye = 1;
}

int main(void)
{
    // Initialisation de INT0...
```

```

    sei();

    while (bouton_appuye == 0)
    {
        // Attente de l'appui sur le bouton
    }

    // Le bouton a ete appuye
    PORTB |= (1 << PB5);
}

```

À première vue, le fonctionnement semble évident :

1. le programme principal attend que `bouton_appuye` devienne égal à 1 ;
2. une interruption survient ;
3. l'ISR place `bouton_appuye` à 1 ;
4. le programme principal quitte la boucle.

Pourtant, ce programme peut ne pas fonctionner correctement lorsque les optimisations du compilateur sont activées.

4.3.1 Le compilateur ne connaît pas nécessairement les événements asynchrones

Le compilateur cherche à produire un programme aussi rapide que possible.

Lorsqu'il analyse :

```

while (bouton_appuye == 0)
{
}

```

il peut constater qu'aucune instruction visible dans cette boucle ne modifie la variable `bouton_appuye`.

Pour éviter de relire inutilement la mémoire à chaque tour de boucle, il peut conserver la valeur de cette variable dans un registre du processeur.

Le raisonnement peut alors devenir conceptuellement :

```

// Representation simplifiee de ce que pourrait faire le compilateur

registre = bouton_appuye;

while (registre == 0)
{
}

```

Le problème est que l'ISR peut modifier la variable en mémoire :

```

ISR(INT0_vect)
{
    bouton_appuye = 1;
}

```

mais le programme principal peut continuer à utiliser l'ancienne valeur conservée dans son registre. Il risque alors de ne jamais constater le changement.

Attention

Une interruption est un événement **asynchrone** : elle peut modifier une variable sans que cette modification apparaisse dans le flux normal des instructions du programme principal.

4.3.2 Le rôle de volatile

Pour indiquer au compilateur qu'une variable peut être modifiée de manière imprévisible pour le programme courant, on utilise le mot-clé `volatile` :

```
volatile uint8_t bouton_appuye = 0;
```

Le mot-clé `volatile` signifie essentiellement :

« La valeur de cette variable peut changer en dehors du flux normal du programme. Ne suppose pas que sa valeur est restée inchangée : relis-la lorsque le programme la demande. »

Le programme devient donc :

```
volatile uint8_t bouton_appuye = 0;

ISR(INT0_vect)
{
    bouton_appuye = 1;
}

int main(void)
{
    sei();

    while (bouton_appuye == 0)
    {
        // bouton_appuye est relu
    }

    PORTB |= (1 << PB5);
}
```

Lorsque l'ISR positionne la variable à 1, le programme principal peut alors observer cette nouvelle valeur.

4.3.3 Exemple : compteur d'impulsions d'un encodeur

Le cas est particulièrement important avec un encodeur.

À chaque front détecté, une interruption incrémente un compteur :

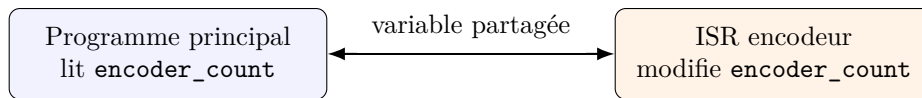
```
volatile uint32_t encoder_count = 0;

ISR(INT0_vect)
{
    encoder_count++;
}
```

Pendant ce temps, le programme principal utilise ce compteur pour calculer la position ou la vitesse :

```
while (1)
{
    if (flag_100ms)
    {
        // Utilisation du nombre d'impulsions
        calcul_vitesse(encoder_count);
    }
}
```

La variable `encoder_count` est donc partagée entre deux contextes :



Dans cette situation, la déclaration :

```
volatile uint32_t encoder_count;
```

est nécessaire pour indiquer au compilateur que la valeur peut être modifiée par l'ISR.

4.3.4 Quand utiliser volatile ?

Dans ce module, on utilisera notamment `volatile` pour les variables partagées entre une ISR et le programme principal.

Exemples :

```
volatile uint8_t flag_100ms;
volatile uint32_t encoder_count;
volatile uint16_t adc_value;
volatile uint16_t period_ticks;
```

Ces variables peuvent être modifiées dans une ISR et utilisées dans le programme principal.

À retenir

Une variable partagée entre une ISR et le programme principal doit généralement être déclarée `volatile`.

Ce que fait concrètement `volatile` :

- Force une **lecture en RAM** à chaque fois que la variable est appelée.
- Force une **écriture en RAM** dès que la variable est modifiée.
- Interdit au compilateur de supprimer des lignes de code qui lui semblent inutiles (comme des boucles de temporisation vides).

4.3.5 volatile ne signifie pas “atomique”

Il faut distinguer deux problèmes différents :

1. **visibilité de la variable** : le compilateur doit tenir compte du fait que sa valeur peut changer ; c'est le rôle de `volatile` ;
2. **atomicité de l'accès** : il faut éviter qu'une interruption modifie la variable pendant que le programme principal est en train de la lire.

Le mot-clé `volatile` résout le premier problème, **mais pas le second**.

L'ATmega328P est un microcontrôleur 8 bits. Il peut donc naturellement manipuler des données de 8 bits en une opération élémentaire.

Considérons maintenant :

```
volatile uint32_t encoder_count;
```

Cette variable occupe 32 bits, soit quatre octets :

B_3	B_2	B_1	B_0
-------	-------	-------	-------

Sa lecture nécessite donc plusieurs opérations.

Supposons que :

```
encoder_count = 0x0000FFFF
```

et que le programme principal commence à lire cette variable.

Une interruption encodeur survient alors et effectue :

```
encoder_count++
```

La nouvelle valeur devient :

```
0x00010000.
```

Si l'interruption survient entre deux lectures d'octets, le programme principal peut théoriquement construire une valeur composée d'une partie de l'ancienne valeur et d'une partie de la nouvelle.

La valeur lue est alors incohérente.

4.3.6 Lecture atomique d'une variable multi-octets

Pour effectuer une copie cohérente du compteur, on peut protéger temporairement la lecture :

```
#include <util/atomic.h>

volatile uint32_t encoder_count = 0;

uint32_t snapshot;

ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
{
    snapshot = encoder_count;
}
```

La variable partagée n'est protégée que pendant le temps nécessaire à sa copie.
Les calculs plus longs sont ensuite effectués hors de la section atomique :

```
ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
{
    snapshot = encoder_count;
}

// Interruptions de nouveau autorisées
rpm = calcul_vitesse(snapshot);
```

À retenir

volatile et atomicité répondent à deux problèmes différents.

Problème	Solution
Le compilateur pourrait conserver une ancienne valeur	Déclarer la variable volatile
Une ISR peut interrompre la lecture d'une variable multi-octets	Protéger la lecture par une section atomique

Ainsi :

volatile ≠ atomique

4.4 Interruptions sur changement de broche – PCINT

4.4.1 Pourquoi les interruptions PCINT ?

Nous avons vu précédemment les interruptions externes `INT0` et `INT1`. Elles sont très pratiques, mais l'ATmega328P ne possède que deux entrées de ce type :

$$INT0 \longrightarrow PD2 \qquad INT1 \longrightarrow PD3.$$

Que faire si l'on souhaite détecter par interruption un changement d'état sur une autre broche, par exemple `PB0`, `PB4` ou `PC2` ?

L'ATmega328P propose pour cela les interruptions **PCINT** (*Pin Change Interrupt*).

À retenir

Une interruption PCINT permet de déclencher une interruption lorsqu'une broche numérique change d'état :

$$0 \rightarrow 1 \quad \text{ou} \quad 1 \rightarrow 0.$$

Contrairement à `INT0` et `INT1`, les PCINT sont disponibles sur un grand nombre de broches du microcontrôleur.

4.4.2 INT0/INT1 et PCINT : quelle différence ?

Les deux mécanismes ne fonctionnent pas exactement de la même manière.

	INT0 / INT1	PCINT
Broches	PD2 et PD3	Nombreuses broches des ports B, C et D
Type d'événement	Configurable : front montant, front descendant, changement, niveau bas	Changement d'état de la broche
Vecteur	Un vecteur spécifique pour <code>INT0</code> et un pour <code>INT1</code>	Un vecteur commun à un groupe de broches
Identification de la source	Immédiate	À déterminer dans l'ISR si plusieurs broches sont activées

Ainsi, `INT0` et `INT1` sont particulièrement adaptés lorsqu'on souhaite surveiller précisément une ou deux entrées importantes.

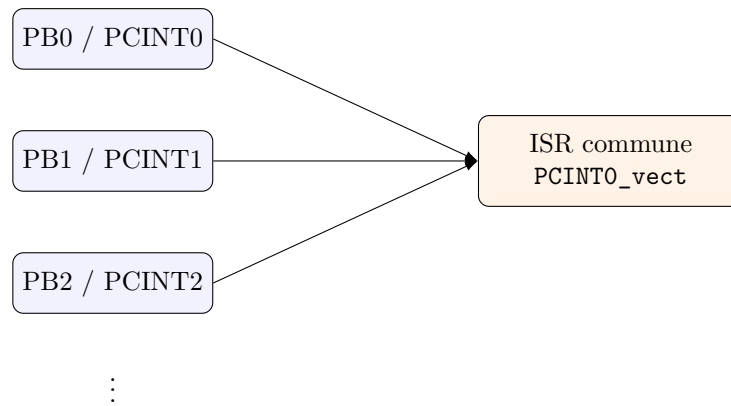
Les PCINT sont intéressantes lorsqu'il faut surveiller davantage d'entrées.

4.4.3 Organisation des PCINT de l'ATmega328P

Les broches PCINT sont regroupées en trois groupes.

Groupe	Port	PCINT	Vecteur ISR
PCIE0	PORTB	PCINT0 à PCINT7	PCINT0_vect
PCIE1	PORTC	PCINT8 à PCINT14	PCINT1_vect
PCIE2	PORTD	PCINT16 à PCINT23	PCINT2_vect

Par exemple, les différentes broches du `PORTB` appartiennent au même groupe d'interruptions.



Il faut donc bien distinguer :

- le **numéro PCINT**, qui identifie une broche ;
- le **groupe PCINT**, qui possède un vecteur d'interruption commun.

4.4.4 Les registres utilisés

La configuration d'une interruption PCINT fait principalement intervenir trois types de registres.

Le registre PCICR

Le registre PCICR (*Pin Change Interrupt Control Register*) permet d'activer un groupe d'interruptions PCINT.

Les bits utiles sont :

PCIE0, PCIE1, PCIE2.

Par exemple :

```
// Autoriser les interruptions PCINT du PORTB
PCICR |= (1 << PCIE0);
```

Cela autorise le groupe PCINT associé au PORTB.

Attention

Cette instruction ne suffit pas à sélectionner une broche particulière. Elle autorise seulement le **groupe** d'interruptions. Il faut ensuite indiquer quelles broches du groupe doivent être surveillées.

Les registres PCMSKx

Les registres PCMSKx (*Pin Change Mask Register*) permettent de sélectionner individuellement les broches devant provoquer une interruption.

On dispose de :

PCMSK0, PCMSK1, PCMSK2.

Ils correspondent respectivement aux trois groupes PCINT.

Pour le PORTB, on utilise par exemple PCMSK0.

Pour activer PCINT0 :

```
PCMSK0 |= (1 << PCINT0);
```

Pour activer simultanément PCINT0 et PCINT4 :

```
PCMSK0 |= (1 << PCINT0) | (1 << PCINT4);
```

Le registre PCIFR

Le registre PCIFR (*Pin Change Interrupt Flag Register*) contient les indicateurs d'interruption des différents groupes.

Lorsqu'un changement est détecté sur une broche autorisée, le flag correspondant au groupe est positionné.

Si les interruptions sont globalement autorisées, le processeur exécute alors l'ISR correspondante.

4.4.5 Méthode générale de configuration

Pour utiliser une interruption PCINT, on peut retenir la démarche suivante :

1. Configurer la broche en entrée avec DDRx.
2. Activer éventuellement sa résistance de pull-up interne avec PORTx.
3. Sélectionner la broche dans le registre PCMSKx.
4. Autoriser le groupe correspondant dans PCICR.
5. Autoriser globalement les interruptions avec `sei()`.
6. Écrire l'ISR correspondant au groupe.

On peut résumer la configuration par :

GPIO → PCMSKx → PCICR → `sei()` → ISR

4.4.6 Exemple 1 – Bouton sur PB0

Considérons un bouton connecté entre PB0 et la masse.

Nous souhaitons déclencher une interruption à chaque changement d'état du bouton.

La résistance de pull-up interne est utilisée :

bouton relâché → $PB0 = 1$

bouton appuyé → $PB0 = 0$.

Le programme peut être écrit ainsi :

```
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint8_t evenement_bouton = 0;

ISR(PCINT0_vect)
{
    evenement_bouton = 1;
}

int main(void)
{
    // PB0 en entree
    DDRB &= ~(1 << PB0);

    // Pull-up interne
    PORTB |= (1 << PB0);

    // Activation de PCINT0
    PCMSK0 |= (1 << PCINT0);

    // Activation du groupe PCINT du PORTB
```

```

    PCICR |= (1 << PCIE0);

    // Autorisation globale des interruptions
    sei();

    while (1)
    {
        if (evenement_bouton)
        {
            evenement_bouton = 0;

            // Traitement du changement d'etat
        }
    }
}

```

À chaque changement de PB0 :

$$1 \rightarrow 0 \quad \text{ou} \quad 0 \rightarrow 1,$$

l'ISR PCINT0_vect est exécutée.

4.4.7 Attention : PCINT détecte les deux changements

Avec INT0 ou INT1, nous pouvons choisir par exemple de ne détecter qu'un front montant.

Avec une PCINT, l'interruption est associée au **changement d'état**.

Pour notre bouton :

appui	$1 \rightarrow 0$	$\Rightarrow$	interruption
-------	-------------------	---------------	--------------

mais également :

relâchement	$0 \rightarrow 1$	$\Rightarrow$	interruption
-------------	-------------------	---------------	--------------

Si nous souhaitons effectuer une action uniquement lors de l'appui, il faut donc tester l'état de la broche dans l'ISR ou dans le programme principal.

4.4.8 Exemple 2 – Ne détecter que l'appui

On peut tester PB0 dans l'ISR :

```

volatile uint8_t bouton_appuye = 0;

ISR(PCINT0_vect)
{
    if (!(PINB & (1 << PB0)))
    {
        // PB0 = 0 : bouton appuye
        bouton_appuye = 1;
    }
}

```

L'ISR est toujours appelée à l'appui et au relâchement, mais l'événement n'est enregistré que lorsque :

$$PB0 = 0.$$

4.4.9 Plusieurs broches dans un même groupe

Une caractéristique importante des PCINT apparaît lorsque plusieurs broches du même groupe sont activées.

Supposons que nous souhaitions surveiller PB0 et PB4 :

```
PCMSK0 |= (1 << PCINT0) | (1 << PCINT4);
PCICR |= (1 << PCIE0);
```

Un changement sur PB0 provoque :

```
ISR(PCINT0_vect)
{
    ...
}
```

mais un changement sur PB4 provoque exactement la même ISR :

```
ISR(PCINT0_vect)
{
    ...
}
```

Le microcontrôleur indique donc :

“une broche du groupe a changé”

mais il ne fournit pas directement à l'ISR le numéro de la broche responsable.
Il faut donc identifier cette broche par logiciel.

4.4.10 Comment savoir quelle broche a changé ?

Une méthode consiste à mémoriser l'état précédent du port.
Par exemple :

```
volatile uint8_t ancien_PINB;

int main(void)
{
    DDRB &= ~(1 << PB0) | (1 << PB4);

    PORTB |= (1 << PB0) | (1 << PB4);

    // Memorisation de l'etat initial
    ancien_PINB = PINB;

    PCMSK0 |= (1 << PCINT0) | (1 << PCINT4);
    PCICR |= (1 << PCIE0);

    sei();

    while (1)
    {
        // Programme principal
    }
}
```

Dans l'ISR, on lit le nouvel état :

```
ISR(PCINT0_vect)
{
    uint8_t nouveau_PINB;
```

```

uint8_t changement;

nouveau_PINB = PINB;

changement = nouveau_PINB ^ ancien_PINB;

ancien_PINB = nouveau_PINB;

// Test de PB0
if (changement & (1 << PB0))
{
    // PB0 a change
}

// Test de PB4
if (changement & (1 << PB4))
{
    // PB4 a change
}
}

```

4.4.11 Pourquoi utiliser un XOR ?

L'opération XOR (OU exclusif) est particulièrement utile pour détecter un changement. Sa table de vérité est :

Ancien état	Nouvel état	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Ainsi :

$$\text{ancien état} \oplus \text{nouvel état} = 1 \iff \text{la broche a changé}$$

Par exemple :

$$\text{ancien_PINB} = 00100001$$

et après un événement :

$$\text{nouveau_PINB} = 00110000.$$

L'opération :

$$\text{ancien_PINB} \oplus \text{nouveau_PINB}$$

fait apparaître à 1 uniquement les bits ayant changé.

Cette technique permet donc de déterminer efficacement la ou les broches responsables de l'interruption.

4.4.12 Application possible : encodeur incrémental

Les interruptions PCINT peuvent également être intéressantes pour l'acquisition d'un encodeur incrémental.

Un encodeur en quadrature fournit deux signaux :

A et B .

Si les deux voies sont placées sur des broches PCINT, les changements des deux signaux peuvent provoquer des interruptions.

On peut alors :

1. lire simultanément les états de A et B ;
2. comparer le nouvel état avec l'état précédent ;
3. déterminer le sens de rotation ;
4. incrémenter ou décrémenter un compteur de position.

Cette approche devient particulièrement intéressante pour un décodage en quadrature $\times 4$, puisque les fronts montants et descendants des deux voies peuvent être exploités.

Elle sera reprise plus loin dans l'application de mesure de vitesse du moteur.

4.4.13 PCINT et rebond des boutons

L'utilisation d'un bouton mécanique avec une PCINT met en évidence un autre problème : le **rebond**.

Lors d'un appui, le contact électrique ne passe pas toujours directement de 1 à 0.

On peut observer pendant quelques millisecondes :

$$1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0.$$

Une PCINT peut donc générer plusieurs interruptions pour un seul appui physique.

Attention

Une interruption PCINT ne réalise aucun anti-rebond matériel ou logiciel.

Pour un bouton mécanique, il faut prévoir une stratégie d'anti-rebond si chaque appui doit correspondre à un événement unique.

4.4.14 INT0/INT1 ou PCINT ?

Le choix dépend de l'application.

INT0 / INT1	PCINT
Seulement deux entrées	Beaucoup plus de broches disponibles
Choix précis du type de déclenchement	Détection d'un changement d'état
ISR spécifique à l'entrée	ISR commune à plusieurs broches
Identification simple de la source	Identification logicielle parfois nécessaire
Très pratique pour un signal critique	Très pratique pour surveiller plusieurs entrées

À retenir

Pour configurer une PCINT, retenir les trois niveaux suivants :

PCMSKx : quelle broche ?

PCICR : quel groupe ?

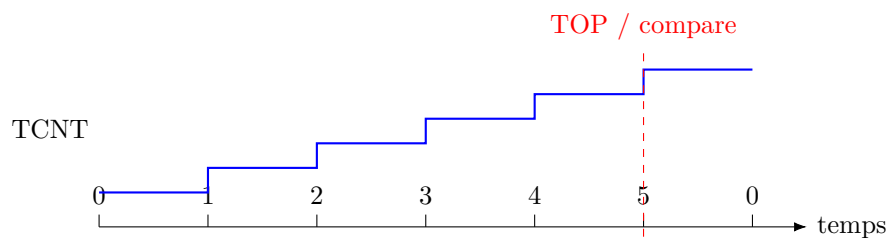
PCINTx_vect : *quelleISR?*

Si plusieurs broches d'un même groupe sont autorisées, elles partagent la même ISR. Le programme doit alors déterminer quelle broche a changé.

4.5 Rebonds mécaniques

Un bouton peut générer plusieurs fronts en quelques millisecondes. Un anti-rebond peut être matériel (RC/Schmitt) ou logiciel. Éviter un long `delay()` dans l'ISR ; mieux vaut mémoriser l'événement puis valider l'état après un délai géré par timer.

Timers : construire une base de temps



Le timer transforme l'horloge CPU en une grandeur mesurable. Avec un prescaler N , un incrément du compteur dure $T_{tick} = N/f_{CPU}$. En CTC, le compare match définit directement la période logicielle ou matérielle.

Objectifs

Relier fréquence d'horloge, prescaler, compteur, compare match et fréquence d'interruption.

L'Arduino Uno utilise typiquement une horloge CPU de 16 MHz. Après division par un prescaler N , la durée d'un tick vaut

$$T_{tick} = \frac{N}{f_{CPU}}.$$

Exemple : $f_{CPU} = 16\text{ MHz}$ et $N = 64$ donnent $T_{tick} = 4\text{ }\mu\text{s}$.

5.1 Timer0 et Timer1

Timer0 est 8 bits ($0 \dots 255$), Timer1 est 16 bits ($0 \dots 65535$). Timer1 est particulièrement intéressant pour la mesure car il offre une plage plus grande et une unité Input Capture.

5.2 Mode normal et overflow

En mode normal, le compteur s'incrémente jusqu'à sa valeur maximale puis repasse à zéro. Le flag d'overflow peut générer une interruption.

5.3 CTC : Clear Timer on Compare Match

CTC signifie *Clear Timer on Compare Match* : le compteur est remis à zéro lorsqu'il atteint la valeur de comparaison choisie. Le registre **OCRnx** (*Output Compare Register*) contient une valeur de comparaison, tandis que **TCNTn** (*Timer/Counter Register*) contient la valeur courante du compteur. En CTC, le timer repart à zéro lorsqu'il atteint TOP. Pour une interruption périodique avec OCR1A comme TOP :

$$f_{IT} = \frac{f_{CPU}}{N(1 + OCR1A)}.$$

Pour obtenir 1 kHz avec Timer1, $f_{CPU} = 16$ MHz et $N = 64$:

$$OCR1A = \frac{16 \times 10^6}{64 \times 1000} - 1 = 249.$$

```
volatile uint32_t ms = 0;

void timer1_1ms_init(void) {
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1 = 0;
    OCR1A = 249;
    TCCR1B |= (1 << WGM12); // CTC
    TCCR1B |= (1 << CS11) | (1 << CS10); // /64
    TIMSK1 |= (1 << OCIE1A); // interruption Compare A
}

ISR(TIMER1_COMPA_vect) { ms++; }
```

5.4 Génération matérielle d'un signal carré

En CTC avec basculement automatique de OC0A à chaque compare match :

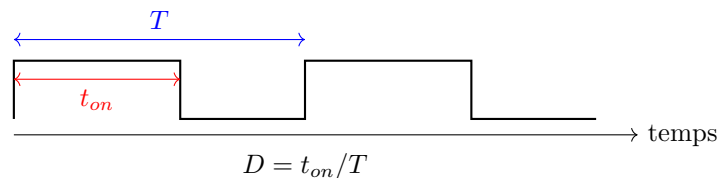
$$f_{OC0A} = \frac{f_{CPU}}{2N(1 + OCR0A)}.$$

C'est précisément le principe utilisé dans le TP2 fourni pour 40 kHz, 10 kHz et 1 kHz.

PWM : agir sur la puissance

À retenir

PWM signifie *Pulse Width Modulation*, en français **modulation de largeur d'impulsion**. La fréquence fixe la rapidité de répétition du signal et le rapport cyclique fixe la fraction de période passée à l'état actif.



La fréquence PWM et le rapport cyclique sont deux paramètres distincts : la première est imposée par le mode, le prescaler et TOP ; le second est principalement fixé par OCRnx.

Objectifs

Configurer une sortie PWM et calculer fréquence et rapport cyclique.

Une PWM alterne entre 0 et 1 avec une période T et un rapport cyclique $D = t_{on}/T$. Pour une charge lente devant la période PWM, la valeur moyenne est approximativement proportionnelle à D .

6.1 Fast PWM

Pour Timer0 en Fast PWM 8 bits avec TOP=255 :

$$f_{PWM} = \frac{f_{CPU}}{N \cdot 256}.$$

En mode non-inversé, le rapport cyclique est approximativement

$$D \approx \frac{OCR0A}{255}.$$

6.2 Timer1 et TOP programmable

Le Timer1 permet des modes où TOP est fixé par ICR1. En Fast PWM mode 14 :

$$f_{PWM} = \frac{f_{CPU}}{N(1 + ICR1)}.$$

Puis OCR1A fixe le rapport cyclique sur OC1A.

```
// Exemple : PWM Timer1 mode 14, sortie OC1A (PB1)
DDRB |= (1 << PB1);
TCCR1A = (1 << COM1A1) | (1 << WGM11);
TCCR1B = (1 << WGM13) | (1 << WGM12) | (1 << CS11); // /8
ICR1 = 3999; // fPWM = 16MHz/(8*4000)=500 Hz
OCR1A = 1200; // environ 30 %
```

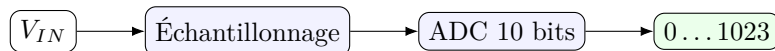
Attention

Pour piloter un moteur, une broche du microcontrôleur ne doit pas alimenter directement le moteur. Utiliser un étage de puissance (MOSFET, pont en H ou driver) et une protection adaptée à la charge inductive.

Convertisseur analogique-numérique – ADC

À retenir

ADC signifie *Analog-to-Digital Converter*, soit **convertisseur analogique-numérique (CAN)**. Il transforme une tension analogique en un nombre numérique. Sur l'ATmega328P, le résultat est codé sur 10 bits, soit 1024 niveaux possibles.



V_{REF} fixe la pleine échelle

À retenir

Le choix du prescaler ADC n'est pas seulement un calcul de fréquence : il conditionne le compromis entre vitesse de conversion et respect des conditions recommandées par le constructeur.

Objectifs

Configurer une conversion 10 bits, choisir la référence et le canal, puis exploiter l'interruption de fin de conversion.

L'ADC fournit un résultat 10 bits, donc 1024 niveaux. Pour une référence V_{REF} :

$$ADC \approx \frac{V_{IN}}{V_{REF}} \times 1023, \quad V_{IN} \approx \frac{ADC}{1023} V_{REF}.$$

Avec $V_{REF} = 5$ V, un LSB (*Least Significant Bit*, bit de poids faible ; ici utilisé pour désigner le pas de quantification) vaut environ 4,89 mV.

7.1 Registres essentiels

- ADMUX : référence de tension et sélection du canal ;
- ADCSRA : activation ADC, lancement, auto-trigger, flag, interruption, prescaler ;
- ADCSRB : source d'auto-trigger et fonctions complémentaires ;
- ADC (ADCL/ADCH) : résultat de conversion.

```
void adc_init(void) {
    ADMUX = (1 << REFS0); // AVcc comme reference, canal ADC0
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // /128
}

uint16_t adc_read(void) {
    ADCSRA |= (1 << ADSC);
    while (ADCSRA & (1 << ADSC)) { }
    return ADC;
}
```

7.2 Conversion par interruption

```
volatile uint16_t adc_value;

ISR(ADC_vect) {
    adc_value = ADC;
}

// Initialisation : ajouter ADIE puis sei()
ADCSRA |= (1 << ADIE);
```

L'intérêt est de déclencher l'acquisition à une cadence connue sans bloquer le programme principal. Une architecture fréquente est : Timer → déclenchement ADC → ISR ADC → mise à jour de la mesure.

Concevoir un système avec plusieurs interruptions

Objectifs

Structurer un programme embarqué non bloquant.

Une bonne architecture sépare les événements rapides et les traitements longs.

- ISR encodeur : incrémenter/décroquer un compteur.
- ISR Timer : produire le top d'échantillonnage.
- ISR ADC : mémoriser la conversion.
- Boucle principale : calculs, affichage, communication, diagnostic.

À retenir

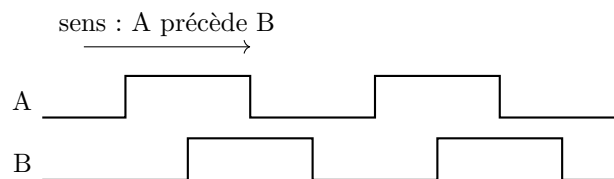
Une ISR doit être courte : acquitter implicitement l'événement, capturer l'information minimale, mettre à jour un compteur/flag, puis rendre la main. Éviter les impressions série et les calculs flottants dans une ISR.

```
volatile uint8_t flag_100ms = 0;
volatile uint32_t encoder_count = 0;

ISR(INT0_vect) { encoder_count++; }
ISR(TIMER1_COMPA_vect) { flag_100ms = 1; }

int main(void) {
    // init...
    sei();
    while (1) {
        if (flag_100ms) {
            flag_100ms = 0;
            // copie atomique, calcul vitesse, communication...
        }
    }
}
```

Encodeur incrémental



Les deux voies sont déphasées d'environ un quart de période électrique. L'ordre des transitions fournit le sens, tandis que le nombre de transitions retenues fixe la résolution de décodage.

Décodage	Événements comptés	Comptes/tour pour N périodes
x1	un front de A	N
x2	deux fronts de A	$2N$
x4	fronts de A et B	$4N$

Objectifs

Comprendre le principe A/B en quadrature, déterminer le sens et convertir des impulsions en angle.

9.1 Une voie : compter des impulsions

Si un encodeur fournit N impulsions par tour et que n impulsions sont comptées :

$$\theta = 2\pi \frac{n}{N}, \quad \theta_{deg} = 360 \frac{n}{N}.$$

Une seule voie donne le déplacement relatif mais pas le sens de rotation sans information supplémentaire.

9.2 Deux voies A/B en quadrature

Les voies A et B sont déphasées d'environ 90 degrés électriques. Selon le sens, A précède B ou B précède A.

Sens + :

A ----
B ----

Sens - : ordre des transitions inverse

9.3 Décodage x1, x2, x4

On peut compter un seul front de A (x1), les deux fronts de A (x2), ou toutes les transitions A/B (x4). Si le constructeur annonce P périodes par tour sur une voie, le nombre de comptes par tour dépend du mode de décodage.

Attention

PPR signifie généralement *Pulses Per Revolution* (impulsions par tour) et **CPR** *Counts Per Revolution* (comptes par tour). Leur usage n'est toutefois pas uniforme selon les fabricants. Toujours préciser l'unité fournie par le constructeur : PPR, CPR, lignes/tour ou comptes/tour ne sont pas toujours employés de manière uniforme. Le logiciel doit utiliser le nombre réel de comptes par révolution après décodage.

9.4 Décodage simple par interruption

Sur un front montant de A, lire B :

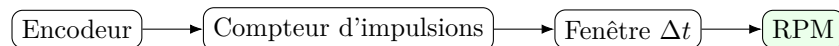
```
ISR(INT0_vect) {  
    if (PIND & (1 << PD3))  
        encoder_count--;  
    else  
        encoder_count++;  
}
```

C'est un décodage x1 très pédagogique. Un décodage x4 robuste peut utiliser une table d'états construite avec l'ancien état AB et le nouvel état AB.

Mesure de vitesse d'un moteur

À retenir

RPM signifie *Revolutions Per Minute*, c'est-à-dire **tours par minute (tr/min)**. Dans les équations suivantes, N_c désigne le nombre réel de comptes d'encodeur par tour après prise en compte du mode de décodage.



	Comptage sur fenêtre	Mesure de période
Principe	Compter n événements pendant Δt	Mesurer le temps entre deux événements
Faible vitesse	résolution souvent médiocre si fenêtre courte	très bonne finesse, mais attente plus longue
Grande vitesse	très efficace	exige une base de temps et une capture suffisamment rapides
Arrêt moteur	naturellement $n = 0$	nécessite un timeout

Objectifs

Comparer la mesure par comptage sur fenêtre temporelle et la mesure de période.

10.1 Méthode A : compter pendant une fenêtre

Si n comptes sont observés pendant Δt et si l'encodeur fournit N_c comptes par tour :

$$RPM = \frac{60n}{N_c \Delta t}.$$

Exemple : $N_c = 500$, $n = 42$ pendant 0,1 s :

$$RPM = \frac{60 \times 42}{500 \times 0,1} = 50,4 \text{ tr/min.}$$

La résolution en vitesse augmente quand la fenêtre est plus longue, mais la réponse devient plus lente.

10.2 Méthode B : mesurer la période entre deux fronts

Si deux fronts consécutifs d'une voie sont séparés de ΔT :

$$RPM = \frac{60}{N_c \Delta T}.$$

Cette méthode est souvent meilleure à faible vitesse mais devient sensible à la résolution temporelle et au jitter à grande vitesse.

10.3 Input Capture du Timer1

La broche ICP1 (*Input Capture Pin 1*, PB0 / Arduino D8) peut provoquer la copie matérielle de TCNT1 vers ICR1 lors d'un front. L'événement dispose du vecteur `TIMER1_CAPT`. Ce mécanisme réduit l'incertitude liée à la latence d'une ISR logicielle.

```
volatile uint16_t last_capture = 0;
volatile uint16_t period_ticks = 0;

ISR(TIMER1_CAPT_vect) {
    uint16_t now = ICR1;
    period_ticks = now - last_capture; // arithmetique modulo 16 bits
    last_capture = now;
}
```

Si le timer tourne à $f_t = f_{CPU}/N$, alors $\Delta T = period_ticks/f_t$.

10.4 Choix de méthode

Méthode	Avantages	Limites
Comptage/fenêtre	Simple, robuste à vitesse moyenne/élevée	Quantification forte à faible vitesse, compromis temps de réponse
Mesure de période	Bonne résolution à faible vitesse	Timeout à l'arrêt, attention overflow, plus sensible aux variations instantanées
Input Capture	Horodatage matériel précis	Utilise ICP1 et Timer1, configuration plus avancée