# TP 3 : MICROCONTRÔLEUR ATMEL ATMEGA 2560

Le but de ce TP sur machine est de se familiariser avec le principe de fonctionnement des interruptions. Pour cela, nous considérerons la façon dont cela est mis en place avec le microcontrôleur de type Atmel ATmega 2560 intégré à une carte Arduino.

On fournit en Annexes les éléments suivants :

- une documentation détaillée sur les instructions proposées par l' ATmega 2560,
- détails sur la carte Arduino et le microcontrôleur Atmel ATmega 2560,
- branchement de la carte au PC,
- compiler et téléverser un programme assembleur dans la mémoire flash de la carte.

## 1. Fonctionnement des interruptions avec l'ATmega 2560

Dans cette section, nous revenir sur le concept d'interruption et les détails de mise en place dans le microcontrôleur ATmega 2560.

On rappelle que les interruptions est un signal envoyé au processeur, elles représentent des événements qui nécessitent l'intervention du processeur. Lorsqu'une interruption se produit, un signal est envoyé au processeur ayant pour conséquence d'interrompre temporairement le programme exécuté par celui-ci pour aller exécuter une **routine d'interruption** (en anglais Interrupt Service Routine – ISR). Lorsque la routine d'interruption a été complètement exécutée, le processeur reprend le cours de l'exécution du programme interrompu.

Une routine d'interruption est une suite d'instructions exécutées par le microcontrôleur associée à une interruption particulière du microcontrôleur. Ainsi, chaque interruption possède sa propre routine d'interruption. Afin que le microcontrôleur puisse détecter la fin de la routine d'interruption, une instruction particulière est à placer à la fin de chaque routine. C'est l'instruction **RTI** qui a cet effet dans le jeu d'instruction des microcontrôleurs Atmel.

Les microcontrôleurs Atmel prennent en charge deux types d'interruptions :

- interruptions internes : ces interruptions sont associées aux périphériques internes au microcontrôleur, comme des timers, des comparateurs analogiques, ...
- interruptions externes : ces interruptions sont associées à des périphériques externes au microcontrôleur et connectés, comme les USART0-3, les périphériques connectés via les broches d'extension de la carte.

### 1.1. Activation des interruptions

Les interruptions ne sont pas activées par défaut sur l'ATmega 2560. Il faut indiquer au microcontrôleur qu'il doit prendre en compte certaines interruptions, sans quoi il ne sera pas possible de mettre en place des routines d'interruptions.
Pour réaliser cela, il faut trois éléments pour que le microcontrôleur prenne en compte une interruption :

- **Drapeau de l'interruption :** à chaque interruption prise en charge par l'ATmega 2560 correspond un drapeau (ou bit), ce drapeau est mis à 1 lorsqu'une interruption spécifique est généré au niveau de la carte.
- **Masque d'interruptions :** il est possible de spécifier les interruptions que le microcontrôleur doit prendre compte, cela est réalisé à l'aide d'un masque d'interruptions contenant 1 bit pour chaque interruption. Seuls les interruptions dont les bits sont à 1 dans le masque seront pris en compte. Le masque peut être modifié à

tout moment, il permet de retarder le traitement d'une interruption durant l'exécution d'une partie de programme si cela peut être gênant pour celui-ci.

- **Activation global des interruptions :** le microcontrôleur peut fonctionner en tout-ou-rien et ignorer toutes les interruptions. C'est le comportement par défaut de l'ATmega 2560. Cela est contrôlé par le bit **I** (bit 7) du registre d'état **SREG** du microcontrôleur,comme illustré ci-dessous. Ainsi, pour prendre en compte des interruptions il faut activer ce bit. L'activation ou la désactivation du bit I est réalisé à l'aide des instructions **SEI** (SEt global Interrupt) et **CLI** (CLear global Interrupt), qui active (mise à 1) et désactive (mise à 0) le bit **I** respectivement.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3F (0x5F) | I | T | H | S | V | N | Z | C | SREG |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Ainsi, pour traiter une interruption il faut que les trois points suivants soientt réunis :

1. le bit I du registre SREG doit être positionné à 1,
2. le bit de l'interruption doit être positionné à 1 dans le masque,
3. l'interruption considérée doit être levée.

## 1.2. Traitement d'une interruption

Nous allons voir les principaux éléments à considérer pour la mise en place d'une routine d'interruption.
Tout d'abord, voici ci-dessous les différentes étapes suivies par le microcontrôleur lorsqu'une interruption à prendre en compte est levée :

1. le microcontrôleur termine d'exécuter l'instruction courante, positionne le bit **I** à 0 et sauvegarde au sommet de la pile le contenu du compteur de pile **PC**,
2. le microcontrôleur charge l'instruction contenu en mémoire à l'adresse dans le vecteur d'interruptions associée à l'interruption considérée et exécute la routine d'interruption,
3. à l'exécution de l'instruction RTI (fin routine d'interruption) l'adresse contenu au sommet de la pile (sauvegardée à l'étape 1) est chargée dans le compteur ordinal **PC**, et positionne à nouveau le bit **I** à 1,
4. le microcontrôleur reprendre l'exécution du programme qui a été interrompu.

Le **vecteur d'interruption** est une table stockée en mémoire SRAM contenant pour chaque interruption l'adresse de la routine d'interruption correspondante à exécuter. A chaque interruption correspond une entrée du vecteur d'interruption. Initialement, aucune adresse n'est indiqué dans les entrées du vecteur d'interruption. Le tableau ci-dessous (colonne **Program Address**) indique pour chaque interruption l'adresse en mémoire SRAM où il faut placer l'adresse de la routine d'interruption associée à exécuter.

Par exemple, pour indiquer l'adresse de la routine associée à l'interruption RESET (levée lors de l'alimentation de la carte Arduino ou de l'appuie du bouton RESET) il faut placer les instructions suivantes en langage d'assemblage :

```
.org 0x0000
rjmp Reset
```

L'instruction `.org` indique que l'instruction suivante doit être placée à l'adresse `0x0000` en mémoire SRAM. En effet, l'adresse dans le vecteur d'interruption associée à l'interruption

`RESET` est l'adresse `0x0000`, comme spécifié dans le tableau ci-dessous. L'instruction qui suit est un saut inconditionnel vers la sous-routine de nom `Reset`, dont l'adresse sera calculée automatiquement par l'assembleur.

| Vector No. | Program Address(2) | Source | Interrupt Definition |
|---|---|---|---|
| 1 | $0000(1) | RESET | External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset |
| 2 | $0002 | INT0 | External Interrupt Request 0 |
| 3 | $0004 | INT1 | External Interrupt Request 1 |
| 4 | $0006 | INT2 | External Interrupt Request 2 |
| 5 | $0008 | INT3 | External Interrupt Request 3 |
| 6 | $000A | INT4 | External Interrupt Request 4 |
| 7 | $000C | INT5 | External Interrupt Request 5 |
| 8 | $000E | INT6 | External Interrupt Request 6 |
| 9 | $0010 | INT7 | External Interrupt Request 7 |
| 10 | $0012 | PCINT0 | Pin Change Interrupt Request 0 |
| 11 | $0014 | PCINT1 | Pin Change Interrupt Request 1 |
| 12 | $0016(3) | PCINT2 | Pin Change Interrupt Request 2 |
| 13 | $0018 | WDT | Watchdog Time-out Interrupt |
| 14 | $001A | TIMER2 COMPA | Timer/Counter2 Compare Match A |
| 15 | $001C | TIMER2 COMPB | Timer/Counter2 Compare Match B |
| 16 | $001E | TIMER2 OVF | Timer/Counter2 Overflow |
| 17 | $0020 | TIMER1 CAPT | Timer/Counter1 Capture Event |
| 18 | $0022 | TIMER1 COMPA | Timer/Counter1 Compare Match A |
| 19 | $0024 | TIMER1 COMPB | Timer/Counter1 Compare Match B |
| 20 | $0026 | TIMER1 COMPC | Timer/Counter1 Compare Match C |
| 21 | $0028 | TIMER1 OVF | Timer/Counter1 Overflow |
| 22 | $002A | TIMER0 COMPA | Timer/Counter0 Compare Match A |
| 23 | $002C | TIMER0 COMPB | Timer/Counter0 Compare match B |
| 24 | $002E | TIMER0 OVF | Timer/Counter0 Overflow |
| 25 | $0030 | SPI, STC | SPI Serial Transfer Complete |
| 26 | $0032 | USART0 RX | USART0 Rx Complete |
| 27 | $0034 | USART0 UDRE | USART0 Data Register Empty |
| 28 | $0036 | USART0 TX | USART0 Tx Complete |
| 29 | $0038 | ANALOG COMP | Analog Comparator |
| 30 | $003A | ADC | ADC Conversion Complete |

| Vector No. | Program Address(2) | Source | Interrupt Definition |
|---|---|---|---|
| 31 | $003C | EE READY | EEPROM Ready |
| 32 | $003E | TIMER3 CAPT | Timer/Counter3 Capture Event |
| 33 | $0040 | TIMER3 COMPA | Timer/Counter3 Compare Match A |
| 34 | $0042 | TIMER3 COMPB | Timer/Counter3 Compare Match B |
| 35 | $0044 | TIMER3 COMPC | Timer/Counter3 Compare Match C |
| 36 | $0046 | TIMER3 OVF | Timer/Counter3 Overflow |
| 37 | $0048 | USART1 RX | USART1 Rx Complete |
| 38 | $004A | USART1 UDRE | USART1 Data Register Empty |
| 39 | $004C | USART1 TX | USART1 Tx Complete |
| 40 | $004E | TWI | 2-wire Serial Interface |
| 41 | $0050 | SPM READY | Store Program Memory Ready |
| 42 | $0052(3) | TIMER4 CAPT | Timer/Counter4 Capture Event |
| 43 | $0054 | TIMER4 COMPA | Timer/Counter4 Compare Match A |
| 44 | $0056 | TIMER4 COMPB | Timer/Counter4 Compare Match B |
| 45 | $0058 | TIMER4 COMPC | Timer/Counter4 Compare Match C |
| 46 | $005A | TIMER4 OVF | Timer/Counter4 Overflow |
| 47 | $005C(3) | TIMER5 CAPT | Timer/Counter5 Capture Event |
| 48 | $005E | TIMER5 COMPA | Timer/Counter5 Compare Match A |
| 49 | $0060 | TIMER5 COMPB | Timer/Counter5 Compare Match B |
| 50 | $0062 | TIMER5 COMPC | Timer/Counter5 Compare Match C |
| 51 | $0064 | TIMER5 OVF | Timer/Counter5 Overflow |
| 52 | $0066(3) | USART2 RX | USART2 Rx Complete |
| 53 | $0068(3) | USART2 UDRE | USART2 Data Register Empty |
| 54 | $006A(3) | USART2 TX | USART2 Tx Complete |
| 55 | $006C(3) | USART3 RX | USART3 Rx Complete |
| 56 | $006E(3) | USART3 UDRE | USART3 Data Register Empty |
| 57 | $0070(3) | USART3 TX | USART3 Tx Complete |

# 2. Clignotement de la Led TEST avec un Timer

Dans cette section, nous allons reprendre le programme réalisé en langage d'assemblage lors du précédent TP et nous allons le modifier afin d'utiliser l'un des timers du microcontrôleur ATmega 2560 pour réaliser la temporisation de la sous-routine `Delay`. Cette méthode permettra de réaliser une temporisation beaucoup plus précise qu'en utilisant une boucle d'incrémentation d'un compteur.

## 2.1. Timers de l'Atmega 2560

Les « timers » sont utilisés à des fins différentes, comme compter le nombre d'occurrences d'un événement, le temps qui s'est écoulé ou tout simple comme un signal d'horloge de période inférieure à celle du microcontrôleur pour déclencher un traitement.
Un timer reçoit un signal d'horloge en entrée qui déclenche à nouveau cycle d'horologe l'incrémentation du contenu du registre qui lui est associé.
L'Atmega 2560 dispose de 6 timers différents numérotés de 0 à 5 :

- Timer/Counter 0 et 2 : utilisent un registre nommé **TCNT0** et **TCNT2** de 8 bits chacun,

- Timer/Counter1, 3 , 4 et 5 : utilisent un registre nommé **TCNT1**, **TCNT3**, **TCNT4** et **TCNT5** de 16 bits chacun.

Dans la suite, nous prendrons l'exemple du Timer/Counter0 pour illustrer les explications. D'autre part, chaque timer dispose de deux autres registres qui lui sont associés afin de configurer son fonctionnement (où x est à substituer par le numéro du timer :

- Registre **TCCRxB** : sélection de la source du signal d'horloge, cette source peut être interne (basé sur le signal d'horloge du processeur) ou externe au microcontrôleur. Voici ci-dessous le détail du registre **TCCT0B**, dans lequel il faut sélectionner la source via les bits **CS00**, **CS01** et **CS02**.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x25 (0x45) | FOC0A | FOC0B | – | – | WGM02 | CS02 | CS01 | CS00 | TCCR0B |
| Read/Write | W | W | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Le tableau ci-dessous (extrait de la documentation) indique la signification des différentes valeurs que peuvent prendre ces bits :

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped) |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge |

Lorsque les trois bits sont à 0, aucune source n'est sélectionnée pour le timer ; les 5 valeurs suivantes permettent de choisir le signal d'horloge interne au microcontrôleur pour le timer (**CS00** à uniquement signal original, sinon la fréquence est divisée par la valeur indiquée) ; enfin les deux dernières valeurs correspondent à signal externe au microcontrôleur.

- Registre **TIMSKx** : ce registre correspond contient un masque permettant d'indiquer si une interruption doit être levée suite à un overflow survenant sur le timer **x**. Voici-ci le détail du registre **TIMSK0**, dans lequel il faut positionner le bit **TOIE0** pour générer une interruption en cas d'overflow. Les bits **OCIE0B** ou **OCIE0A** permettent de lever une interruption lorsque le timer a été programmé pour atteindre une valeur définie.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x6E) | – | – | – | – | – | OCIE0B | OCIE0A | TOIE0 | TIMSK0 |
| Read/Write | R | R | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Les timers peuvent être configurés de diverses manières, nous allons décrire une façon de les utiliser qui nous permettra par la suite de détecter une interruption suite à un overflow sur le registre d'un timer pour exécuter la routine d'interruption associée.

## 2.2. Paramétrage du Timer/Counter 0

Comme indiqué précédemment, nous allons utiliser le Timer/Counter 0 pour incrémenter le contenu de son registre **TCNT0** à une certaine fréquence à partir du signal d'horloge interne au microcontrôleur. Lorsqu'un overflow se produira sur le registre **TCNT0**, l'interruption **TIMER0 OVF** sera levée et la routine associée que nous mettrons en place sera exécutée. Cette routine sera la base de notre fonction de temporisation.

### a) Initialisation du vecteur d'interruptions

Nous allons tout d'abord commencer par indiquer dans l'entrée associée à l'interruption **TIMER OVF** du vecteur d'interruption l'adresse de la routine à exécuter.

**Question 1 :** Utiliser le tableau de vecteur d'interruptions fourni précédemment pour déterminer l'adresse de l'entrée correspondant à l'interruption **TIMER0 OVF**. Remplacez xxx dans le code suivant par utiliser avec la directive `.org`. Dans le code ci-dessous, `Reset` fera référence à la routine exécuter lors de l'initialisation de la ccarte et `Overflow` fera référence à la routine d'interruption à exécuter.

```
.org 0x0000
rjmp Reset

.org xxx
rjmp Overflow
```

**Question 2 :** Reprenez le programme assembleur faisant clignoter la Led TEST du TP précédent et ajoutez les lignes ci-dessus en début de programme.

Dans la suite, nous allons initialiser le Timer/Counter 0 pour réaliser notre fonction de temporisation.

### b) Initialisation du registre TCCR0B

Nous allons maintenant initialiser la source du signal d'horloge associé au Timer/Counter 0. Nous allons prendre le signal d'horloge interne au microcontrôleur et à partir duquel nous allons définir une période d'incrémentation du registre **TCNT0** égale à la fréquence d'horloge originale divisé par 1024 (c'est-à-dire que le timer sera incrémenté 16 million/1024 = 15 625 fois par seconde).

**Question 3 :** Ajoutez les lignes nécessaire dans la routine `Reset` pour initialiser le registre **TCCR0B** comme indiqué ci-dessus. Vous utiliserez la présentation faite précédemment pour le registre **TCCR0B**.

### c) Initialisation du registre TIMSK0

Nous devons activer le déclenchement d'une interruption lors d'un overflow sur le registre **TCNT0**. Pour cela, il faut positionner le bon masque de bits dans le registre **TIMSK0**.

**Question 4 :** Pour positionner le masque dans le registre **TIMSK0**, nous devons utiliser l'instruction **STS**. Consulter la documentation sur le jeu d'instruction fournie en annexes pour expliquer à quoi sert cette instruction par rapport à l'instruction **OUT**.

**Question 5 :** Ajoutez les lignes nécessaire dans la routine `Reset` pour initialiser le registre **TIMSK0** comme indiqué ci-dessus. Vous utiliserez la présentation faite précédemment pour le registre **TIMSK0**.

### d) Activation des interruptions

Nous devons ensuite indiquer au processeur qu'il doit prendre en compte les interruptions qui sont levées. Pour cela, nous devons positionner le bit **I** dans le registre d'état **SREG**.

**Question 6 :** Ajouter dans la routine `Reset` l'instruction permettant de positionner le bit I du registre d'état SREG. Vous vous aiderez de la documentation sur le jeu d'instruction fournie en annexes.

### e) Initialisation du registre TCNT0

Pour finir l'initialisation du Timer/Counter 0, nous allons placer une valeur initiale dans le registre **TCNT0**. La valeur de départ sera la valeur nulle, car nous désirons incrémenter ce registre 255 fois avant de générer un overflow.

**Question 7 :** Ajouter dans la routine `Reset` les instructions nécessaires pour initialiser le registre **TCNT0** à 0.

## 2.3. Réalisation de la routine d'interruption

Après avoir initialisé le Timer/Counter 0, nous allons maintenant définir la routine `Overflow` qui sera exécutée par le microcontrôleur lors de la levée de l'interruption **TIMER0 OVF**.

La routine `Overflow` doit réaliser un traitement assez simple, elle doit uniquement un registre du microcontrôleur afin de compter le nombre d'occurrences d'un overflow sur le registre **TCNT0** du Timer/Counter 0.

**Question 8 :** Créez une routine Overflow permettant d'incrémenter de 1 le contenu du registre r17. N'oubliez que cette routine est associée à une interruption et donc le microcontrôleur a besoin de savoir où est la fin de cette routine pour reprendre l'exécution du programme qui a été interrompu.

## 2.4. Modification de la routine Delay

C'est la dernière étape du programme dans laquelle il ne nous reste plus qu'à modifier ma routine `Delay` réalisée lors du précédent TP.
Pour rappel, voici ci-dessous un exemple pour cette routine réalisant une temporisation de 1 s :

```
Delay: LDI R26, low(4000) ; temporisation de 1s
    LDI R27, high(4000)
Bou1: LDI R24, low(800)
    LDI R25, high(800)
Bou: SBIW R24, 1
    NOP
    BRNE Bou
    SBIW R26, 1
```

```
        BRNE Bou1
        RET
```

La nouvelle routine doit tout d'abord initialiser le registre r17 à 0, puis contenir une boucle permettant de tester si le registre r17 (incrémentée par la routine `Overflow`) a atteint la valeur 60. Lorsque cette valeur est atteinte, le microcontrôleur doit sortir de la routine `Delay`.

**Question 9 :** Expliquez pourquoi nous devons tester si la valeur placée dans le registre r17 par la routine Overflow est égale à 60.

**Question 10 :** Donnez le code de la routine Delay ainsi modifiée.

# ANNEXES

# 3. Tableau d'instructions de l'ATmega 2560

*Tableau 1 : Instructions arithmétiques et logiques*

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| **ARITHMETIC AND LOGIC INSTRUCTIONS** | | | | | |
| ADD | Rd, Rr | Add two Registers | Rd ← Rd + Rr | Z, C, N, V, H | 1 |
| ADC | Rd, Rr | Add with Carry two Registers | Rd ← Rd + Rr + C | Z, C, N, V, H | 1 |
| ADIW | Rdl,K | Add Immediate to Word | Rdh:Rdl ← Rdh:Rdl + K | Z, C, N, V, S | 2 |
| SUB | Rd, Rr | Subtract two Registers | Rd ← Rd - Rr | Z, C, N, V, H | 1 |
| SUBI | Rd, K | Subtract Constant from Register | Rd ← Rd - K | Z, C, N, V, H | 1 |
| SBC | Rd, Rr | Subtract with Carry two Registers | Rd ← Rd - Rr - C | Z, C, N, V, H | 1 |
| SBCI | Rd, K | Subtract with Carry Constant from Reg. | Rd ← Rd - K - C | Z, C, N, V, H | 1 |
| SBIW | Rdl,K | Subtract Immediate from Word | Rdh:Rdl ← Rdh:Rdl - K | Z, C, N, V, S | 2 |
| AND | Rd, Rr | Logical AND Registers | Rd ← Rd • Rr | Z, N, V | 1 |
| ANDI | Rd, K | Logical AND Register and Constant | Rd ← Rd • K | Z, N, V | 1 |
| OR | Rd, Rr | Logical OR Registers | Rd ← Rd v Rr | Z, N, V | 1 |
| ORI | Rd, K | Logical OR Register and Constant | Rd ← Rd v K | Z, N, V | 1 |
| EOR | Rd, Rr | Exclusive OR Registers | Rd ← Rd ⊕ Rr | Z, N, V | 1 |
| COM | Rd | One's Complement | Rd ← 0xFF – Rd | Z, C, N, V | 1 |
| NEG | Rd | Two's Complement | Rd ← 0x00 – Rd | Z, C, N, V, H | 1 |
| SBR | Rd,K | Set Bit(s) in Register | Rd ← Rd v K | Z, N, V | 1 |
| CBR | Rd,K | Clear Bit(s) in Register | Rd ← Rd • (0xFF - K) | Z, N, V | 1 |
| INC | Rd | Increment | Rd ← Rd + 1 | Z, N, V | 1 |
| DEC | Rd | Decrement | Rd ← Rd – 1 | Z, N, V | 1 |
| TST | Rd | Test for Zero or Minus | Rd ← Rd • Rd | Z, N, V | 1 |
| CLR | Rd | Clear Register | Rd ← Rd ⊕ Rd | Z, N, V | 1 |
| SER | Rd | Set Register | Rd ← 0xFF | None | 1 |
| MUL | Rd, Rr | Multiply Unsigned | R1:R0 ← Rd x Rr | Z, C | 2 |
| MULS | Rd, Rr | Multiply Signed | R1:R0 ← Rd x Rr | Z, C | 2 |
| MULSU | Rd, Rr | Multiply Signed with Unsigned | R1:R0 ← Rd x Rr | Z, C | 2 |
| FMUL | Rd, Rr | Fractional Multiply Unsigned | R1:R0 ← (Rd x Rr) << 1 | Z, C | 2 |
| FMULS | Rd, Rr | Fractional Multiply Signed | R1:R0 ← (Rd x Rr) << 1 | Z, C | 2 |
| FMULSU | Rd, Rr | Fractional Multiply Signed with Unsigned | R1:R0 ← (Rd x Rr) << 1 | Z, C | 2 |

*Tableau 2 : Instructions de branchement*

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| **BRANCH INSTRUCTIONS** | | | | | |
| RJMP | k | Relative Jump | PC ← PC + k + 1 | None | 2 |
| IJMP | | Indirect Jump to (Z) | PC ← Z | None | 2 |
| EIJMP | | Extended Indirect Jump to (Z) | PC ←(EIND:Z) | None | 2 |
| JMP | k | Direct Jump | PC ← k | None | 3 |
| RCALL | k | Relative Subroutine Call | PC ← PC + k + 1 | None | 4 |
| ICALL | | Indirect Call to (Z) | PC ← Z | None | 4 |
| EICALL | | Extended Indirect Call to (Z) | PC ←(EIND:Z) | None | 4 |
| CALL | k | Direct Subroutine Call | PC ← k | None | 5 |
| RET | | Subroutine Return | PC ← STACK | None | 5 |
| RETI | | Interrupt Return | PC ← STACK | I | 5 |
| CPSE | Rd,Rr | Compare, Skip if Equal | if (Rd = Rr) PC ← PC + 2 or 3 | None | 1/2/3 |
| CP | Rd,Rr | Compare | Rd − Rr | Z, N, V, C, H | 1 |
| CPC | Rd,Rr | Compare with Carry | Rd − Rr − C | Z, N, V, C, H | 1 |
| CPI | Rd,K | Compare Register with Immediate | Rd − K | Z, N, V, C, H | 1 |
| SBRC | Rr, b | Skip if Bit in Register Cleared | if (Rr(b)=0) PC ← PC + 2 or 3 | None | 1/2/3 |
| SBRS | Rr, b | Skip if Bit in Register is Set | if (Rr(b)=1) PC ← PC + 2 or 3 | None | 1/2/3 |
| SBIC | P, b | Skip if Bit in I/O Register Cleared | if (P(b)=0) PC ← PC + 2 or 3 | None | 1/2/3 |
| SBIS | P, b | Skip if Bit in I/O Register is Set | if (P(b)=1) PC ← PC + 2 or 3 | None | 1/2/3 |
| BRBS | s, k | Branch if Status Flag Set | if (SREG(s) = 1) then PC←PC+k + 1 | None | 1/2 |
| BRBC | s, k | Branch if Status Flag Cleared | if (SREG(s) = 0) then PC←PC+k + 1 | None | 1/2 |
| BREQ | k | Branch if Equal | if (Z = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRNE | k | Branch if Not Equal | if (Z = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRCS | k | Branch if Carry Set | if (C = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRCC | k | Branch if Carry Cleared | if (C = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRSH | k | Branch if Same or Higher | if (C = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRLO | k | Branch if Lower | if (C = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRMI | k | Branch if Minus | if (N = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRPL | k | Branch if Plus | if (N = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRGE | k | Branch if Greater or Equal, Signed | if (N ⊕ V= 0) then PC ← PC + k + 1 | None | 1/2 |
| BRLT | k | Branch if Less Than Zero, Signed | if (N ⊕ V= 1) then PC ← PC + k + 1 | None | 1/2 |
| BRHS | k | Branch if Half Carry Flag Set | if (H = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRHC | k | Branch if Half Carry Flag Cleared | if (H = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRTS | k | Branch if T Flag Set | if (T = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRTC | k | Branch if T Flag Cleared | if (T = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRVS | k | Branch if Overflow Flag is Set | if (V = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRVC | k | Branch if Overflow Flag is Cleared | if (V = 0) then PC ← PC + k + 1 | None | 1/2 |
| BRIE | k | Branch if Interrupt Enabled | if ( I = 1) then PC ← PC + k + 1 | None | 1/2 |
| BRID | k | Branch if Interrupt Disabled | if ( I = 0) then PC ← PC + k + 1 | None | 1/2 |

*Tableau 3 : Instructions de manipulation et de test des bits*

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| **BIT AND BIT-TEST INSTRUCTIONS** | | | | | |
| SBI | P,b | Set Bit in I/O Register | I/O(P,b) ← 1 | None | 2 |
| CBI | P,b | Clear Bit in I/O Register | I/O(P,b) ← 0 | None | 2 |
| LSL | Rd | Logical Shift Left | Rd(n+1) ← Rd(n), Rd(0) ← 0 | Z, C, N, V | 1 |
| LSR | Rd | Logical Shift Right | Rd(n) ← Rd(n+1), Rd(7) ← 0 | Z, C, N, V | 1 |
| ROL | Rd | Rotate Left Through Carry | Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7) | Z, C, N, V | 1 |
| ROR | Rd | Rotate Right Through Carry | Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0) | Z, C, N, V | 1 |
| ASR | Rd | Arithmetic Shift Right | Rd(n) ← Rd(n+1), n=0..6 | Z, C, N, V | 1 |
| SWAP | Rd | Swap Nibbles | Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0) | None | 1 |
| BSET | s | Flag Set | SREG(s) ← 1 | SREG(s) | 1 |
| BCLR | s | Flag Clear | SREG(s) ← 0 | SREG(s) | 1 |
| BST | Rr, b | Bit Store from Register to T | T ← Rr(b) | T | 1 |
| BLD | Rd, b | Bit load from T to Register | Rd(b) ← T | None | 1 |
| SEC | | Set Carry | C ← 1 | C | 1 |
| CLC | | Clear Carry | C ← 0 | C | 1 |
| SEN | | Set Negative Flag | N ← 1 | N | 1 |
| CLN | | Clear Negative Flag | N ← 0 | N | 1 |
| SEZ | | Set Zero Flag | Z ← 1 | Z | 1 |
| CLZ | | Clear Zero Flag | Z ← 0 | Z | 1 |
| SEI | | Global Interrupt Enable | I ← 1 | I | 1 |
| CLI | | Global Interrupt Disable | I ← 0 | I | 1 |
| SES | | Set Signed Test Flag | S ← 1 | S | 1 |
| CLS | | Clear Signed Test Flag | S ← 0 | S | 1 |
| SEV | | Set Twos Complement Overflow. | V ← 1 | V | 1 |
| CLV | | Clear Twos Complement Overflow | V ← 0 | V | 1 |
| SET | | Set T in SREG | T ← 1 | T | 1 |
| CLT | | Clear T in SREG | T ← 0 | T | 1 |
| SEH | | Set Half Carry Flag in SREG | H ← 1 | H | 1 |
| CLH | | Clear Half Carry Flag in SREG | H ← 0 | H | 1 |

*Tableau 4 : Instructions de transfert de données*

| Mnemonics | Operands | Description | Operation | Flags | #Clocks |
|---|---|---|---|---|---|
| **DATA TRANSFER INSTRUCTIONS** | | | | | |
| MOV | Rd, Rr | Move Between Registers | Rd ← Rr | None | 1 |
| MOVW | Rd, Rr | Copy Register Word | Rd+1:Rd ← Rr+1:Rr | None | 1 |
| LDI | Rd, K | Load Immediate | Rd ← K | None | 1 |
| LD | Rd, X | Load Indirect | Rd ← (X) | None | 2 |
| LD | Rd, X+ | Load Indirect and Post-Inc. | Rd ← (X), X ← X + 1 | None | 2 |
| LD | Rd, - X | Load Indirect and Pre-Dec. | X ← X - 1, Rd ← (X) | None | 2 |
| LD | Rd, Y | Load Indirect | Rd ← (Y) | None | 2 |
| LD | Rd, Y+ | Load Indirect and Post-Inc. | Rd ← (Y), Y ← Y + 1 | None | 2 |
| LD | Rd, - Y | Load Indirect and Pre-Dec. | Y ← Y - 1, Rd ← (Y) | None | 2 |
| LDD | Rd,Y+q | Load Indirect with Displacement | Rd ← (Y + q) | None | 2 |
| LD | Rd, Z | Load Indirect | Rd ← (Z) | None | 2 |
| LD | Rd, Z+ | Load Indirect and Post-Inc. | Rd ← (Z), Z ← Z+1 | None | 2 |
| LD | Rd, -Z | Load Indirect and Pre-Dec. | Z ← Z - 1, Rd ← (Z) | None | 2 |
| LDD | Rd, Z+q | Load Indirect with Displacement | Rd ← (Z + q) | None | 2 |
| LDS | Rd, k | Load Direct from SRAM | Rd ← (k) | None | 2 |
| ST | X, Rr | Store Indirect | (X) ← Rr | None | 2 |
| ST | X+, Rr | Store Indirect and Post-Inc. | (X) ← Rr, X ← X + 1 | None | 2 |
| ST | - X, Rr | Store Indirect and Pre-Dec. | X ← X - 1, (X) ← Rr | None | 2 |
| ST | Y, Rr | Store Indirect | (Y) ← Rr | None | 2 |
| ST | Y+, Rr | Store Indirect and Post-Inc. | (Y) ← Rr, Y ← Y + 1 | None | 2 |
| ST | - Y, Rr | Store Indirect and Pre-Dec. | Y ← Y - 1, (Y) ← Rr | None | 2 |
| STD | Y+q,Rr | Store Indirect with Displacement | (Y + q) ← Rr | None | 2 |
| ST | Z, Rr | Store Indirect | (Z) ← Rr | None | 2 |
| ST | Z+, Rr | Store Indirect and Post-Inc. | (Z) ← Rr, Z ← Z + 1 | None | 2 |
| ST | -Z, Rr | Store Indirect and Pre-Dec. | Z ← Z - 1, (Z) ← Rr | None | 2 |
| STD | Z+q,Rr | Store Indirect with Displacement | (Z + q) ← Rr | None | 2 |
| STS | k, Rr | Store Direct to SRAM | (k) ← Rr | None | 2 |
| LPM | | Load Program Memory | R0 ← (Z) | None | 3 |
| LPM | Rd, Z | Load Program Memory | Rd ← (Z) | None | 3 |
| LPM | Rd, Z+ | Load Program Memory and Post-Inc | Rd ← (Z), Z ← Z+1 | None | 3 |
| ELPM | | Extended Load Program Memory | R0 ← (RAMPZ:Z) | None | 3 |
| ELPM | Rd, Z | Extended Load Program Memory | Rd ← (RAMPZ:Z) | None | 3 |
| ELPM | Rd, Z+ | Extended Load Program Memory | Rd ← (RAMPZ:Z), RAMPZ:Z ←RAMPZ:Z+1 | None | 3 |
| SPM | | Store Program Memory | (Z) ← R1:R0 | None | - |
| IN | Rd, P | In Port | Rd ← P | None | 1 |
| OUT | P, Rr | Out Port | P ← Rr | None | 1 |
| PUSH | Rr | Push Register on Stack | STACK ← Rr | None | 2 |
| POP | Rd | Pop Register from Stack | Rd ← STACK | None | 2 |
| **MCU CONTROL INSTRUCTIONS** | | | | | |
| NOP | | No Operation | | None | 1 |
| SLEEP | | Sleep | (see specific descr. for Sleep function) | None | 1 |
| WDR | | Watchdog Reset | (see specific descr. for WDR/timer) | None | 1 |
| BREAK | | Break | For On-chip Debug Only | None | N/A |

# 4. Matériel utilisé

Vous utiliserez une carte Arduino MEGA 2560 disposant d'un microcontrôleur Atmel ATmega 2560. Arduino est une entreprise Italienne concevant diverses cartes de développement grand public basé sur des microcontrôleur Atmel et un environnement de développement facile à utiliser. Atmel est un fabriquant de composants électroniques spécialisé dans les microcontrôleurs.

Un microcontrôleur, contrairement à un processeur d'ordinateur classique, est une sorte de mini-ordinateur qui embarque dans une puce électronique pratiquement tout ce qui nécessaire pour son fonctionnement. Cette puce contient entre autres :

- un processeur (CPU) : exécutant les instructions du programme,
- une mémoire RAM : utilisée durant l'exécution du programme,
- une mémoire morte ROM : utilisée pour stocker des données permanentes,
- une mémoire flash : utilisée pour stocker les programmes,
- des entrées-sorties : pour communiquer avec l'extérieur.

Le schéma ci-dessous illustre les différents composants de la carte Arduino MEGA 2560 :

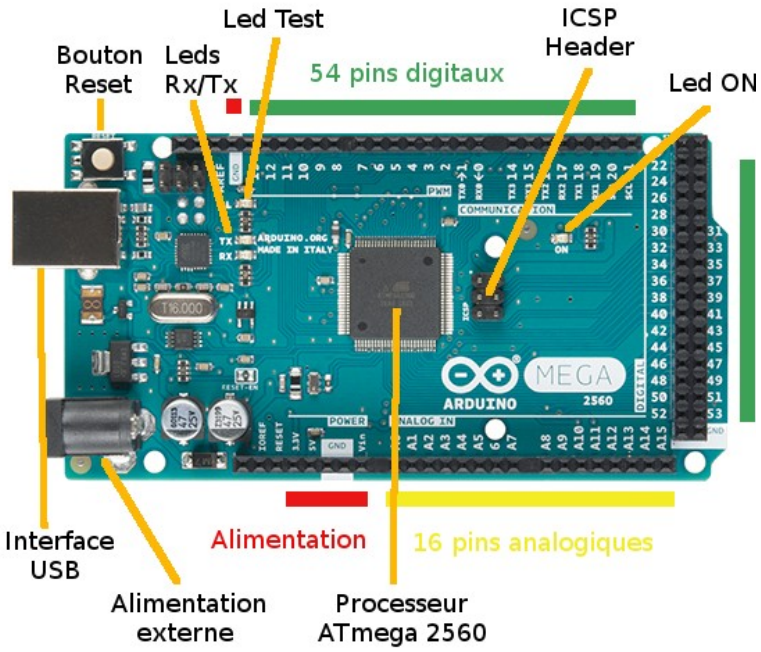



*Figure 1: Composants de la carte Arduino MEGA*

Voici ci-dessous les caractéristiques détaillées du microcontrôleur Atmel Atmega 2560 que nous utiliserons :

- Fréquence CPU : 16 MHz
- Type CPU : processeur de type RISC, 8 bits
- Mémoire SRAM : 8 Ko
- Mémoire EEPROM : 4 Ko

- Mémoire Flash : 256 Ko (dont 8 Ko utilisé par bootloader)
- Nbr Pins I/O digital : 54 (dont 14 sorties PWM – Pulse Width Modulation)
- Nbr Pins analogique en entrée : 16
- Voltage de fonctionnement : 5V
- Courant Pins I/O : 40 mA (DC)
- Courant autre Pins : 50 mA (DC)

Le schéma ci-dessous indique les connexions réalisées sur les différentes broches de l'ATmega 2560 :

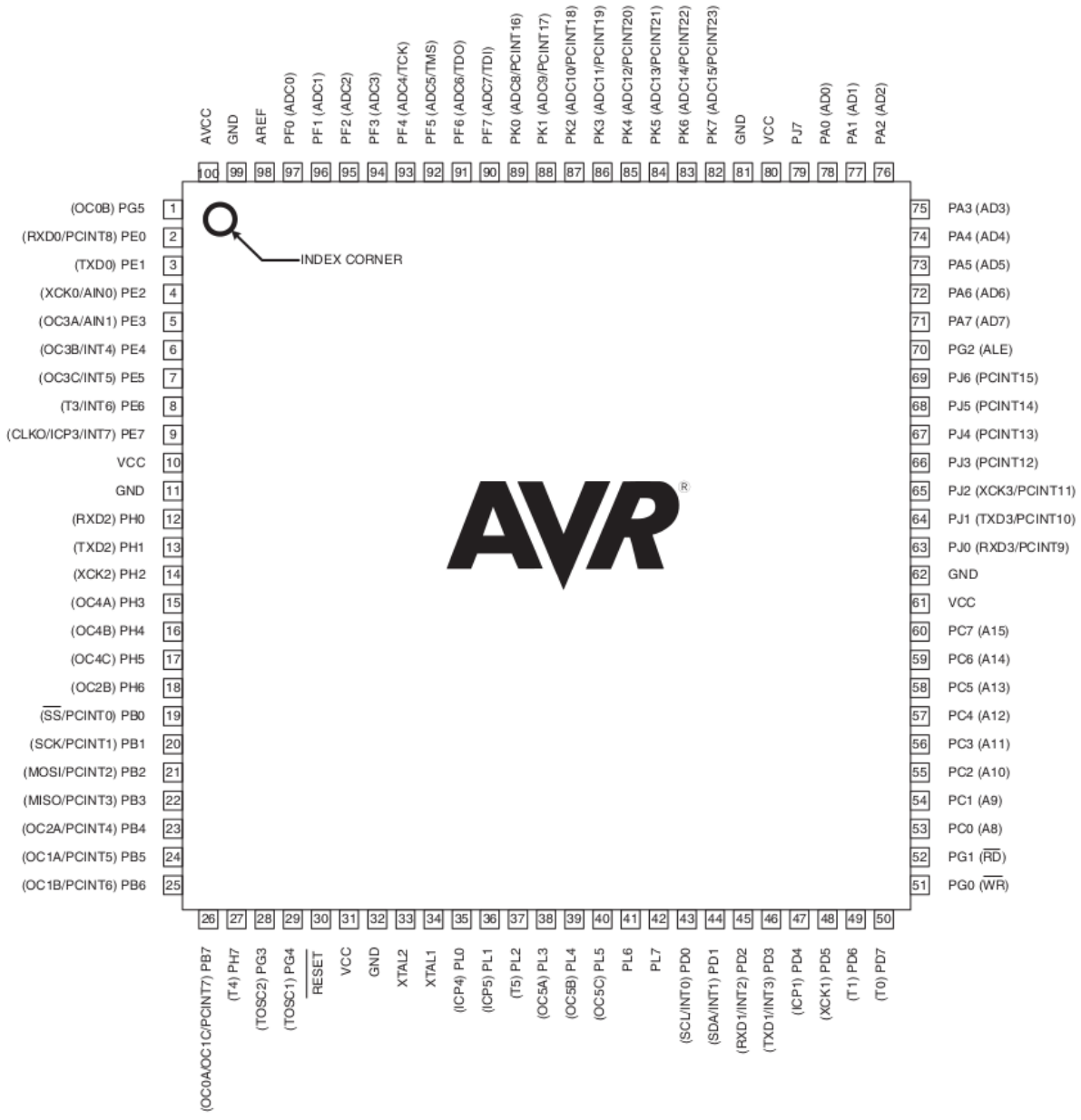



*Figure 2 : Connectique de l'ATmega 2560*

Le schéma ci-dessous illustre les composants internes de l'ATmega 2560 en complément du schéma de la Figure 1 :

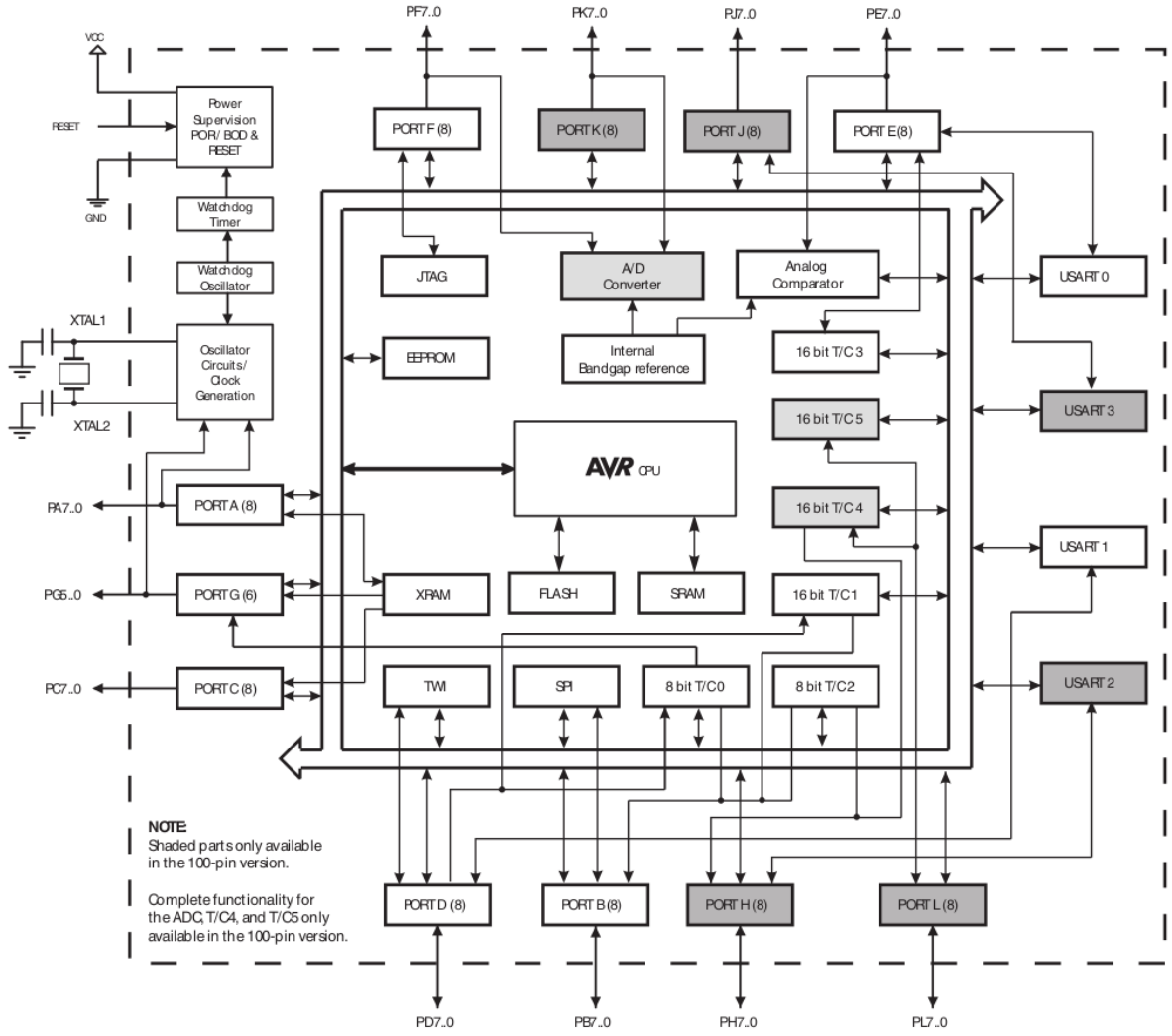



*Figure 3 : Schéma interne de l'ATmega 2560*

Les mémoires des cartes Arduino sont organisées suivant une architecture de type Harvard, c'est-à-dire que des bus séparés sont utilisés pour l'espace mémoire du programme (mémoire Flash) et l'espace mémoire associée aux données (mémoire SRAM). Cela autorise la réutilisation des plages mêmes plage d'adresses mémoire comme spécifié ci-dessous.

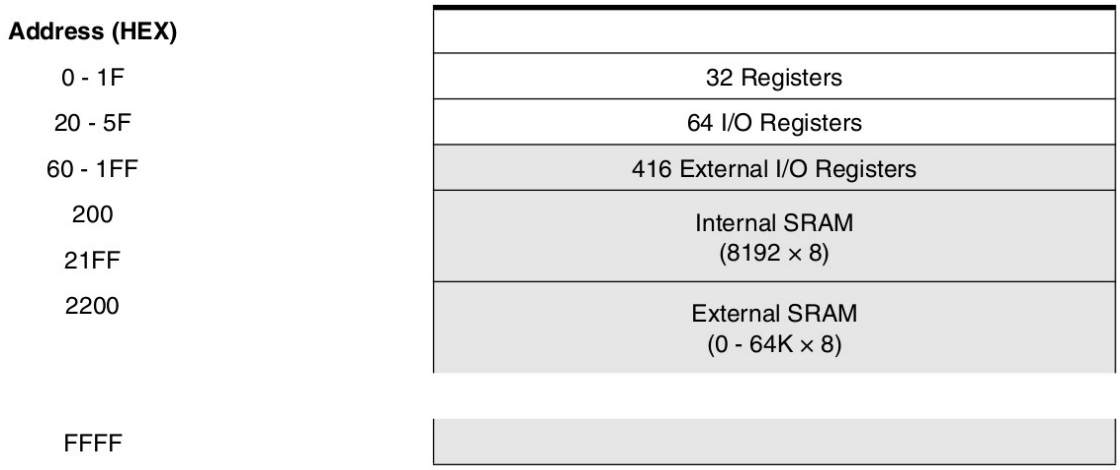
### a) Mémoire Flash

La mémoire flash est une mémoire de type ROM (Read Only Memory) et donc non volatile, c'est-à-dire que les informations sont enregistrées de façon permanente. Cette mémoire a une taille totale de 256 Koctets sur l'ATmega 2560. Les instructions du jeu d'instructions AVR sont toutes de taille 16 ou 32 bits, ainsi cet espace mémoire est organisé 128 Kmots de 16 bits. Cette espace est divisé en 2 sections distinctes : la section associée au programme de boot (des adresses 0x10000 à 0x1FFFF) et celle associée au programme exécuté (des adresses 0x00000 à 0x0FFFF). D'un point de vue physique, cette mémoire peut être modifiée environ 10 000 fois.
Enfin, il est à noter que le registre de compteur ordinal (**PC**) est de taille 17 bits.

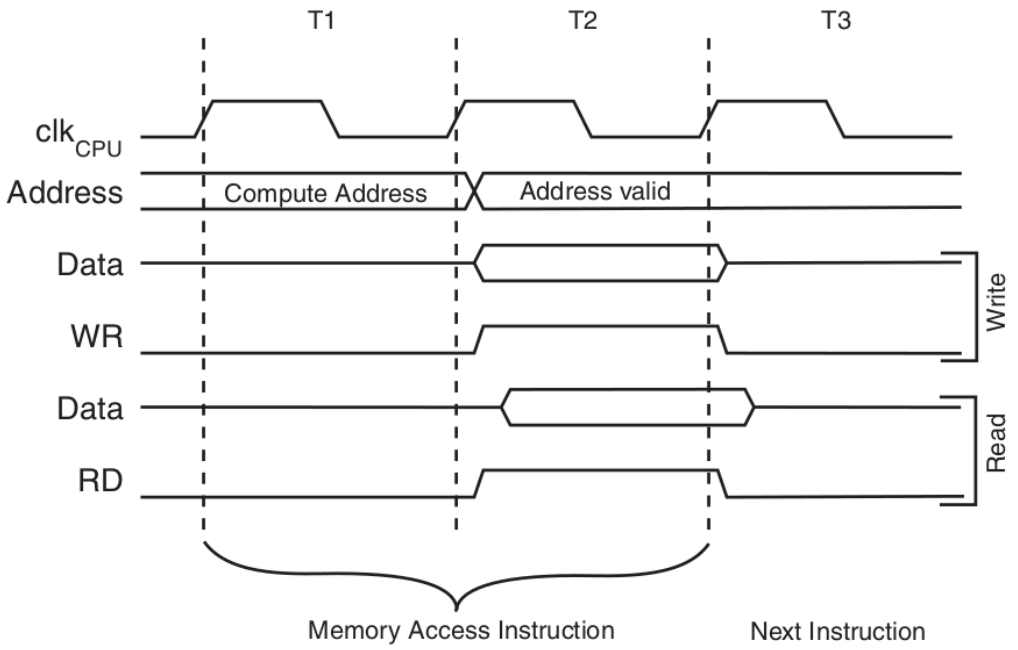
### b) Mémoire SRAM

La mémoire SRAM (Static Random Access memory) est de type volatile, les informations ne sont pas enregistrées de façon permanente (elles disparaissent dès lors que la carte n'est plus

alimentée). Cette mémoire permet de stocker les informations (instructions et données) du programme en exécution dans la mémoire dite interne, mais également d'accéder aux 32 registres généraux et aux 64 registres d'Entrées-Sorties. D'autre part, il est possible d'ajouter de la mémoire dite externe accessible à partir d'une certaine valeur d'adresse.
Le schéma ci-dessous donne les plages d'adresses pour chaque type d'informations :



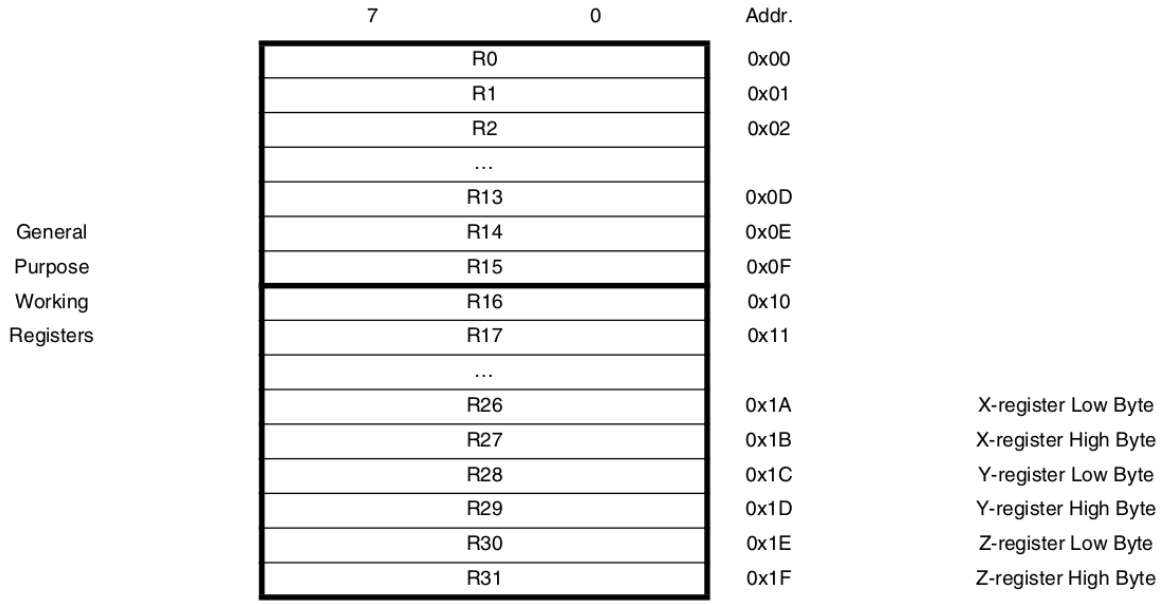

L'accès aux données nécessite 2 cycles d'horloge processeur en lecture ou écriture, comme illustré dans le schéma ci-dessous (3 cycles processeur pour un accès en SRAM externe :




## 4.2. REGISTRES DE l'ATmega 2560

### a) Registres de travail

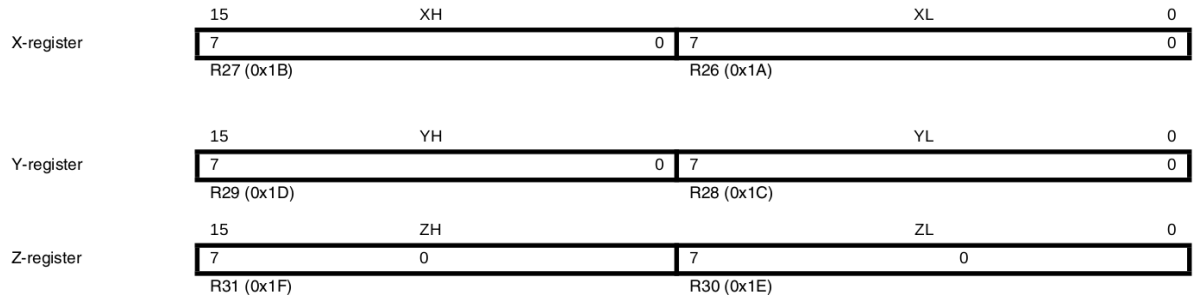
Le microcontrôleur ATmega 2560 utilisé en TP dispose de 32 registres de travail de 8 bits. Ces registres sont utilisables à l'aide de leur nom spécifique de R0 à R31, et sont associés au 32 premiers emplacement de l'espace de données de l'utilisateur. Le schéma ci-dessous indique les adresses en mémoire correspondantes aux 32 registres.




Il est à noter que le microcontrôleur ATmega 2560 dispose de 3 registres d'adresses dénommés X, Y et Z de 16 bits. Ces 3 registres d'adresses sont utilisés pour stocker une adresse pour faire référence à des données situées dans l'espace mémoire de l'utilisateur. Ces 3 registres sont obtenu en regroupant 2 registres de 8 bits parmi les 32 registres comme précisé ci-dessous :

- **Registre X :** associé aux registres R26 (8 bits partie basse) et R27 (8 bits partie haute)
- **Registre Y :** associé aux registres R28 (8 bits partie basse) et R29 (8 bits partie haute)
- **Registre Z :** associé aux registres R30 (8 bits partie basse) et R31 (8 bits partie haute)

Cela est illustré par le schéma ci-dessous :




### b) Registre de pile

On rappel que le registre de pile permet de stocker des données temporaires pour des variables locales et pour les adresses de retour après le traitement d'une interruption ou l'appel à une fonction. La valeur initiale de ce registre est la plus grande adresse de la mémoire SRAM. Ce registre est manipulé par les instructions PUSH, qui décrémente l'adresse du registre, et POP, qui incrémente l'adresse du registre. Le contenu du registre de pile est incrémenté/décrémenté de 1 lorsqu'une donnée est empilée/dépilée et de 3 lorsqu'il s'agit d'une adresse.

Le registre de pile **SP** est implémenté à l'aide de 2 registres de 8 bits dans l'espace d'entrées-sorties, comme illustré ci-dessous :

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3E (0x5E) | **SP15** | **SP14** | **SP13** | **SP12** | **SP11** | **SP10** | **SP9** | **SP8** | **SPH** |
| 0x3D (0x5D) | **SP7** | **SP6** | **SP5** | **SP4** | **SP3** | **SP2** | **SP1** | **SP0** | **SPL** |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

### c) Registre d'état

Le processeur ATmega 2560 dispose d'un registre d'état fournissant des informations sur l'instruction arithmétique et logique exécutée le plus récemment. Ces informations sont utilisées par les instructions de test conditionnels afin de réaliser des branchement dans le programme.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x3F (0x5F) | **I** | **T** | **H** | **S** | **V** | **N** | **Z** | **C** | **SREG** |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Comme illustré ci-dessus, le registre d'état **SREG** est composé de 8 bits :

- Interruption (**I**) – bit 7 : Lorsque ce bit est à 1 le processeur traite une interruption, sinon celle-ci est ignorée. Ce bit est remis à zéro au retour d'une routine de traitement d'interruption.
- Copie de bit (**T**) - bit 6 : bit manipulé par les instructions BLD (Bit LoaD) et BST (Bit STore) pour chargé un bit dans **T** depuis un registre ou copier le contenu de **T** vers un registre respectivement.
- Demi-retenu (**H**) – bit 5 : bit indiquant la présence d'un demi-retenue produit par certaines instructions arithmétiques et logiques.
- Signe (**S**) – bit 4 : bit donnant signe du résultat généré par une instruction arithmétique.
- Overflow (**O**) – bit 3 : bit indiquant si un overflow a été généré par une instruction arithmétique.
- Négatif (**N**) – bit 2 : bit signalant si le résultat produit par une instruction arithmétique et logique est négatif.
- Zéro (**Z**) – bit 1 : bit indiquant si le résultat produit par une instruction arithmétique et logique est nul.
- Retenue (**C**) – bit 0 : bit indiquant si une instruction arithmétique et logique a produit une retenue (carry).

# 5. Mise en route et premier programme

Dans cette section, nous allons voir la procédure à suivre par la suite pour charger et exécuter un programme sur la carte Arduino MEGA. Cette procédure est similaire pour d'autres cartes Arduino à quelques paramétrage près.

### 5.1. Branchement

Avant d'aller plus loin, vous devez brancher la carte Arduino MEGA fournie pour le TP et utiliser le câble USB pour relier l'interface USB externe de la carte à un port USB de l'ordinateur. Nous utiliserons le câble USB pour alimenter la carte Arduino et également échanger des données avec la mémoire du microcontrôleur.

Une fois la carte branchée, vous devez voir la Led ON (voir Figure1) s'allumer et éventuellement les Led Rx/Tx.

**Attention :** pour éviter d'endommager la carte Arduino vous ne devez en aucun cas toucher aux composants électronique avec vos doigts, faire attention à l'électricité statique dû au contact avec certains vêtements, et brancher/débrancher avec précaution l'interface USB.

## 6. Premier programme en langage d'assemblage

Dans cette section, vous allez étudier un premier programme décrit avec le langage d'assemblage de l'ATmega 2560.

On rappelle qu'un langage d'assemblage est une représentation alphanumérique du code machine. Les instructions définissant un programme en langage d'assemblage est spécifique à chaque microcontrôleur, ainsi chaque constructeur de microcontrôleur propose un jeu d'instructions propre.

### 6.1. Principaux modes d'adressage

Dans le jeu d'instruction de chaque microcontrôleur il existe plusieurs modes d'adressage pour chaque instruction. A chaque mode d'adressage correspond une façon d'accéder aux données.

Le tableau suivant présente quelques exemples pour chaque mode d'adressage.

| Mode d'adressage | Type de l'opérande | Exemple | Signification |
|---|---|---|---|
| Immédiat | valeur | ldi R16, 24<br>ldi R16, 0x1A<br>ldi R16, 0b00011010 | Valeur décimale<br>Valeur hexadécimale<br>Valeur binaire |
| Direct | Adresse sur 2 octets (utilisation registres d'adresse X, Y et Z) | ld R16, X | R16 <- (X) |
| Direct avec décrément avant | Adresse sur 2 octets (utilisation registres d'adresse X, Y et Z) | ld R16, X | X <- X-1<br>R16 <- (X) |
| Direct avec incrément après | Adresse sur 2 octets (utilisation registres d'adresse X, Y et Z) | ld R16, X | R16 <- (X)<br>X <- X+1 |

### 6.2. Analyse du programme

Une bonne organisation d'un programme en langage d'assemblage est d'indenter le code suivant quatre colonnes :

- **première colonne :** utilisée pour les étiquettes (ou labels) associant un nom logique à une ligne du programme,

- **deuxième colonne :** utilisée pour indiquer le nom de l'instruction,
- **troisième colonne :** contient la ou les opérandes (séparées par une virgule) de l'instruction,
- **quatrième colonne :** contient un commentaire commençant par le symbole « ; », tout ce qui suit sur la ligne est considéré comme faisant partie du commentaire et ne sera pas utilisé par l'assembleur.

En plus des instructions, un programme en langage d'assemblage peut être composé de directives. Les directives permettent de fournir des indications à l'assembleur pour la production du code machine, par exemple définir des noms (ou macros) associé à une valeur ou un registre, ou définir localisation où sera stockée en mémoire une sous-routine. Ces directives ne seront en aucun cas manipulées par le processeur.

Voici ci-dessous un tableau avec les principales directives de l'assembleur AVR[1]. Lors de l'utilisation d'une directive, celle-ci doit être précédée par « . ».

| Directives | Descriptions |
|---|---|
| byte | Réservation d'un octet à une variable |
| def | Définition d'un nom symbolique |
| device | Définition du microcontrôleur cible |
| dw | Définition d'un mot (16 bits) constant |
| equ | Association d'une valeur à un nom symbole |
| exit | Sortie du fichier |
| include | Lecture du code source d'un fichier |
| macro | Début définition d'une macro |
| endmacro | Fin définition d'une macro |
| org | Définition de l'origine du programme |
| set | Association d'un symbole à une expression |

## 6.3. Compilation et téléversement du programme

Dans cette section, nous allons voir comment compiler le programme précédent et le charger dans la mémoire du microcontrôleur pour être exécuté.

Tout d'abord, utilisez un éditeur de texte pour enregistrer le code du programme dans fichier appelé `prog1.asm`.
Contrairement au TP1, nous n'utiliserons pas l'IDE Arduino pour réaliser des programme assembleur car il n'est pas optimisé pour cela. Nous utiliserons deux programmes en ligne de commande : `avra` et `avrdude`. Le premier programme est l'assembleur pour microcontrôleur de type Atmel AVR, tandis que le second permet de charger le programme exécutable dans la mémoire du microcontrôleur.

1 Directives complètes disponibles au lien suivant : http://www.avr-tutorials.com/sites/default/files/AVR%20Assembler%20User%20Guide.pdf

**Compilation :**
Pour compiler le programme prog1.asm, il suffit d'exécuter la commande suivante dans le répertoire courant où se trouve le programme assembleur.

```
/usr/local/bin/avra prog1.asm
```

Si la compilation se termine correctement, l'assembleur indique en résultat la taille du programme exécutable (nombre de mots de 16 bits). Ce programme a une taille de 6 mots (soit 12 octets), placé en mémoire des adresses 0x0000 à 0x0005.
Le résultat produit par l'assembleur est le fichier exécutable `prog1.hex`, qu'il faudra téléverser ensuite dans la mémoire du microcontrôleur.

**Téléversement sur la carte :**
Pour téléverser le programme exécutable `prog1.hex` dans la mémoire, il faut exécuter la commande suivante :

```
avrdude -p m2560 -c stk500v2 -P /dev/ttyACM0 -F -D -U flash:w:prog1.hex:i
```

Cette commande prend plusieurs paramètres :

- `-p m2560` : indique que le microcontrôleur cible est un ATmega 2560,
- `-c stk500v2` : protocole de communication série utilisé pour charger le programme,
- `-P /dev/ttyACM0` : interface d'accès à la carte Arduino,
- `-F` : force le chargement même si problème de détection du type de microcontrôleur,
- `-D` : désactive effacement automatique de la mémoire avant chargement (réalisé par défaut pour ATmega 2560),
- `-U flash:w :prog1.hex:i` : indique chargement du fichier prog1.hex dans la mémoire flash en accès en écriture.

Si le téléversement a été effectué avec succès, vous devriez voir parmi les messages affichés par cette commande le message suivant :

```
avrdude: safemode: Fuses OK (E:FD, H:D0, L:FF)
```

**Note :** si une erreur se produit débranchez le câble USB du PC puis rebranchez-le et exécutez à nouveau la commande précédente.