TP 1 : MICROCONTRÔLEUR ATMEL ATMEGA 2560

Le but de ce TP sur machine est de se familiariser avec le fonctionnement d'un microcontrôleur de type Atmel Atmega 2560 intégré à une carte Arduino, ainsi que de mettre en œuvre le jeu d'instruction et les modes d'adressage pris en charge par ce microcontrôleur (une documentation détaillée sur les instructions proposées par ce processeur sont en annexe).

1. 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 :

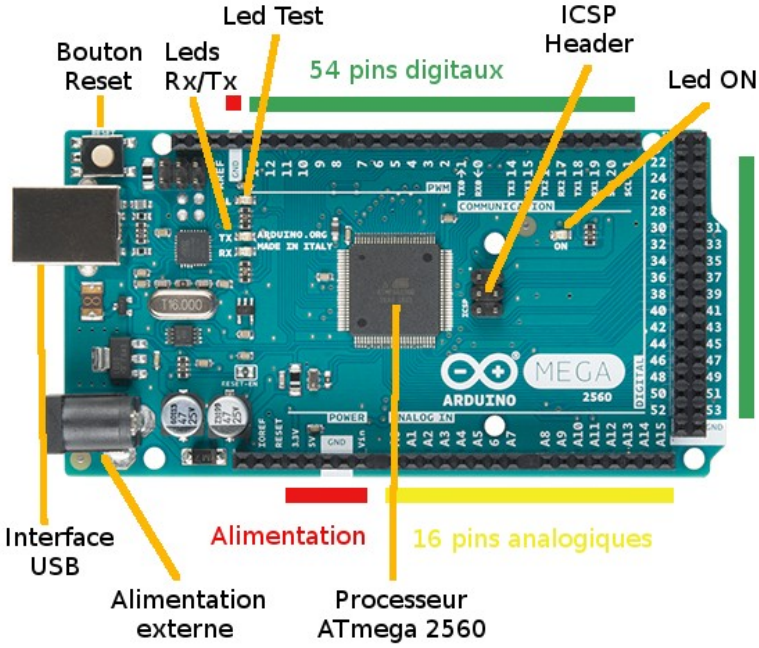


Figure 1: Composants de la carte Arduino MEGA

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 :

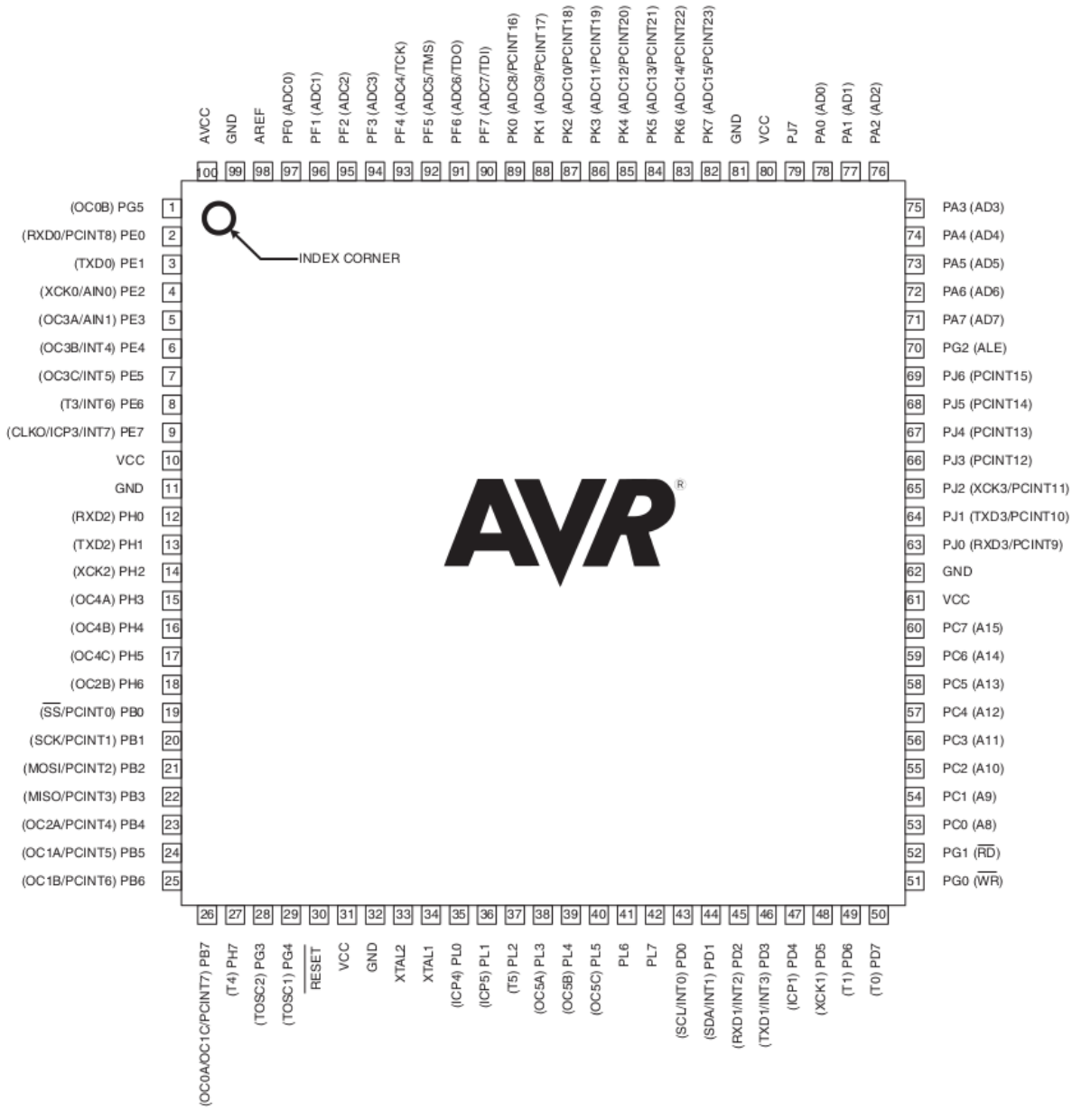


Figure 2 : Connectique de l'ATmega 2560

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

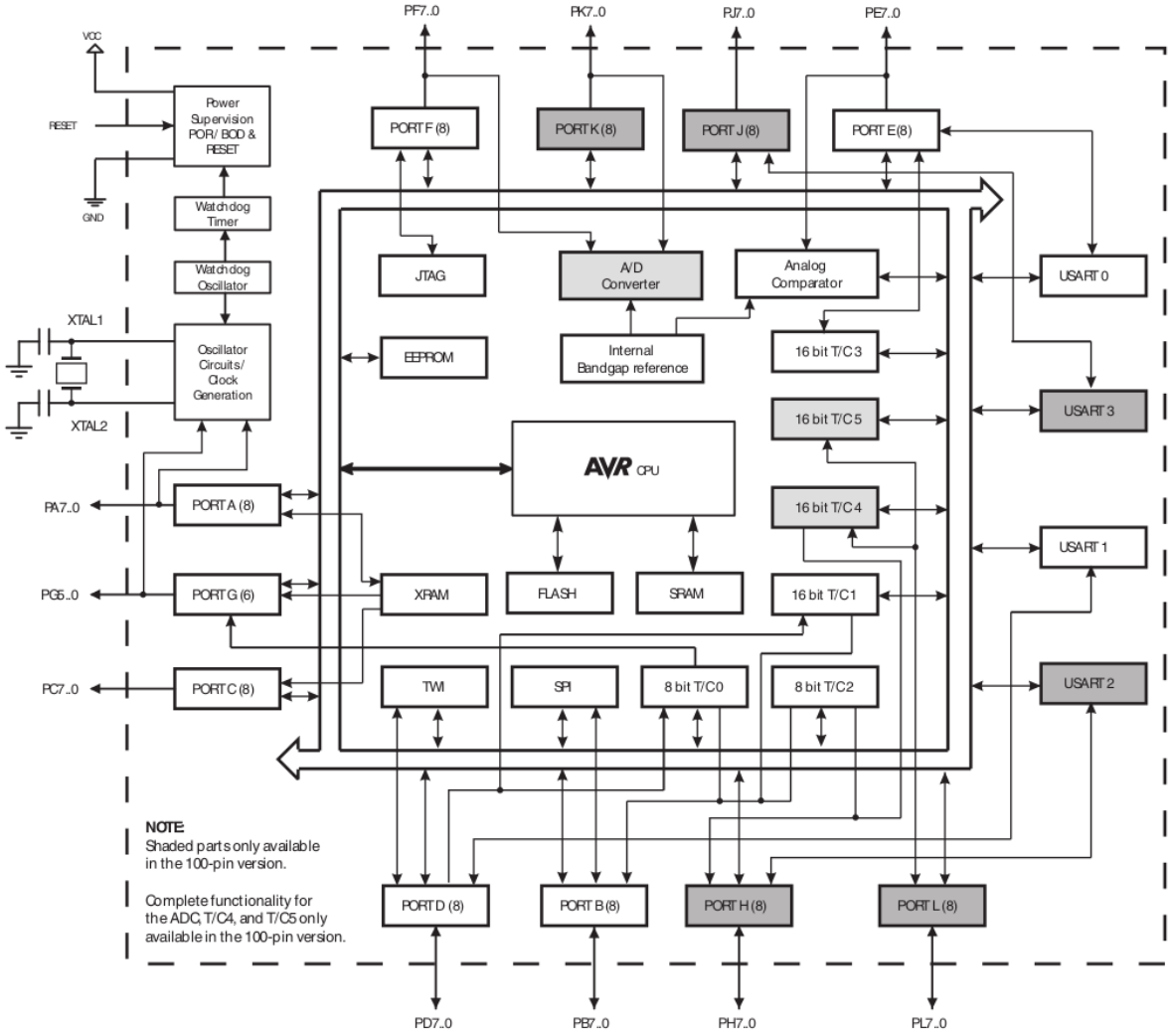


Figure 3 : Schéma interne de l'ATmega 2560

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

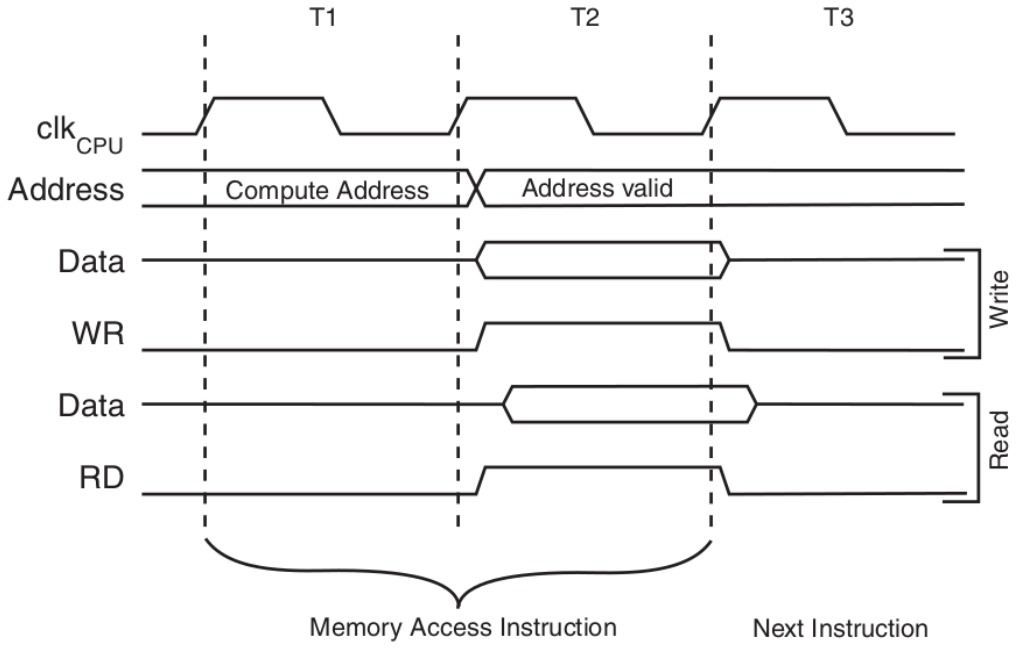
La mémoire flash est une mémoire de type ROM (Read Only Memory) et donc non volatile, c'est-à-dire que les informations sont enregistrées de façon permanente. Cette mémoire a une taille totale de 256 Koctets sur l'ATmega 2560. Les instructions du jeu d'instructions AVR sont toutes de taille 16 ou 32 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 :

Address (HEX)	
0 - 1F	32 Registers
20 - 5F	64 I/O Registers
60 - 1FF	416 External I/O Registers
200 21FF	Internal SRAM (8192 × 8)
2200	External SRAM (0 - 64K × 8)
FFFF	

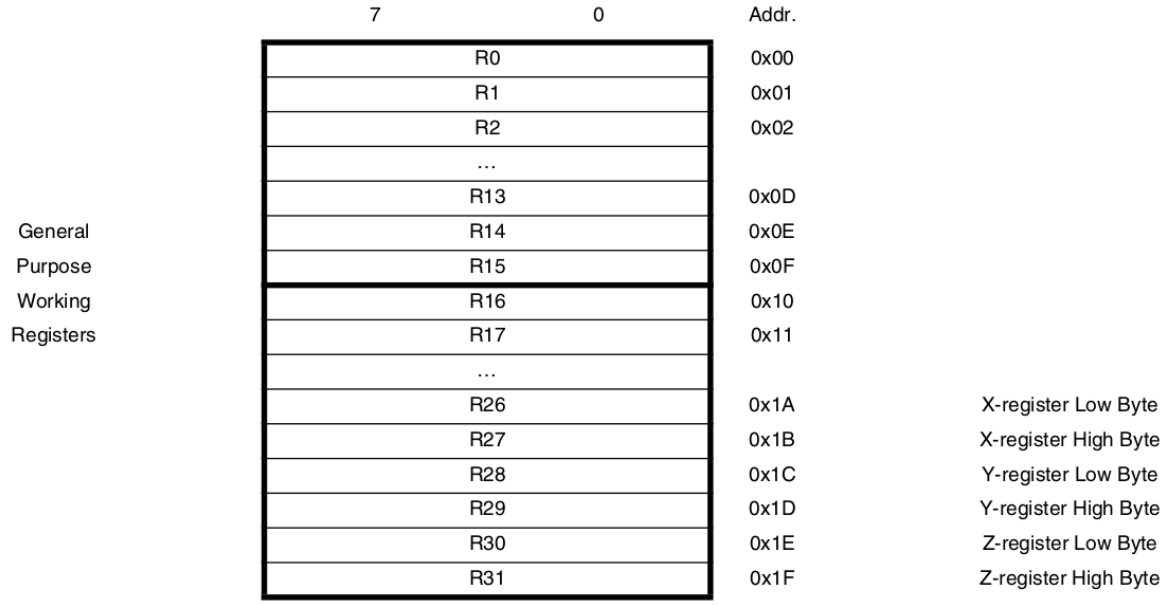
L'accès aux données nécessite 2 cycles d'horloge processeur en lecture ou écriture, comme illustré dans le schéma ci-dessous (3 cycles processeur pour un accès en SRAM externe :



1.2. REGISTRES DE l'ATmega 2560

a) Registres de travail

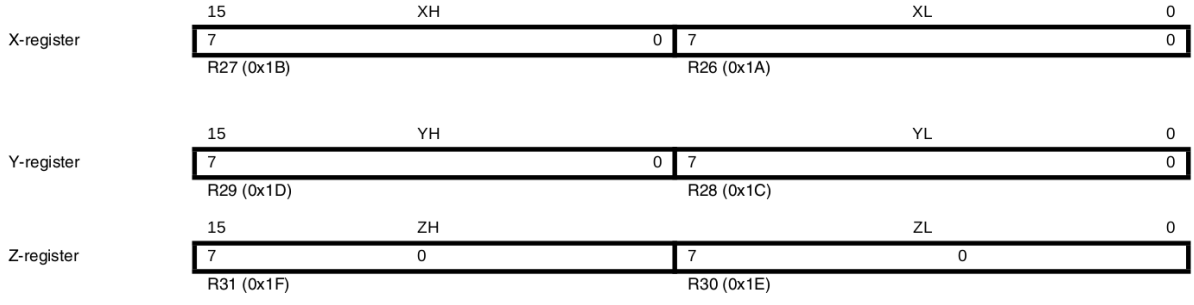
Le microcontrôleur ATmega 2560 utilisé en TP dispose de 32 registres de travail de 8 bits. Ces registres sont utilisables à l'aide de leur nom spécifique de R0 à R31, et sont associés au 32 premiers emplacement de l'espace de données de l'utilisateur. Le schéma ci-dessous indique les adresses en mémoire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

2. 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.

2.1. 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.

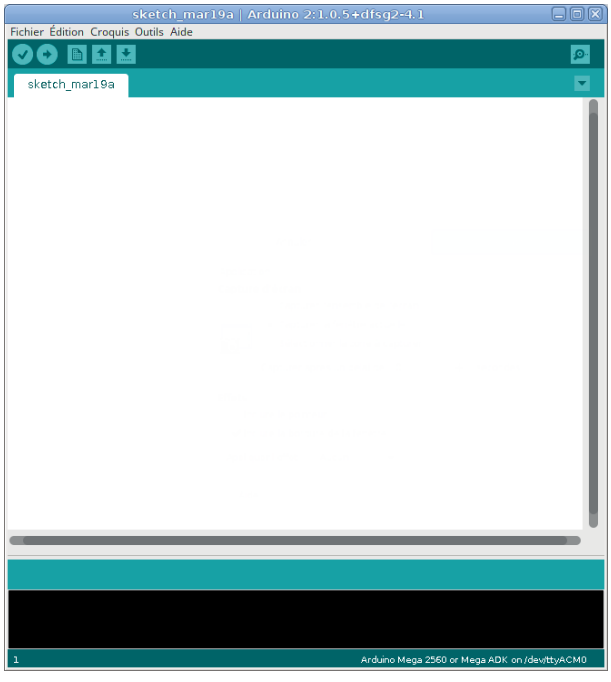
Attention : pour éviter d'endommager la carte Arduino vous ne devez en aucun cas toucher aux composants électronique avec vos doigts, faire attention à l'électricité statique dû au contact avec certains vêtements, et brancher/débrancher avec précaution l'interface USB.

2.2. Environnement de développement

Nous utiliserons l'environnement de développement Arduino IDE fourni par Arduino. Cet IDE peut être lancé soit via le menu des applications soit exécutant la commande (sans taper $>) suivante dans un terminal :

```
$> arduino &
```

Vous devez voir s'afficher l'interface suivante ci-dessous.



A près lancement de l'IDE, la Led TEST doit s'allumer et peut être clignoter. Cela signifie que la carte est prête à être utilisée.

Nous allons indiquer à l'IDE les paramètres nécessaires pour compiler les programmes du TP pour la carte Arduino MEGA. Pour cela, vous devez configurer les trois points suivants :

- Choix de carte : allez dans le menu **Outils -> Type de carte** et sélectionnez **Arduino Mega 2560** ou **Mega ADK**
- Port de connexion : allez dans le menu Outils -> Port série et cochez le port disponible
- Interface ; allez dans le menu **Outils -> Programmateur** et sélectionnez **USBasp**

Une fois ces actions effectuées, l'outil est prêt à compiler et charger sur la carte Arduino MEGA un programme qui sera développé dans ce TP soit en C soit en Assembleur AVR.

2.3. Premier programme

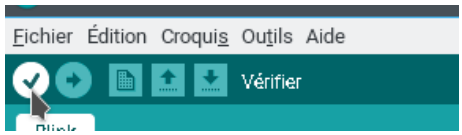
Pour démarrer, nous allons utiliser l'un des programmes d'exemple fournis avec l'IDE pour essayer la carte.
Pour cela, vous devez aller dans le menu **Fichier -> Exemples -> 01. Basics -> Blink**. Vous devez voir apparaître une nouvelle fenêtre contenant le programme suivant :

```
int led = 13;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(1000);
  digitalWrite(led, LOW);
  delay(1000);
}
```

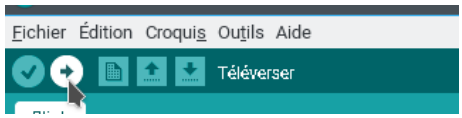
Ce programme a pour but de faire clignoter la Led TEST toutes les secondes. Le première ligne initialise une variable `led` à 13 pour indiquer le numéro de broche sur la quelle est connectée la led au microcontrôleur. La fonction `loop()` est exécutée à l'infini, dont le corps contient une mise au niveau de la Led TEST (état allumé) suivi d'une temporisation de 1000 ms, soit 1s, (avec `delay(1000)`). Après cette attente, l'état de la Led TEST est changé au niveau bas (état éteint) suivi à nouveau d'une temporisation de 1s. Ces quatre instructions seront exécutées à l'infini.
Si cela ne fonctionne pas, il faut appuyer sur le bouton RESET pour exécuter la fonction `setup()`, qui a pour effet d'initialiser la broche associée à la Led TEST en sortie permettant de changer l'état la Led TEST. Cette fonction sera exécutée à chaque appuie sur le bouton RESET.

2.4. Compilation et chargement du programme

Pour compiler le programme, il suffit de cliquer sur le bouton indiqué avec le pointeur de souris dans l'image ci-dessous.



Un cadre avec un fond noir vous donne le résultat de la compilation, s'il n'y a pas d'erreur (message indiquant que la compilation s'est terminée) vous pouvez téléverser le programme binaire dans la mémoire flash de la carte Arduino en cliquant sur le bouton indiqué avec le pointeur de souris dans l'image ci-dessous.



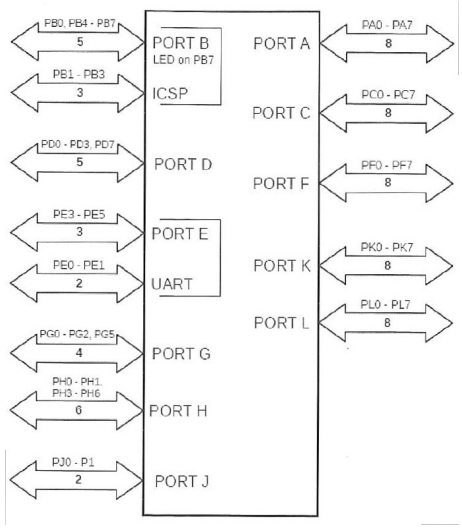
Une fois le téléversement réalisé avec succès, vous devez observer la Led TEST qui se met à clignoter en changeant d'état chaque seconde.

3. Premier programme traduit en C

Dans la section précédente, vous avez utilisé un programme permettant de faire clignoter la Led TEST. Ce programme utilise cependant une surcouche Arduino au C afin de simplifier le développement de programmes pour ce type de cartes. Cette surcouche alourdit le code exécutable généré et peut réduire les performances des programmes.
Nous allons dans cette section, nous abstraire de cette surcouche et écrire ce même programme en langage C natif en faisant appel à quelques librairie nécessaire pour la bonne prise en charge du microcontrôleur.

3.1. Ports d'Entrées-Sorties

Le microcontrôleur Atmega 2560 dispose de 40 pins d'Entrée-Sortes (car 14 sont fixées en sortie) peuvent être programmées soit en Entrée soit en Sortie afin de lire ou envoyer des données respectivement. Ces pins sont regroupées par ensemble d'au plus 8 pins et identifiés à l'aide de lettres, par exemple Port A, Port B, Port C …
L'Atmega 2560 dispose de 11 ports différents chacun composé de 2 à 8 pins comme illustré à la figure ci-dessous :



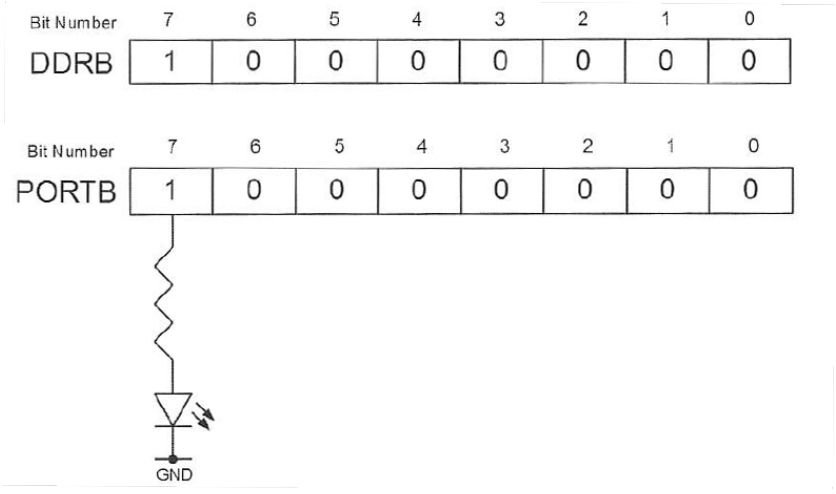
Sur le schéma ci-dessus, on peut noter que 3 pins du Port B (PB1-PB3) sont connectés à l'ICSP Header (voir Figure 1), utilisé pour faire du débogage de programme par les développeurs. De plus, la pin PB7 est associée à la Led TEST nous devrons l'utiliser par la suite pour faire clignoter cette Led. Le Port D ne dispose que de 5 pins qui sont réellement connectées. D'autre part, les pins PE0 et PE1 du Port E sont dédiées à l'USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter) pour la connexion du port série via l'interface USB à l'ordinateur. Cela laisse seulement 3 pins de disponible sur le Port E (PE3-PE5). Enfin, le Port G ne dispose que de 4 pins connectées, le port H que 6 pins de connectées et enfin le Port J uniquement 2 pins.

3.2. Commander la Led TEST

Nous allons maintenant contrôler la Led TEST en changeant son état. Pour cela, nous devons accéder à l'un des ports auquel la Led TEST est connectée. Comme indiqué précédemment, la Led TEST est rattachée à la pin PB7, c'est-à-dire la pin 7 du Port B. La Pin PB7 fait partie des pins d'Entrées-Sorties programmables.

Par défaut, lorsque le carte Arduino MEGA mise sous tension toutes les pins programmables sont configurées en Entrée. Pour pouvoir changer l'état de la Led TEST, nous devons pouvoir écrire l'état désiré sur la pin PB7, c'est-à-dire que nous devons programmer cette pin en sortie. Pour spécifier le sens de communication à utiliser pour les pins d'un port, nous devons modifier la valeur associée à la direction de chaque pin pour le port considéré dans un registre DDR (Data Direction Register) spécifique à chaque port. Le registre DDRA est le registre DDR associé au port A, et cela va de même pour chacun des autres ports, soit DDRB pour le Port B, DDRC pour le PortC …
Ces registres possèdent une taille de 8 bits afin d'indiquer le sens de communication de chacun des bits du port associé. Un bit à 0 dans un registre DDR indique que le bit correspondant du port associé est configuré en Entrée, à l'inverse un bit à 1 configure le bit en Sortie. Ainsi, si l'on place la valeur hexadécimale 0x92 (soit 1001 0010 en binaire) cela indique que les bits PA1, PA4 et PA7 sont à 1 donc configurés en Sortie, alors que les autres bits sont à 0 et donc configurés en Entrée.
La déclaration des registres du microcontrôleur est effectuée en incluant la bibliothèque `avr/io.h`. Cela permet de référencer les registres du microcontrôleur par leur nom et non par leur adresse de correspondance en mémoire SRAM.

Nous avons tous les éléments dont nous avons besoin afin de changer l'état de la Led TEST, qui est reliée à la pin PB7 comme illustré dans le schéma ci-dessous.



Voici les étapes à suivre :

1) Pour cela, nous devons configurer la pin PB7 en sortie en plaçant à 1 le bit 7 du registre DDRB. Cela peut être effectué en affectant la valeur 0x80 au registre DDRB via l'affectation suivante : `DDRB = 0x80`
   Cette configuration est à effectuer la première fois avant d'utiliser le Port B, sauf changement de direction des pins du port. En effet, une fois les directions de communication configurées celles-ci sont conservées durant le fonctionnement de la carte.
2) Ensuite, nous pouvons envoyer l'état de la pin PB7 sur le Port B en affectant la valeur 0x80 au registre associé au Port B comme suit : `PORTB = 0x80`

Soit le programme suivant qui est une traduction du premier programme Arduino présenté à la section précédente :

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
  // Bit 7 registre DDRB mis à 1, PB7 configuré en sortie
  DDRB = 0x80;

  // Boucle infinie changeant l'état de PB7
  // boucle obligatoire, pas de retour du microcontroleur
  while(1)
  {
    PORTB = 0x80;  // place PB7 à 1
    _delay_ms(2000);  // attente de 2 secondes
    _delay_ms(2000);  // attente de 2 secondes
  }

  return 1;
}
```

Note : La fonction `_delay_ms()` permet de faire une pause d'une durée en millisecondes passée en paramètre. Cette fonction nécessite l'utilisation de la librairie `util/delay.h`.

Question 1 : Créez un nouveau projet avec l'IDE Arduino (**Fichier -> Nouveau**) et entrez le code du programme ci-dessus, puis vous compilerez et téléverserez le programme sur la carte Arduino MEGA.
Que constatez vous ? Obtient on le résultat attendu ? Si non, expliquez pourquoi ?

Question 2 : Modifiez le programme ci-dessus pour obtenir le comportement du programme désiré.

4. Programmation du port série

La fonction `printf()` disponible en standard dans le langage C est assez pratique pour afficher de l'information dans un terminal suivant différents formats. Cependant, cette fonction est assez coûteuse en mémoire pour être utilisée avec des systèmes embarqués très contraints en ressource mémoire. Dans cette section, nous allons implémenter l'affichage de chaînes de caractères dans un terminal sans utiliser la fonction `printf()`. Pour cela, nous utiliserons le port série simulé à travers l'interface USB de la carte Arduino.

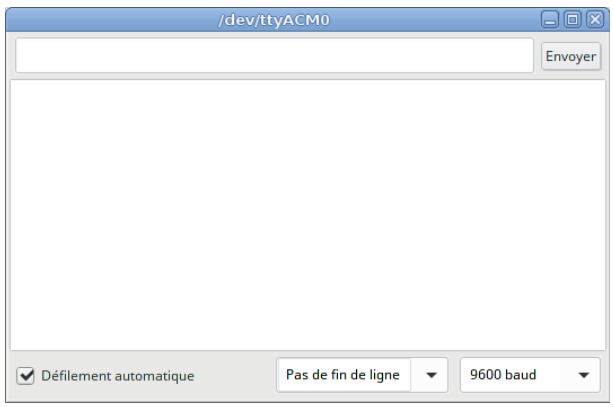
4.1. Initialisation de l'USART

Le composant matériel inclus dans le microcontrôleur ATmega 2560 contrôlant le port série est appelé USART (Universal Synchronous and Asynchronous serial Receiver and Transmitter). L'USART convertie d'une part un octet en un flux série de bits qui sont transmis à travers la liaison série, et d'autre part des données série reçues à travers une liaison série en octets lisibles par le microcontrôleur.
L'USART0 est situé à côté de l'interface USB, ce module permet de simuler un port COM série de type RS-232 avec 9 pins. L'ATmega 2560 dispose de trois autres USARTs (USART1 à USART3).

Pour pouvoir utiliser l'USART0 comme un port série pour échanger de l'information, il faut avant tout configurer ce composant. Pour cela, il est nécessaire d'initialiser deux registres pour indiquer le débit d'échange d'information exprimé en Baud, ainsi que d'autres paramètres nécessaires à la communication comme indiqué ci-dessous :

- **Débit :** 9600 Baud,
- **Taille des données échangées :** 8 bits (1 caractère),
- **Délimiteur :** 1 bit de stop,
- **Test de parité :** pas de parité.

Les paramètres ci-dessus doivent être également utilisés sur le terminal qui permettra d'envoyer et afficher les informations échangées avec la carte Arduino MEGA. Dans la suite, nous utiliserons le terminal série fourni avec l'IDE Arduino qui est déjà paramétré comme indiqué, et accessible en allant dans le menu **Outils -> Moniteur série**. Une nouvelle fenêtre comme illustrée ci-dessous s'affiche.



Plus précisément, les registres suivant seront utilisé pour configurer la communication à travers l'USART0 :

- **Registres UBRR0H et UBRR0L :** indique le débit de communication en Baud,

- **Registre UCSR0B :** Autorise l'utilisation des pins de transmission et réception de l'USART0,
- **Registre UCSR0C :** paramètres de communication de l'USART,
- **Registre UCSR0A :** utilisé pour vérifier si un octet a été reçu et si l'USART est prêt à transmettre,
- **Registre UDR0 :** registre utilisé pour envoyer ou recevoir un octet de donnée.

Voici ci-dessous la fonction permettant d'initialiser l'USART en modifiant les registres listé ci-dessus :

```
#define F_CPU 16000000UL
#define BAUD 9600
#define BAUD_TOL 2

#include <avr/io.h>
#include <util/setbaud.h>

void UartInit(void)
{
  // indique le débit en Baud
  UBRR0H = UBRRH_VALUE;
  UBRR0L = UBRRL_VALUE;

  // activation pins Tx et Rx de l'USART0
  UCSR0B = 0x18;

  // autres paramètres de l'USART0
  // 8-bits, 1 bit de stop, pas de parité, et asynchrone
  UCSR0C = 0x06;
}
```

Dans la fonction ci-dessus, le débit en Baud est tout d'abord placé dans les registres UBRR0H et UBRROL. Pour cela, la librairie `avr` fournit des valeurs `UBRRH_VALUE` et `UBRRL_VALUE` qui sont automatiquement calculées à partir de la fréquence CPU `F_CPU`, et le débit en Baud `BAUD` et `BAUD_TOL`. Ensuite, les bits 3 (TXEN0) et 4 (RXEN0) du registre UCSR0B sont positionnés à 1. Enfin, le registre UCSR0C est modifié pour indiquer les derniers paramètres de la communication. Les bits 7 (UMSEL01) et 6 (UMSEL00) sont à 0 pour indiquer une communication asynchrone, les bits 5 (UMP01) et 4 (UMP00) sont mis à 0 pour indiquer qu'il n'y a pas de parité, enfin les bits 3 (USBS0) et 0 (UCPOL0) SONT à 1 et les bits 2 (UCSZ01) et 1 (UCSZ00) sont à 1 pour indiquer l'échange de 8 bits à la fois.

4.2. Transmission et Réception de caractères

Après avoir vu l'initialisation de l'USART, nous allons nous intéresser à l'envoi et la réception d'un caractère à travers le port série grâce à l'USART.

La fonction ci-dessous permet la transmission d'un caractère vers la carte Arduino.

```
void UartTxByte(char data)
{
  while(!(UCSR0A & 0x20));  // attente si non prêt en Tx

  UDR0 = data;    // envoi caractère
```

```
}
```

La fonction ci-dessus prend en entrée un caractère à transmettre à la carte Arduino. Avant de pouvoir transmettre un caractère, nous devons nous assurer que l'USART est prête pour la transmission du caractère. Pour cela, nous devons tester la valeur du bit UDRE0 du registre UCSR0A, s'il est à 1 alors l'USART est prêt sinon l'USART est occupée. Pour tester la valeur de ce bit, on compare la valeur de ce bit avec celui de la chaîne binaire 0010 0000 à l'aide un ET logique. Tant que les valeurs sont différentes (bit UDRE0 différent de 1) alors il faut rester dans la boucle while, d'où l'utilisation de l'opérateur de négation « ! ».
Ensuite, le caractère à transmettre à l'USART est placé dans le registre UDR0.

La fonction ci-dessous permet la réception d'un caractère depuis la carte Arduino.

```
int UartRxByte(void)
{
    int data;

    if(UCSR0A & 0x80)  // test si caractère reçu
    {
        data = UDR0;  // réception caractère
    }
    else
    {
        data =-1;  // aucun caractère reçu
    }

    return data;
}
```

La fonction ci-dessus lit un caractère depuis l'USART si un caractère est disponible. Cela est réalisé en comparant la valeur du bit RXC0 du registre UCSR0A à l'aide d'un ET logique. Si le bit est à 1, alors la comparaison retournera Vrai et le contenu du registre UDR0 sera lu et retourné. Dans le cas contraire, une valeur égale à -1 est retournée par la fonction indiquant qu'aucune nouvelle donnée n'a été récupérée.

Question 3 : Développez un programme C permettant d'envoyer des caractères à l'USART à l'infini et de les afficher sur le terminal de l'IDE Arduino.

5. Annexe : 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