TP 2 : MICROCONTROLEUR ATMEL ATMEGA 2560

Le but de ce TP sur machine est de se familiariser avec le langage d'assemblage proposé par le microcontrôleur de type Atmel ATmega 2560 intégré à une carte Arduino, afin de se familiariser avec le jeu d'instruction et les modes d'adressage pris en charge par ce microcontrôleur.

On fournit en Annexes les éléments suivants :

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

1. 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.

1.1. 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
		ldi R16, 0x1A
		ldi R16, 0b00011010
			Valeur décimale
			Valeur hexadécimale
			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 R16 <- (X)
Direct avec incrément après	Adresse sur 2 octets (utilisation registres d'adresse X, Y et Z)	ld R16, X	R16 <- (X) X <- X+1

1.2. 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

Soit le programme suivant décrit en langage d'assemblage :

```
.org 0x00

     clr r16
     ldi r17, 15
```

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

```
debut:
      add r16, r17
      dec r17
      mov r18, r16
      rjmp debut
```

On peut noter que dans le programme ci-dessus la directive `.org 0x00` indique à l'assembleur que le code qui suit doit être stocké à l'adresse 0 dans la mémoire RAM du microcontrôleur.

Question 1 : En vous aidant de la documentation du jeu d'instruction fournit en annexe, indiquez ce que réalise chaque instruction du programme ci-dessus, et ce que fait globalement ce programme.

1.3. 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.

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.

1.4. Branchements

Le jeu d'instructions fournit par l'ATmega 2560 dispose de plusieurs branchements conditionnels.

Question 2 : En vous aidant du jeu d'instructions fourni en annexes, donnez les instructions de branchement conditionnelles correspondant aux instructions suivantes vues en cours : JMPO, JMPC, JMPP et JMPN.

Question 3 : En vous aidant du jeu d'instructions fourni en annexes, donnez des exemples de branchements conditionnelles non considérés dans le langage d'assemblage considéré en cours.

2. Temps d'exécution d'un programme

Nous allons nous intéresser dans cette section au calcul du temps d'exécution d'un programme ne langage d'assemblage.

La durée d'exécution d'un programme dépend de la fréquence d'horloge du processeur et du nombre total de cycles d'horloge processeur correspondant à toutes les instructions du programme à exécuter.

2.1. Durée d'un cycle d'horloge processeur

La durée d'un cycle d'horloge processeur dépend de la fréquence de l'horloge de fonctionnement du processeur. Plus précisément, nous devons calculer la durée d'une période d'horloge **T** obtenue par la formule suivante :

P = 1 / **F**

où **F** est la fréquence d'horloge du processeur.

Question 4 : Calculez la durée d'un cycle d'horloge processeur de l'ATmega 2560, vous pourrez vous aider des caractéristiques de l'ATmega 2560 fournies en annexes.

2.2. Temps d'exécution d'un programme simple

Le temps d'exécution total **T** d'un programme assembleur est déterminé par la somme du nombre de cycle d'horloge des instructions du programme multiplié par la durée d'un cycle d'horloge processeur, comme donné dans la formule suivante :

T = C * P

où **C** est le nombre total de cycles d'horloge processeur du programme et **P** la durée d'un cycle d'horloge.
Soit le programme assembleur suivant :

```
LDI   R16, 0x0F
LDI   R17, 7
ADD   R17, R16
MOV   R20, R17
```

Question 5 : Donnez le temps d'exécution total du programme ci-dessus. Vous pourrez vous aider du détail sur le jeu d'instructions de l'ATmega 2560 fourni en annexes.

2.3. Temps d'exécution d'un programme avec boucle

Nous allons cette fois-ci nous intéresser au calcul du temps d'exécution d'un programme assembleur contenant une boucle. La même approche pourra être généralisée pour des programmes assembleur contenant plusieurs boucles.

Soit le programme assembleur suivant contenant une boucle :

```
    LDI   R16, 5
Boucle: DEC R16
    NOP
    BRNE  Boucle
    NOP
    NOP
```

Question 6 : Donnez le temps d'exécution total du programme ci-dessus en adaptant la formule précédente. Vous pourrez vous aider du détail sur le jeu d'instructions de l'ATmega 2560 fourni en annexes.
Note : L'instruction BRNE est particulière dans le sens où elle n'est pas exécutée avec un nombre de cycles d'horloge constant. Son exécution nécessite **1 cycle d'horloge** si la condition est évaluée à **faux**, et **2 cycles d'horloge** si la condition est **vrai**.

Soit le programme ci-dessous contenant deux boucles imbriquées :

```
        LDI   R16, 5
B_ext:  LDI   R17, 4
B_int:  DEC   R17
        NOP
```

```
BRNE  B_int
DEC   R16
NOP
BRNE  B_ext
NOP
NOP
```

Question 7 : Donnez le temps d'exécution total du programme ci-dessus en adaptant la formule précédente. Vous pourrez vous aider du détail sur le jeu d'instructions de l'ATmega 2560 fourni en annexes.

3. Manipulation de la pile

Dans cette section, nous allons nous intéresser au fonctionnement de la pile en mémoire et des instructions pour manipuler cette structure de données.

On rappelle qu'une pile est une zone mémoire allouée et gérée par le programmeur de façon spécifique par rapport au reste de la mémoire, afin de maintenir un ensemble de données consécutives en mémoire. Ces données sont temporaires et correspondent à des variables locales ou des adresses de retour après le traitement d'une interruption ou d'appel de fonction. L'accès des données dans une pile est réalisé suivant un ordre LIFO (Last In First Out).

L'accès à une pile est réalisé à l'aide d'un pointeur de pile indiquant l'adresse pour ajouter un nouvel élément au sommet de la pile. Sur les microcontrôleur AVR, ce pointeur de pile est stocké dans le registre de pile **SP**, qui est implémenté à l'aide de 2 registres de 8 bits **SPL (Stack Pointer Low)** et **SPH (Stack Pointer High)** 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	

La valeur initiale de ce registre est la plus grande adresse de la mémoire SRAM, soit 0xFF21 comme indiqué dans schéma ci-dessus extrait de la documentation de l'ATmega 2560. Afin de simplifier l'initialisation du registre de pile, une macro nommée RAMEND égal à la plus grande adresse de la mémoire SRAM a été défini pour chaque microcontrôleur. En effet, d'un microcontrôleur à l'autre RAMEND prend des valeurs différentes suivant la taille de la SRAM. Comme cette adresse est sur 16 bits, il est nécessaire de récupérer les 8 bits inférieurs et les 8 bits supérieurs pour les affecter au registre **SPL** et **SPH** respectivement. Cela peut être réalisé avec les fonctions low() et high(). Pour pouvoir utiliser la macro RMAEND, vous devez ajouter la ligne suivante en début de votre programme pour inclure une bibliothèque contenant la spécification du microcontrôleur ATmega 2560.

```
.include "m2560def.inc"
```

3.1. Initialisation du registre SP

L'initialisation du registre de pile **SP** est à la charge du programmeur. Cela peut être réalisé à l'aide des quelques lignes indiquées ci-dessous :

```
LDI   R16, low(RAMEND)
OUT   SPL, R16
LDI   R16, high(RAMEND)
OUT   SPH, R16
```

Question 8 : Expliquez pourquoi on utilise l'instruction OUT pour charger une valeur dans les registres SPL et SPH. Que peut-on déduire de la localisation des adresses 0x3D et 0x3E en mémoire SRAM.

3.2. Opérations sur la pile

La pile est manipulée grâce aux instructions PUSH et POP. L'instruction PUSH permet d'ajouter une donnée (sur 8 bits) au sommet de la pile indiqué par le registre **SP** (décrémentant ensuite l'adresse du registre **SP**), alors que l'instruction POP permet de récupérer la donnée qui au sommet de la pile en utilisant le registre SP (incrémentant d'abord l'adresse du registre **SP**). Le contenu du registre de pile **SP** est incrémenté/décrémenté de 1 lorsqu'une donnée est empilée/dépilée et de 3 lorsqu'il s'agit d'une adresse.

Question 9 : En vous aidant du jeu d'instruction disponible en annexes, donnez un programme qui empile d'abord la donnée 0xF3, suivie de 0x15, puis dépile le sommet de la pile et finit par empiler la donnée 0x0B. Vous n'oublierez pas d'initialiser la pile et de finir votre programme par une boucle infinie.
Indiquez à chaque étape le contenu de la pile et l'adresse contenu dans le registre de pile **SP**.

4. Création de sous-routines

Dans cette section, nous allons voir comment définir une sous-routine (concept voisin de la fonction) et y faire appel dans un programme assembleur.

Une *sous-routine* regroupe un bloc d'instructions qui est appelé de façon répétitive, cela permet d'alléger l'écriture des programmes et de réduire leur taille. Les sous-routines sont manipulées grâce à deux instructions spécifiques :

1. L'instruction RCALL ou CALL prend en paramètre l'étiquette de la première instruction de la sous-routine à appeler. Avant d'aller à l'instruction d'étiquette spécifiée en paramètre, cette instruction sauvegarde au préalable l'adresse du compteur ordinal **CP** au sommet de la pile pour savoir quelle est l'adresse de l'instruction à exécuter au retour de la sous-routine. Cela implique qu'avant de faire appel à la première sous-routine il est obligatoire d'initialiser le registre de pile **SP**.
2. L'instruction RET ne prend aucun paramètre et permet d'indiquer au microcontrôleur que la fin d'une sous-routine a été atteinte. Lorsque le microcontrôleur rencontre cette instruction, il dépile l'adresse au sommet de la pile et charge celle-ci dans le compteur ordinal **CP**.

Attention : Deux principales différences caractérisent une sous-routine d'une fonction dans un langage haut niveau :

- une sous-routine ne prend aucun paramètre en entrée et ne retourne aucune valeur,
- il n'y a pas de notion de contexte (ou environnement) d'exécution propre à la sous-routine, car tous les registres sont accessibles durant l'exécution d'une sous-routine. Dans le cas où une ou plusieurs valeurs stockées dans des registres pourraient être écrasées le contenu de ces registres doivent être sauvegardés dans la pile avant l'appel à la sous-routine.

Soit le programme suivant calculant le carré des 5 premiers entiers :

```
    CLR R16   ; R16 initialisé à 0, contiendra résultat final
    LDI R17, 5        ; carré des 5 premiers nombres à sommer

Bou: MOV R18, R17     ; copie valeur compteur de boucle
Carre: ADD R16, R17   ; ajoute à R16 valeur courante dans R17
    DEC R18           ; décrémente de 1 compteur boucle
    BRNE Carre        ; saut à Carre si R18!=0
    DEC R17           ; soustrait 1 à R17 pour calcul carré
    BRNE Bou          ; saut à Bou si R17!=0
Fin: rjmp fin         ; fin programme
```

Question 10 : Modifiez le programme ci-dessus de sorte à créer une fonction qui calcul le carré d'un entier dont la valeur est indiquée dans le registre R17.

5. Clignotement de la Led TEST en assembleur

Dans cette section, nous allons reproduire le programme vu au précédent TP faisant clignoter la Led TEST, mais en utilisant le langage d'assemblage de l'ATmega 2560. Pour cela, nous allons réaliser les divers morceaux du programme avant de les assembler.

5.1. Fonction de temporisation

Nous allons dans un premier temps réaliser une fonction de temporisation pour effectuer l'alternance de l'état de la Led de TEST.

Soit les instructions suivantes :

```
        LDI R24 low(800)
        LDI R25, high(800)
    Bou: SBIW R24, 1
        NOP
        BRNE Bou
```

Question 11 : Indiquez ce que réalise chacune de ces instructions, puis l'ensemble de ces instructions, en vous aidant du jeu d'instructions en annexes.

Question 12 : Déterminez le nombre total de cycle d'horloge nécessaire pour exécuter ces instructions.

Soit les instructions suivantes :

```
    LDI R26, low(4000)
    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
```

Question 13 : Indiquez ce que réalise chacune de ces instructions, puis l'ensemble de ces instructions, en vous aidant du jeu d'instructions en annexes.

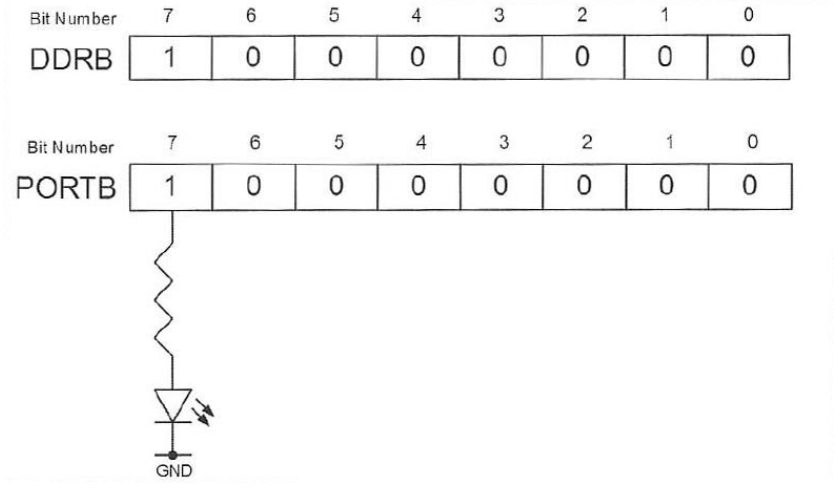
Question 14 : Déterminez le nombre total de cycle d'horloge nécessaire pour exécuter toutes ces instructions, puis le temps d'exécution.

5.2. Changement d'état de la Led de TEST

Nous allons nous intéresser aux instructions nécessaires pour changer l'état de la Led de TEST.

Comme vu au TP précédent, la Led de TEST est connectée à la pin PB7 du PORTB comme illustré dans le schéma ci-dessous.
On rappel que pour changer l'état de la Led de TEST nous devons effectuer les deux étapes suivantes :

1) Il faut configurer la pin PB7 en sortie en plaçant à 1 le bit 7 du registre DDRB, c'est-à-dire en affectant la valeur 0x80 au registre DDRB .
2) Ensuite, nous pouvons envoyer l'état de la pin PB7 sur le PORTB en affectant la valeur 0x80 au registre associé au PORTB .



Question 15 : Donnez les instructions permettant de placer les valeurs nécessaires dans les registres DDRB et PORTB. On précise que ces noms correspondent à des macros qui utilisables, de plus ces registres sont des registres d'Entrées-Sorties.

5.3. Assemblage des morceaux

En se basant sur les morceaux de programme vu précédemment, nous allons pouvoir construire le programme assembleur permettant de faire clignoter la Led de TEST.

Le programme à réaliser est composé des étapes suivantes :

- initialisation de la pile
- configuration en sortie de la Pin 7 du PORTB
- création d'une sous-routine appelée Delay réalisant une attente de 1s
- dans une boucle infinie
  - éteindre la Led de TEST
  - faire appel à la sous-routine Delay
  - allumer la Led de TEST
  - faire appel à la sous-routine Delay

Question 16 : Ecrivez le programme assembleur permettant de faire clignoter la Led de TEST en suivant les étapes données ci-dessus. Vous assemblerez votre programme et le chargerez dans la mémoire de l'ATmega 2560 pour vérifier le bon fonctionnement de votre programme.

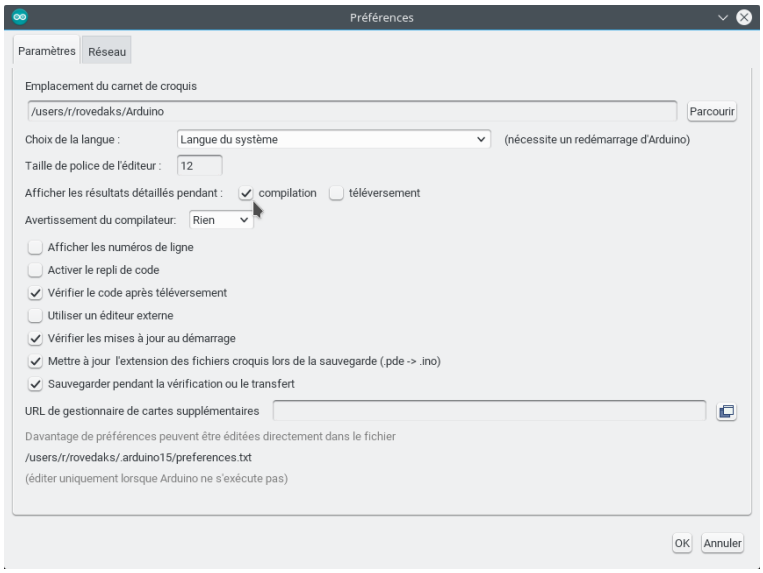
6. Visualiser le code d'assemblage d'un programme

Avant de regarder de plus près le langage d'assemblage de l'ATmega 2560, nous allons visualiser le code qui a été généré par l'IDE Arduino pour l'exemple de programme que nous avons vu au TP 1 et faisant clignoter la Led TEST toutes les secondes.

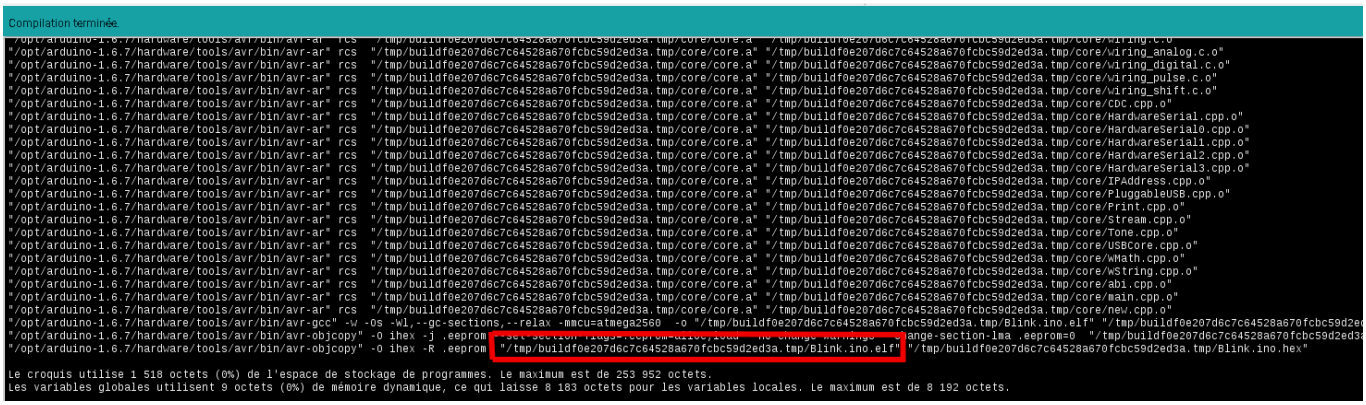
On rappelle que ce programme est accessible avec l'IDE en allant dans le menu **Fichier -> Exemples -> 01. Basics -> Blink**. Vous devez voir apparaître une nouvelle fenêtre contenant le programme suivant :

```
int led = 13;

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

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

Pour pouvoir visualiser le code d'assemblage généré à la compilation du programme ci-dessus, nous devons changer les préférences de l'IDE afin de voir l'emplacement où le fichier exécutable a été généré.
Pour cela, vous devez aller dans le menu **Fichier -> Préférences** et activer l'option « compilation » pour le champ « Afficher les résultats détaillés pendant : », comme indiqué dans l'image ci-dessous.



Ensuite cliquez sur le bouton OK et compilez le programme. Lors de la compilation, l'IDE vous indiquera le répertoire où il a généré le fichier exécutable au format .elf, comme indiqué dans l'image suivante.



Nous allons utiliser l'outil `avr-objdump` est un dé-assembleur qui permet d'extraire le code en langage d'assemblage du programme compilé. Cet outil est normalement rangé dans le répertoire : `/opt/arduino-1.6.7/hardware/tools/avr/bin/`

Tapez la commande suivante (sur une même ligne) pour afficher le code d'assemblage extrait du fichier elf :

```
/opt/arduino-1.6.7/hardware/tools/avr/bin/avr-objdump -C
-d <rep_fichier_elf>/Blink.ino.elf
```

où `<rep_fichier_elf>` est le répertoire contenant le fichier exécutable `Blink.ino.elf` généré par l'IDE Arduino.

Question 17 : Sauvegardez le résultat dans un fichier et éditez-le avec un éditeur de texte. Quels commentaires pouvez-vous faire sur le code associé au bloc suivant :

```
0000023a <loop>:
```

ANNEXES

7. 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

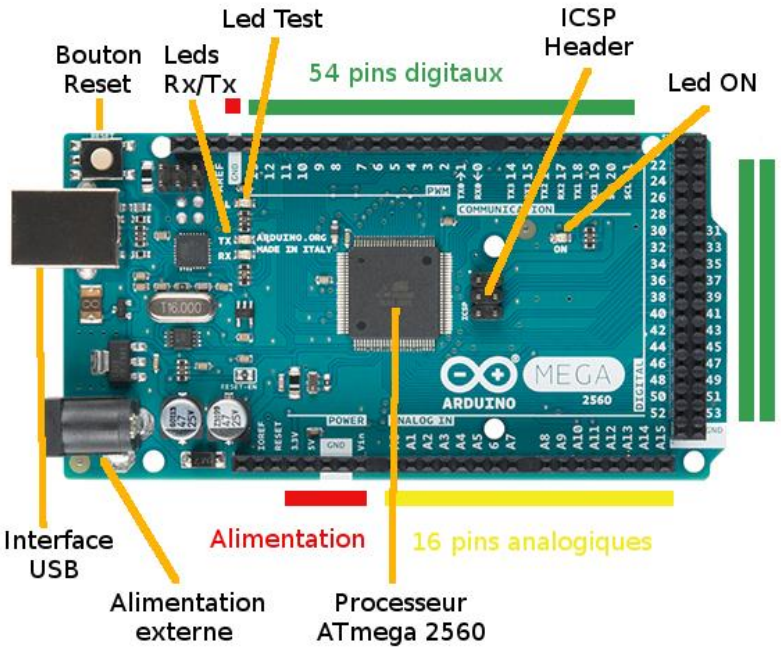
8. Matériel utilisé

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 :



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 :

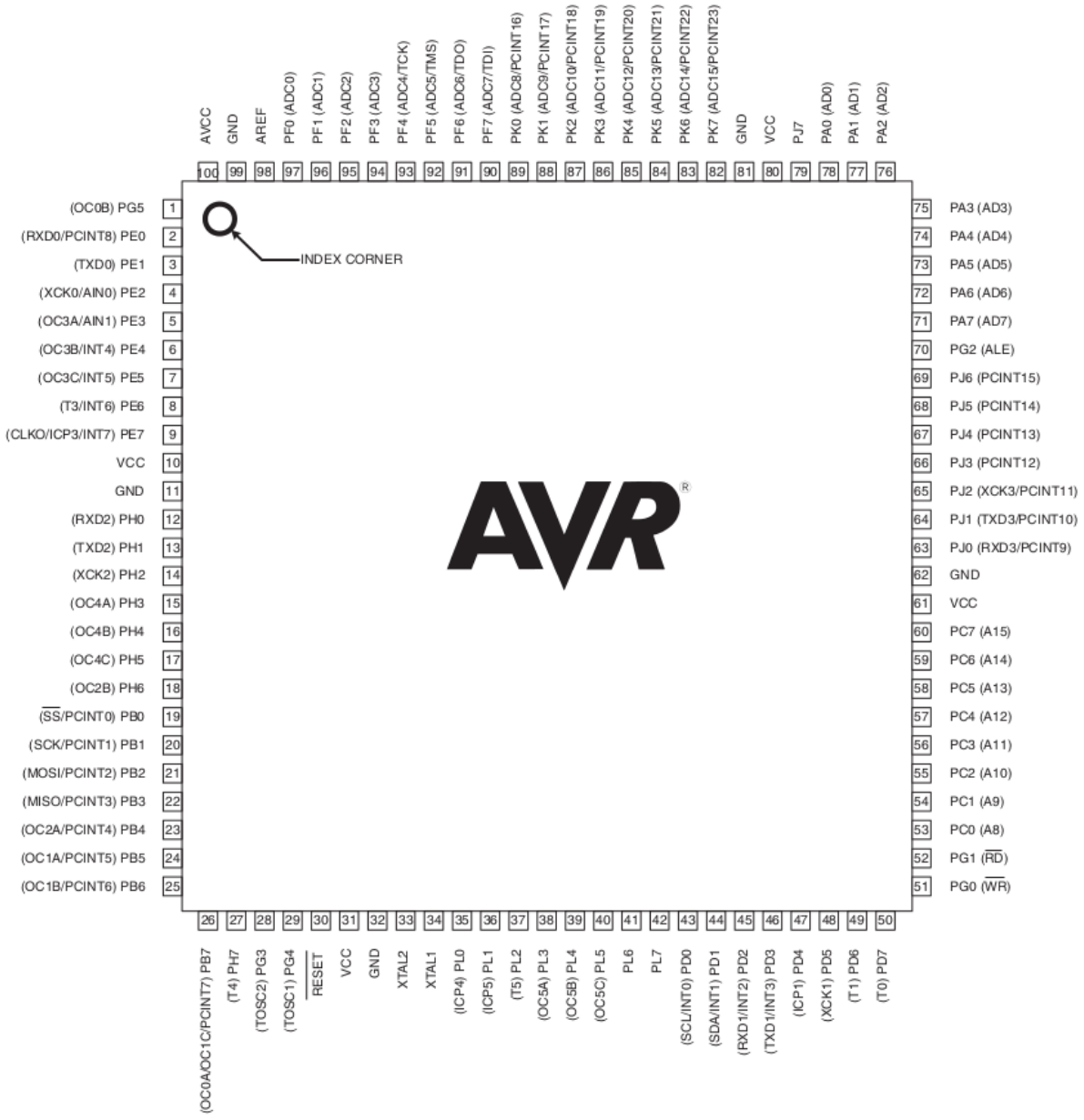


Figure 2 : Connectique de l'ATmega 2560

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

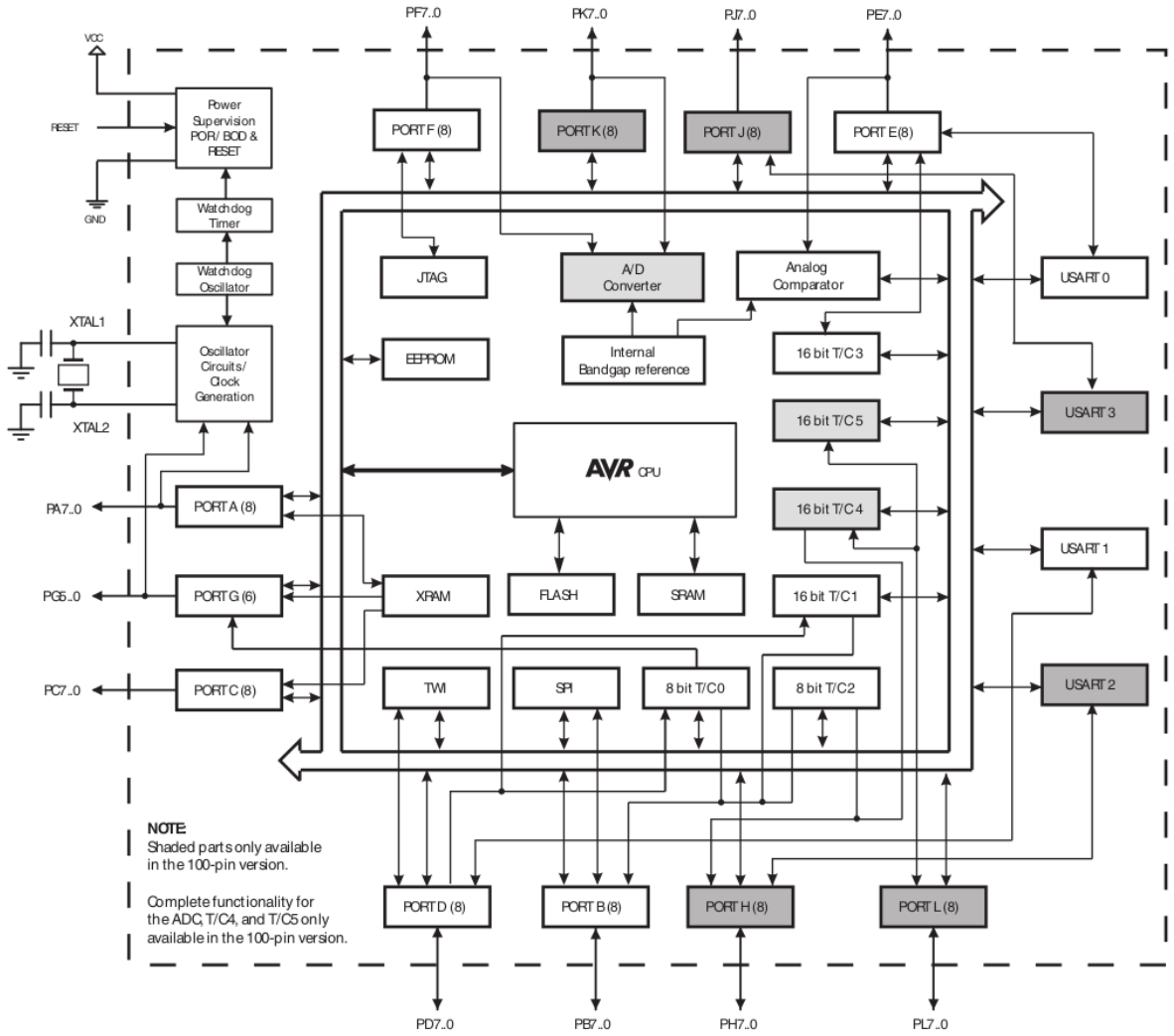


Figure 3 : Schéma interne de l'ATmega 2560

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.

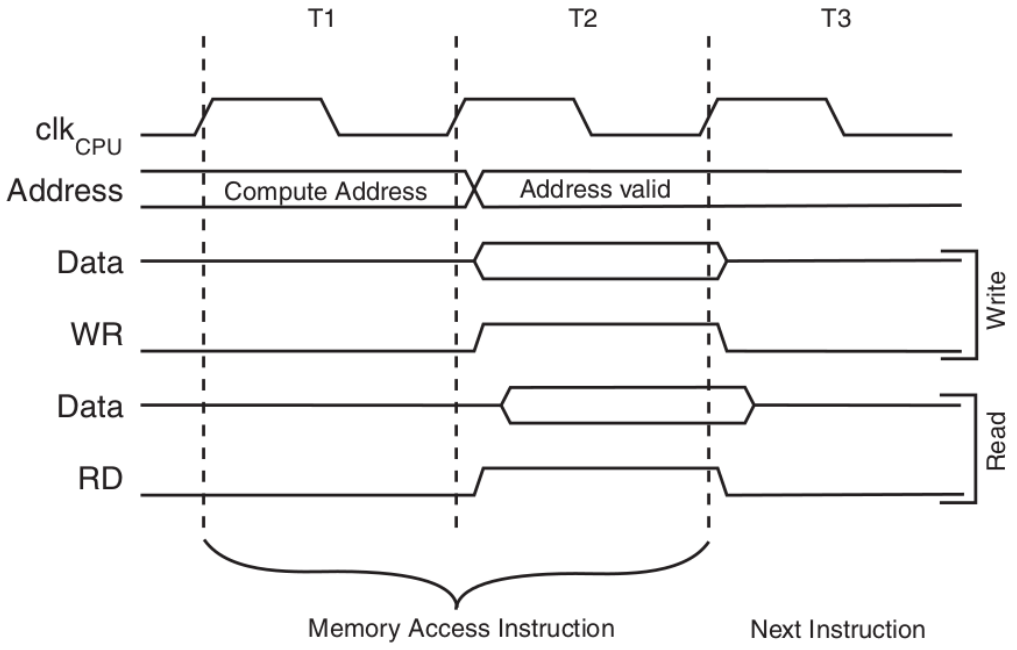
b) Mémoire SRAM

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	Internal SRAM (8192 × 8)
21FF	
2200	External SRAM (0 - 64K × 8)
FFFF	

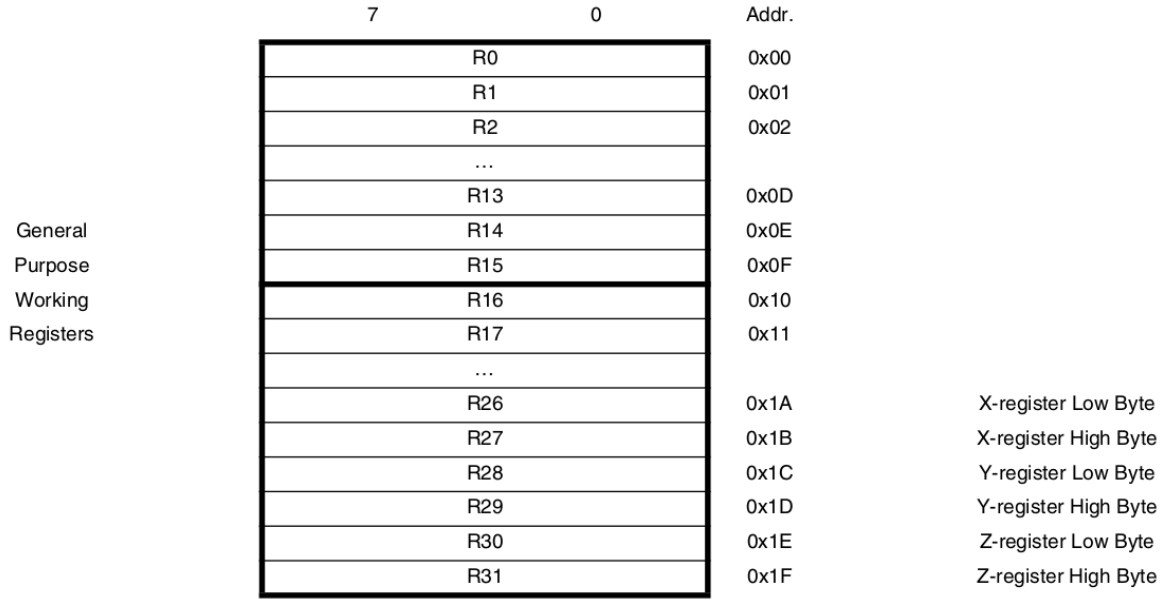
L'accès aux données nécessite 2 cycles d'horloge processeur en lecture ou écriture, comme illustré dans le schéma ci-dessous (3 cycles processeur pour un accès en SRAM externe :



8.2. REGISTRES DE l'ATmega 2560

a) Registres de travail

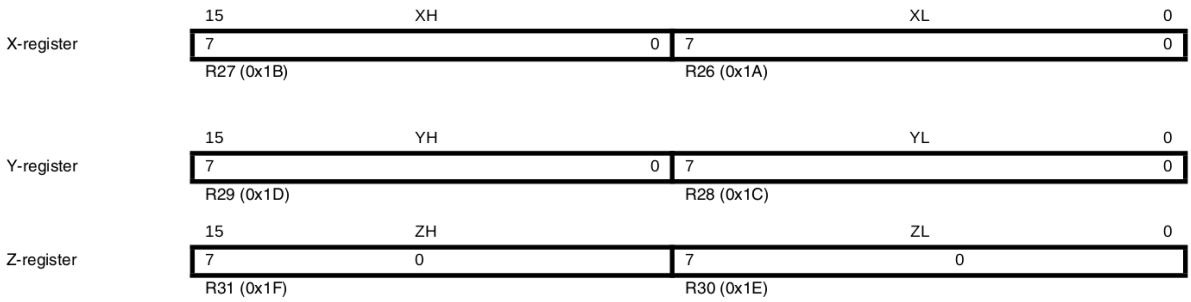
Le microcontrôleur ATmega 2560 utilisé en TP dispose de 32 registres de travail de 8 bits. Ces registres sont utilisables à l'aide de leur nom spécifique de R0 à R31, et sont associés au 32 premiers emplacement de l'espace de données de l'utilisateur. Le schéma ci-dessous indique les adresses en mémoire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

9. 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 paramétrages près.

9.1. 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.

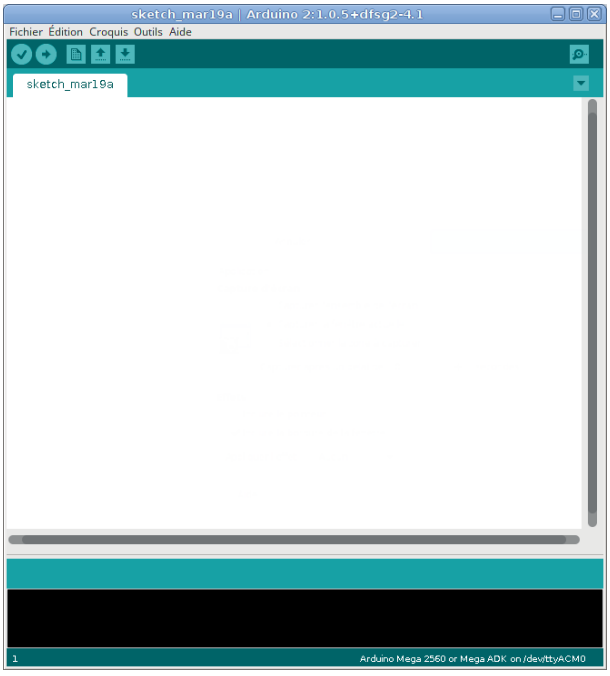
Attention : pour éviter d'endommager la carte Arduino vous ne devez en aucun cas toucher aux composants électroniques avec vos doigts, faire attention à l'électricité statique dû au contact avec certains vêtements, et brancher/débrancher avec précaution l'interface USB.

9.2. 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.

9.3. Premier programme

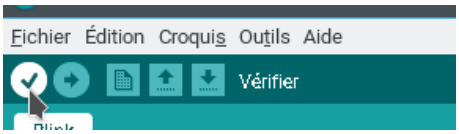
Pour démarrer, nous allons utiliser l'un des programmes d'exemple fournis avec l'IDE pour essayer la carte.
Pour cela, vous devez aller dans le menu **Fichier -> Exemples -> 01. Basics -> Blink**. Vous devez voir apparaître une nouvelle fenêtre contenant le programme suivant :

```
int led = 13;

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

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

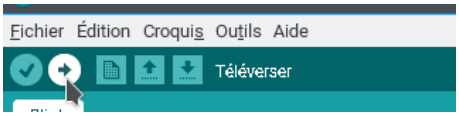
Ce programme a pour but de faire clignoter la Led TEST toutes les secondes. Le première ligne initialise une variable `led` à 13 pour indiquer le numéro de broche sur la quelle est connectée la led au microcontrôleur. La fonction `loop()` est exécutée à l'infini, dont le corps contient une mise au niveau 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.

9.4. 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.


Fichier Édition Croquis Outils Aide
Téléverser