Cours Assembleur ATMEGA32

Introduction

Ce document est la suite du document Assembleur Microcontrôleur **ATMEL ATMEGA** que j'ai déjà réalisé.

Vous trouverez dans ce document plusieurs chapitres sur l'initiation au microcontrôleur **ATMEGA** et sur l'utilisation des instructions assembleurs.

Un rappel sommaire des opérations logiques (**ET**, **OU**, **NON**, **OU Exclusif**) et la conversion décimale hexadécimale et binaire sera aborder.

Le fonctionnement des indicateurs du registre de statut **SREG** avec le positionnement en fonction des opérations arithmétiques et logiques.

Et enfin l'apprentissage de l'assembleur en lui-même avec des exemples commentés et des références aux deux autres documents déjà 'francisée'.

Le mode d'adressage et les opérations de saut et sous-programme avec exemple.

La présentation du programme d'assemblage AVR est vue en dernier pour faire le teste des exemples.

Bien entendu, vous pouvez me signaler les erreurs éventuelles qui se seraient glissées dans ce document à 'l'issus de mon plein gré !'.

Mon mail à changé pour cause de pollution, c'est maintenant : Balade.nono@voila.fr.

Très bonne lecture à vous ! Jean-Noël, en mai 2005 Salvador de Bahia, Brésil.

Le Plan Mémoire

Le cœur **AVR** combine 32 registres spéciaux travaillants directement avec l'**U**nité **A**rithmétique de **L**ogique **ALU**, qui représente le registre d'accumulateur **A** (**B** ou **D**) dans les microcontrôleurs classiques. L'accumulateur est le cœur du système, c'est lui qui s'occupe des opérations de calcul et de comparaison logique.

Ces registres doivent être considérés par l'utilisateur comme des variables mémoires d'un octet (8 bits) pouvant prendre une valeur comprise entre 0 et 255. Ces variables sont notées 'rxx' ou xx prend les valeurs 00 à 31, donc 'r0' à r31'. La variable par défaut est la variable 'r16' qui peut être utilisé par toutes les instructions.

Ces registres spéciaux sont en accès direct avec le cœur du microcontrôleur par l'intermédiaire d'une simple instruction qui est exécutée en un seul cycle d'horloge. Cela signifie que pendant un cycle d'horloge l'Unité Arithmétique et Logique '**ALU'** exécute l'opération et le résultat est stocké, après coup, dans le registre de sortie, le tout dans un seul cycle d'horloge.

L'architecture résultante est plus efficace en réalisant des opérations jusqu'à dix fois plus rapidement qu'avec des microcontrôleurs conventionnels **CISC**.

Les registres spéciaux sont dit aussi registre d'accès rapide et 6 des 32 registres peuvent être employés comme trois registre d'adresse 16 bits pour l'adressage indirects d'espace de données (**X, Y & Z**). Le troisième **Z** est aussi employé comme indicateur d'adresse pour la fonction de consultation de table des constantes, mais nous reviendrons dessus ultérieurement.

Les 32 registres sont détallés dans le tableau qui suit avec l'adresse effective dans la mémoire **SRAM** :

Bit 7 à 0	Adresse	Registre Spéciaux
R0	$00	
R1	$01	
Rn	$xx	
R26	$1A	Registre **X** Partie Basse
R27	$1B	Registre **X** Partie Haute
R28	$1C	Registre **Y** Partie Basse
R29	$1D	Registre **Y** Partie Haute
R30	$1E	Registre **Z** Partie Basse
R31	$1F	Registre **Z** Partie Haute

Comme vous l'avez compris, trois types de mémoire sont utilisés dans la série **ATMEGA**, la mémoire programme **FLASH**, la mémoire de donnée **SRAM** et la mémoire morte de type **EEPROM**.

La mémoire programme

La mémoire programme permet de stoker et de faire fonctionner le microcontrôleur, il contient de 4 à 256 Ko de programme selon le modèle du microcontrôleur. Le nombre d'écriture sur cette mémoire est limité à 10.000, largement suffisant pour la majorité des applications. La figure 1 donne un exemple de l'adressage de la mémoire **FLASH** du modèle **ATMEGA 32**.

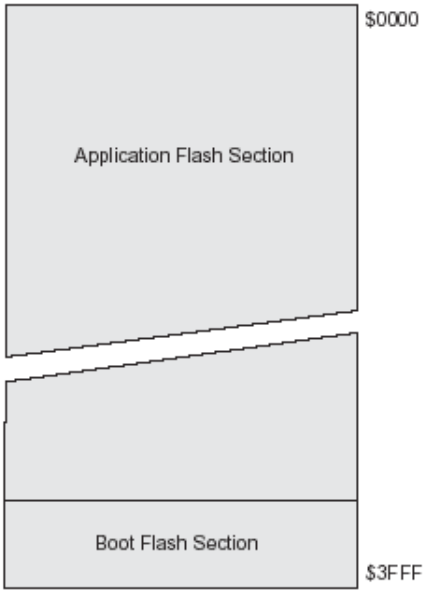


Figure 1, Adressage de la mémoire **FLASH**.

La mémoire de donnée

La mémoire de donnée contient les 32 registres de travail, les 64 registres de commande et la mémoire **SRAM** pour les variables du programme de 2048 octets pour le modèle **ATMEGA 32**. La figure 2 présente les relations entre espace physique et registre.

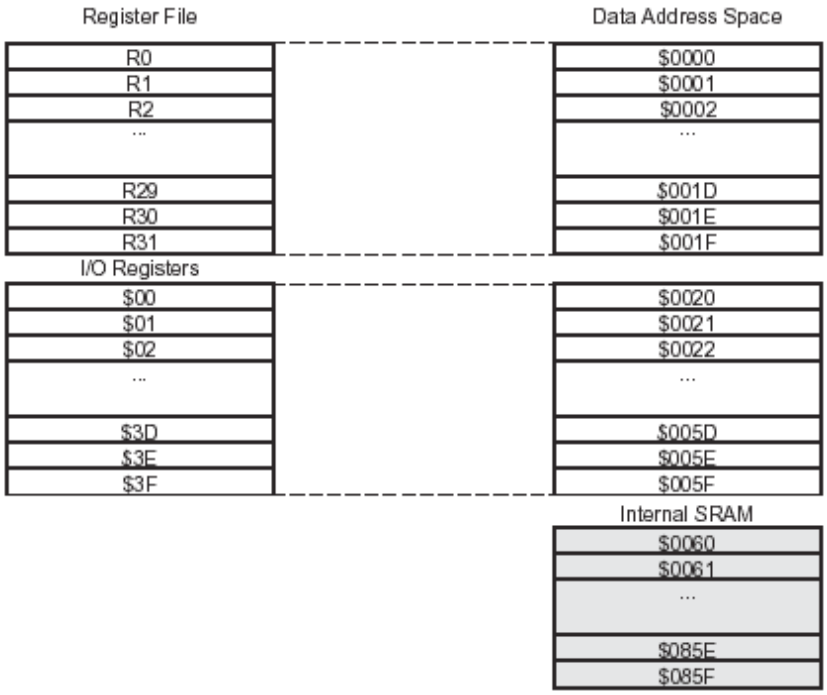


Figure 2, la mémoire de donnée avec la **SRAM**.

La fin de la mémoire **SRAM** est utilisée par la pile pour stocker les données de celle-ci en remontant dans la mémoire. De ce fait, la fin de la **SRAM** n'est pas réellement disponible en fonction de l'usage que vous faite de la pile.

La mémoire morte

La mémoire morte est de type **EEPROM** d'accès plus complexe contiendra la configuration du programme et les données importantes qui seront sauvées pendant l'absence de courant électrique. On peut écrire jusqu'à 100.000 fois dans l'**EEPROM**. La taille de **l'EEPROM** est fonction du modèle de microcontrôleur **ATMEGA** (de 256 bits à 8 Ko).

Base et Opération Logiques

Voici un petit rappel sur les bases hexadécimales, binaire et les opérations logiques pour en comprendre le fonctionnement.

Attention : les opérateurs logiques ici employés sont des références internationales, mais ne sont malheureusement pas les mêmes que ceux utilisés par l'assembleur pour des raisons de droit principalement.

Les Opérations Booléennes

L'opérateur ET (&)

L'opérateur **ET** est une multiplication booléenne qui si les deux valeurs sont à 1 donnera un résultat à 1 :

Opérateur 1	Opérateur 2	ET Logique
0	0	0
0	1	0
1	0	0
1	1	1

L'opérateur OU (!)

L'opérateur **OU** est une addition booléenne qui si l'une des deux valeurs est à 1 donnera un résultat à 1 :

Opérateur 1	Opérateur 2	OU Logique
0	0	0
0	1	1
1	0	1
1	1	1

L'opérateur OU Exclusif (Å)

L'opérateur **OU Exclusif** est une addition booléenne exclusive qui si l'une des deux valeurs seulement est à 1 donnera un résultat à 1 :

Opérateur 1	Opérateur 2	OU Exc. Logique
0	0	0
0	1	1
1	0	1
1	1	0

L'opérateur NON (¯)

L'opérateur **NON** est une négation booléenne qui inverse la valeur, un 1 donnera un résultat à 0 et vis versa :

Opérateur	NON Logique
0	1
1	0

La combinaison des opérations est possible, par exemple un **ET** et un **NON** donnera un **NONET** qui inversera le résultat tout simplement. Dans le document qui suit, les valeurs **NON** seront misent en rouge.

Les Bases 16, 10 et 2

Les chiffres et les nombres ne sont pas représentés tel quel dans la mémoire du microcontrôleur, ils sont converties en information binaire (en base 2) sur 8 bits (Bit = élément binaire valant 0 ou 1), qui donne un octet (octet ensemble de 8 bits) et sont représentées de la manière suivant : b00000000. On peut avoir jusqu'à 256 valeurs différentes avec ce système, de 0 à 255 avec 8 bits d'informations.

Inconvénient majeur, ce n'est pas très pratique d'aligner des lignes de 0 et de 1 pour écriture un programme. La conversion en hexadécimale (base 16) à donc été pris comme référence pour les systèmes à microcontrôleur et à microprocesseur à 8 bits et plus. Il reste que le binaire est pratique pour le contrôle des Port d'entrée/sortie ou pour localiser rapidement une information dans un registre.

La table qui suit donne un exemple de conversion simple et rapide :

Décimale	Binaire	Hexadécimale
0	b0000	$0
1	b0001	$1
2	b0010	$2
3	b0011	$3
4	b0100	$4
5	b0101	$5
6	b0110	$6
7	b0111	$7
8	b1000	$8
9	b1001	$9
10	b1010	$A
11	b1011	$B
12	b1100	$C
13	b1101	$D
14	b1110	$E
15	b1111	$F

Comme on peut le voir, l'hexadécimal ou encore 'l'hexa' est constitué de chiffre et de lettre, avec un chiffre en hexa on représente 4 bits, il faut donc 2 chiffres hexa pour faire un octet, celui de gauche sera le poids fort du nombre et celui de droite sera le poids faible :

Décimal	Poids Fort	Poids Faible	Hexa
165	$A	$5	$A5
37	$2	$5	$25

Le signe de l'hexa est le dollar '$' et le signe du binaire est le 'b', il n'y a pas de signe pour le décimal.

Pour convertir un chiffre de l'hexa en décimale, il suffit de multiplier par 16 le poids fort et d'ajouter le poids faible : $A = 10 et 10 x 16 = 160 + $5 = 165, vraiment simple non.

Pour faire l'inverse c'est un peu plus dur, il faut diviser le nombre décimal par 16 qui donnera le poids fort et le reste de la division donnera le poids faible.

165	16
10	5

Ceci reste valable pour les valeurs inférieures à 255 bien sûr.

Pour les valeurs supérieures, il faut au préalable diviser le nombre par 256 et reprendre le même principe pour les deux parties, d'autre méthode existe, mais je me limiterais à ce rappel qui est suffisant dans la majorité des cas.

Les Registres d'Etat SREG

Le registre indispensable au système pour fonctionner est le registre d'état **SREG**. En effet, ce registre permet de réaliser les opérations de branchements qui autorisent des applications complexes.

Registre SREG (*Status Register*)

Le registre **SREG** ou registre d'état sert principalement avec les fonctions arithmétiques et logiques pour les opérations de branchements. Il indique et autorise aussi le fonctionnement des interruptions. Il est modifié par le résultat des manipulations mathématiques et logiques. C'est le principal registre qui sert à effectuer les branchements conditionnels après une opération arithmétique ou logique. Il peut être lu et écrit à tout moment sans aucune restriction.

Adresse	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
$3F	**I**	**T**	**H**	**S**	**V**	**N**	**Z**	**C**
L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E

I ***Global Interrupt Enable*** sert à activer (1) ou interdire (0) toutes les sources d'interruptions. Si ce bit n'est pas activé alors que vous avez programmé des interruptions, elles ne seront pas prises en compte.

T ***Copy Storage*** joue un rôle de tampon lors de manipulation de bits avec les instructions ***BLD*** et ***BST***.

H ***Half Cary*** signale qu'une demi retenue a été réalisée lors de l'emploi d'une instruction arithmétique.

S ***Sign bit*** bit de signe résultant d'un **OU** exclusif avec le bit **N** et **V**.

V ***Overflow bit*** indique un dépassement de capacité lors des opérations arithmétique et logique.

N ***Negative bit*** signale que le résultat de la dernière manipulation arithmétique est négatif.

Z ***Zéro bit*** le résultat de la dernière manipulation arithmétique est égal à zéro.

C ***Cary bit*** l'opération arithmétique a donné lieu à une retenue.

Et bien maintenant, voici comment cela marche avec quelques exemples simples.

Le programme minimum

Pour commencer, je vais présenter et commenter le programme minimum que le microcontrôleur à besoin pour fonctionner. Cet exemple permet de faire clignoter une **LED** branché sur le port **A0** du microcontrôleur.

```
;***********************************************************************
;COMPOSANT        :  ATMEL AVR 8 Bits (RISC) - ATMega32        Quartz = 8 MHz
;PROGRAMME        :  Cligotant.asm
;VERSION          :  V 1.00
;DATE             :  10/05/2005
;DERNIERE MAJ.    :  xx/xx/2005
;DESCRIPTION      :  Exemple de clignotement d'une LED sur port A0
;AUTEUR           :  Jean-Noël
```

L'entête permet de savoir sur quoi l'on travail et de renseigner l'utilisateur sur le type de programme.

```
;***********************************************************************
;                DIRECTIVES D'ASSEMBLAGE
;***********************************************************************
;.DEVICE      atmega32      ;Type de Microcontrôleur (définit dans l'include)
.INCLUDE      "m32def.inc"  ;Fichier de définition du microcontrôleur
```

Les directives d'assemblage sont indispensables au programme qui va transformer le code assembleur en code machine que seul comprend le processeur. Je vous recommande de regarder le fichier 'm32def.inc' avec un éditeur de texte sans le modifier, cela vous donnera une idée de son contenu.

```
;***********************************************************************
;                PORTS D'ENTREES/SORTIES
;***********************************************************************
; Port A0        Branché sur une LED
;      A1 à A7   Libre
; Port B0 à B7   Libre
; Port C0 à C7   Libre
; Port D0 à D7   Libre
```

L'utilisation des entrés/sorties connecter sur le processeur avec les déférents périphériques faire référence de manière sûr.

```
;***********************************************************************
;                CONSTANTES
;***********************************************************************
; Pas de constante
```

Les constantes peuvent être utilisées dans le programme pour faire référence à des valeurs irrémédiablement fixes.

```
;***********************************************************************
;                DEFINITIONS DE REGISTRES
;***********************************************************************
; r0 à r15       Libre
; r16 à r20      Utilisé par le programme
```

```
; r21 à r25        Libre
; r26 à r31        Réservé pour registre X, Y, Z
```

L'utilisation des registres dans le programme permet de classer et d'utiliser correctement ceux-ci sans nuire à la lisibilité du programme.

```
;***************************************************************************
;                   ORGANISATION RAM
;***************************************************************************
.DSEG
.ORG    $60     ; début de la mémoire disponible pour l'utilisateur (vous)
;Ici on peut définir des variables locales d'un octet pour stocker des valeurs quelconques.
vAlumer:        .BYTE 1         ;Led alumé si = 1, éteinte si 0
```

Ici, nous rentrons dans l'espace **RAM** ou l'on peut définir des adresses mémoires par des noms simples pour stocker des données. L'adresse **RAM** commence toujours à $60, avant il y a les registres spéciaux et les registres d'entrées/sorties (voir le plan mémoire).

```
;***************************************************************************
;                   INTERRUPTIONS ET RESET
;***************************************************************************
; Même si les interruptions ne sont pas utilisées ces définitions doivent être déclarées
.CSEG                           ;Segment de Code
.ORG    $0000                   ;Positionnement au début de la mémoire
    jmp  RESET                  ;Reset Handler
    jmp  EXT_INT0               ;IRQ0 Handler
    jmp  EXT_INT1               ;IRQ1 Handler
    jmp  EXT_INT2               ;IRQ2 Handler
    jmp  TIM2_COMP              ;Timer2 Compare Handler
    jmp  TIM2_OVF               ;Timer2 Overflow Handler
    jmp  TIM1_CAPT              ;Timer1 Capture Handler
    jmp  TIM1_COMPA             ;Timer1 CompareA Handler
    jmp  TIM1_COMPB             ;Timer1 CompareB Handler
    jmp  TIM1_OVF               ;Timer1 Overflow Handler
    jmp  TIM0_COMP              ;Timer0 Compare Handler
    jmp  TIM0_OVF               ;Timer0 Overflow Handler
    jmp  SPI_STC                ;SPI Transfer Complete Handler
    jmp  USART_RXC              ;USART RX Complete Handler
    jmp  USART_UDRE             ;UDR Empty Handler
    jmp  USART_TXC              ;USART TX Complete Handler
    jmp  ADC_COMP               ;ADC Conversion Complete Handler
    jmp  EE_RDY                 ;EEPROM Ready Handler
    jmp  ANA_COMP               ;Analog Comparator Handler
    jmp  TWI                    ;Two-wire Serial Interface Handler
    jmp  SPM_RDY                ;Store Program Memory Ready Handler
```

Les vecteurs d'interruption ne sont pas fixés dans le processeur, c'est à nous de le faire par cette déclaration. Cela est fortement recommandé, même si vous n'utiliser pas les interruptions. Le premier branchement que fait le processeur est un **RESET**, donc si le vecteur n'est pas défini, le programme ne fonctionnera jamais ! Si une interruption à lieu de manière inopportune et que le vecteur n'est pas déclarer le processeur se bloque.

```
;*****************************************************************************
;                   PROGRAMME PRINCIPAL (RESET)
;*****************************************************************************
;Initialise la pile en bas de la mémoire RAM en adresse 16 bits
RESET: ldi      r16, HIGH(RAMEND)       ;Charge la valeur haute de l'adresse en fin mémoire RAM
       out      SPH, r16                ;Positionne le pointeur de pile haut sur cette adresse
       ldi      r16, LOW(RAMEND)        ;Charge la valeur basse de l'adresse en fin mémoire RAM
       out      SPL, r16                ;Positionne le pointeur de pile bas sur cette adresse
```

Le programme va donc exécuter le saut sur le branchement **RESET**, celui-ci va commencer par initialiser la pile à la fin de la mémoire **RAM**, sur une adresse à 16 bits, donc pour cela on utilise les directives de compilations '**HIGH'** et '**LOW'** qui extrait les parties haute et basse de l'adresse 16 bits. On commence toujours par la partie haute puis la partie basse ensuite.

L'instruction '**ldi'** permet de lire une variable mémoire et de la transférer dans un registre spécial (d'accès rapide), soit r16 = partie haute de **RAMEND**. Pour la première ligne du programme.

L'instruction '**out**' permet d'écrire dans les registres d'entrées/sorties, soit **SPH** = r16 (**SPH** adresse haute de la pile).

```
;-----------------------------------------------------------------------------
;Initialise du WatchDog (Chien de garde) pour éviter de bloquer le processeur (Facultatif)
       wdr                              ;Mis à 0 du compteur du Watchdog
       ldi      r16, $0F                ;WDE = 1 avec base de temps maximum 1900 ms
       out      WDTCR, r16              ;Ecriture du registre du WatchDog
```

Le WatchDog (chien de garde) est un programme interne au processeur qui surveille le fonctionnement du processeur, si celui-ci s'arrête d'une manière ou d'une autre, le WatchDog s'en aperçoit et relance le processeur depuis le début (exécution d'un Reset). L'instruction 'wdr' doit être placée judicieusement dans le programme pour remettre à zéro le compteur, ceci autant de fois que nécessaire.

```
;-----------------------------------------------------------------------------
;Initialisation Port A     ;PA0 = Led – PA1 à PA7 libres
       ser      r16             ;Port en sortie (les bits du port sont mis à 1, soit en sortie)
       out      DDRA,  r16      ;Ecriture sur le
       clr      r16             ;Port en bas (les bits du port sont mis à 0, Led éteinte)
       out      PORTA, r16      ;Port A mis à zéro
```

L'initialisation du **Port A** est ensuite effectuée en utilisant la variable rapide 'r16'. Celle-ci est positionnée à 0 ou à 255 par respectivement les instructions '**clr'** et '**ser'**.

Pourquoi utiliser la variable 'r16' et non la variable 'r0', la raison en est fort simple, les variables 'r16' à 'r32' sont utilisables par toutes les instructions alors que les variables 'r0' à 'r15' on des restrictions d'utilisation pour certaines instructions et il faudrait vérifier à chaque fois que la variable peut être utilisée.

Les constantes **DDRA** et **PORTA** sont définies dans le fichier des définitions du processeur, renseigné au début du programme dans la fonction '.include'. Elles remplacent l'adresse exacte des registres des entrées/sorties et permettent de rendre plus lisible le programme en langage assembleur.

```
;-----------------------------------------------------------------------------
;Fin d'initialisation
       wdr                      ;Réamorce le watchdog
       jmp      Debut           ;Fin d'initialisation, saute au début
```

La fin de l'initialisation du programme renvoie vers le point de départ du programme par l'instruction '**jmp'** qui est un saut inconditionnel vers une étiquette* '**Début'**. On en profite pour remettre à zéro le WatchDog.

```
;_____________________________________________________________________
;Interruptions non utilisées
EXT_INT0:       ;IRQ0
EXT_INT1:       ;IRQ1
EXT_INT2:       ;IRQ2
TIM2_COMP:      ;Timer2 Comparaison
TIM2_OVF:       ;Timer2 Overflow
TIM1_CAPT:      ;Timer1 Capture
TIM1_COMPA:     ;Timer1 CompareA
TIM1_COMPB:     ;Timer1 CompareB
TIM1_OVF:       ;Timer1 Overflow
TIM0_COMP:      ;Timer0 Compare
TIM0_OVF:       ;Timer0 Overflow
SPI_STC:        ;SPI Transfer Complete
USART_RXC:      ;USART RX Complete
USART_UDRE:     ;UDR Empty
USART_TXC:      ;USART TX Complete
EE_RDY:         ;EEPROM Ready
ADC_COMP:       ;ADC Conversion Complète
ANA_COMP:       ;Analog Comparator
TWI:            ;Two-wire Serial Interface
SPM_RDY:        ;Store Program Memory Ready
        nop             ;Ne rien faire dans cette interruption
        reti            ;Fin de l'interruption
```

Les vecteurs d'interruption étant définis, il renvoie sur des étiquettes* qu'ils faut définir une fois dans le programme, même si les interruptions ne sont pas utilisés.

Si une interruption est déclanchée, l'exécution des instructions sera faite, soit '**nop'** et '**reti'**.

'**nop'** est une instruction qui ne fait rien, elle consomme un cycle d'horloge c'est tout.

'**reti'** est l'instruction qui signale la fin d'un sous-programme d'interruption. Elle est indispensable en fin de programme lors de l'appel d'une interruption.

*Une étiquette est la représentation d'une adresse physique dans la mémoire programme. Elle est constituée d'un mot suivi par : 'Deux points'. Cela permet d'écrire des programmes avec des branchements sans connaître l'adresse d'arrivée du saut à l'avance, pratique non. C'est le programme d'assemblage qui s'occupera de localiser l'adresse du saut et de la remplacer en lieu et place de l'étiquette.

```
;***********************************************************************
;_____________________________________________________________________
;Programme principal
;_____________________________________________________________________
Debut:                  ;Programme principal

;Exemple, cligotement de la LED
    wdr                 ;Réamorce le WatchDog
    in    r16, PORTA    ;Lire Port A
    ldi   r17, PA0      ;Le port A0
```

```
    eor   r16, r17        ;Inversion du bit 0 de r16
    out   PORTA, r16      ;Ecrir Port A0
```

Nous arrivons enfin au programme principal, celui qui doit faire ceux pour quoi il est fait, faire clignoter une LED à un rythme définit.

En premier lieu, on remet le WatchDog à zéro avec l'instruction **'wdr'**, cela évitera au processeur de faire un reset inutile. Cette opération doit être faite au moins une fois par second, sinon, le processeur exécute un Reset automatique.

Pour faire clignoter la LED, on doit savoir si elle est allumée ou éteinte, pour ce faire, il faut lire l'état du **Port A** et en extraire l'information, allumé si égale à 1, sinon éteinte. C'est le but de l'instruction **'in r16, PORTA'**. L'instruction **'in'** permet de lire le contenu d'un Port d'entrée/sortie, elle ne fait que cela. C'est l'équivalent en basic de l'attribution d'une variable ; **r16** = **PortA**.

Ensuite, on chargera le bit qui spécifie la position du **Port A0** avec l'instruction **'ldi r17, PA0'**. L'instruction **'ldi'** permet de charger la variable **'r17'** par le contenu de la constante **PA0** qui est défini dans le fichier des définitions du processeur, équivalent Basic de **Set r17 = PA0**.

L'instruction **'eor'** effectue un **OU Exclusif** entre la variable **'r16'** et **'r17'**, ce qui a pour effet d'inverser le contenu binaire de la valeur de **r16** « **Port A0 »** (voir les opérateurs logiques au début). L'opération est la suivante :

'r16' = **b00000000** et **'r17'** = **b00000001**, l'on fait un **'eor'** **'r16'** prendra la valeur **b00000001**. Si l'on refait l'opération, **'r16'** reprendra la valeur **b00000000**, on à bien inversion du **bit 0** donc du **Port A0** ! Génial !

Puis on modifie le **Port A** avec l'instruction **'out'** avec le nouveau contenu de la variable **r16** ; la **LED** change d'état et s'allumera au cycle d'horloge suivant.

Ensuite il faut attendre un certain temps, car sinon, le processeur est trop rapide pour que l'œil humain puis voir le clignotement de la **LED**, il faut faire attendre le processeur avec une boucle d'attente !

```
;Boucle d'attente de quelques dixième de seconde
    ldi   r18, 250      ;Charge le temps d'attente primaire (125 µs)
    ldi   r19, 200      ;Charge le temps d'attente secondaire (25 ms)
    ldi   r20, 20       ;Charge le temps d'attente tertiaire (1/2 seconde)
Attente:                ;Attente 0,125 ns (Fréquence oscillateur à 8MHz)
    dec   r18           ;Décrément de 1 de la variable r18 (125 ns)
    nop                 ;Attente d'un cycle d'horloge (125 ns)
    brne  Attente       ;Boucle sur Attente jusqu'à l'obtention d'un zéro dans r18 (250 ns)
```

Pour calculer le temps que va mettre la boucle avec un quartz à 8 Mhz, il faut tout d'abord connaître le temps que dur un cycle d'horloge, soit 1s / 8.000.000 = 125 ns (nano seconde) par cycle d'horloge CPU.

Ensuite il faut connaître le nombre de cycle que chaque instruction consomme, soit dec = 1 cycle, nop = 1 cycle et brne = 2 cycles si saut et 1 cycle sans saut. Soit au total 4 cycles d'horloge tant que la boucle n'est pas finit puis 3 quant le test est vrai (r18 = 0).

Donc au maximum on à 4 * 125 ns = 500 ns par boucle, on le multiplie par 250 boucles ce qui donne 125 µs (micro seconde), il faut donc refaire une autre boucle (secondaire) pour avoisiner la demi seconde.

```
;Boucle secondaire qui réutilise la boucle primaire de 125 µs pour attendre 25 ms
    ldi   r18, 250      ;Réinitialise la boucle primaire
    dec   r19           ;Décrémente de 1 la boucle secondaire
    brne  Attente       ;Boucle sur Attente jusqu'à l'obtention d'un zéro dans r19
```

Donc on à 200 * 125 µs = 25 ms (milliseconde) pour la boucle secondaire, il faut donc refaire une autre boucle (tertiaire) pour avoisiner la demi seconde.

```
;Boucle tertiaire qui réutilise la boucle secondaire et primaire de 25 ms
    ldi   r19, 200        ;Réinitialise la boucle secondaire
    dec   r20             ;Décrémente de 1 la boucle tertiaire
    brne  Attente         ;Boucle sur Attente jusqu'à l'obtention d'un zéro dans r20
```

Nous avons enfin un temps d'attente de 500 ms, plus ou moins quelques ns perdue dans les opérations intermédiaires, donc 20 * 25 ms = 500 ms soit une demi seconde, ce que nous cherchions à faire.

```
    rjmp Debut            ;Boucle infini sur le programme principal
;_____________________________________________________________________________
;*****************************************************************************
```

C'est la fin du programme, on reboucle sur le programme principal avec l'instruction '**rjmp'**, qui fait un branchement inconditionnel court sur le programme de début. On aurait pu utiliser '**jmp'**, mais le nombre de cycle d'horloge est un peu plus important et l'occupation en mémoire aussi, avec un octet de plus. L'instruction '**rjmp'** permet de faire des branchements courts sur moins de 128 octets de déplacement uniquement, le compilateur vous signalera le cas de débordement si le déplacement est supérieur à cette valeur.

Vous pouvez remarquer que le processeur va très vite et qu'il nous faut trois compteurs d'attente pour aboutir à une attente de quelques secondes !

Il est évident que d'autres solutions plus économiques en temps **CPU** existent, par exemple en utilisant les compteurs ou encore les interruptions, mais cela donne une première idée de la forme d'un programme assez simple.

Nous verrons dans un proche avenir la suite de ces exemples.

Jeu d'Instruction

L'ATMEGA à un jeu de 131 instructions qui seront détaillées dans la suite de ce document.

Le code instruction est à utiliser dans les programmes écrit en assembleur, l'opérant spécifie les variables ou constantes utilisées par l'instruction, le registre **SREG** est modifier par l'instruction en fonction du résultat de celle-ci, et le nombre de cycle **CPU** est donnée en dernier.

Les instructions sont présentées dans les tableaux qui suivent par type :

Code	Opérant	Description	Opération	Registre	Cycle
Instructions Arithmétiques et Logique					
ADD	Rd, Rr	Addition sans Retenue	Rd = Rd + Rr	Z C N V S H	1
ADC	Rd, Rr	Addition avec Retenue	Rd = Rd + Rr + C	Z C N V S H	1
ADIW	Rd, k	Addition Immediate 16 bits	Rd+1:Rd = Rd+1:Rd + k	Z C N V S	2 (1)
SUB	Rd, Rr	Soustraction sans Retenue	Rd = Rd – Rr	Z C N V S H	1
SUBI	Rd, k	Soustraction Immédiate sans Retenue	Rd = Rd – k	Z C N V S H	1
SBC	Rd, Rr	Soustraction avec Retenue	Rd = Rd – Rr – C	Z C N V S H	1
SBCI	Rd, K	Soustraction Immédiate avec Retenue	Rd = Rd – k – C	Z C N V S H	1
SBIW	Rd, K	Soustraction Immédiate 16 bits	Rd+1:Rd = Rd+1:Rd – k	Z C N V S	2 (1)
AND	Rd, Rr	ET Logique	Rd = Rd & Rr	Z N V S	1
ANDI	Rd, k	ET Logique Immédiat	Rd = Rd & k	Z N V S	1
OR	Rd, Rr	OU Logique	Rd = Rd ! Rr	Z N V S	1
ORI	Rd, k	OU Logique Immédiat	Rd = Rd ! k	Z N V S	1
EOR	Rd, Rr	OU Exclusif	Rd = Rd ⊕ Rr	Z N V S	1
COM	Rd	Complément à 1	Rd = $FF – Rd	Z C N V S	1
NEG	Rd	Négation (Complément à 2)	Rd = $00 – Rd	Z C N V S	1
SBR	Rd, k	Mise à 1 dans Registre (OU)	Rd = Rd ! k	Z N V S	1
CBR	Rd, k	Mise à 0 dans Registre (ET)	Rd = Rd & ($FF - k)	Z N V S	1
INC	Rd	Incrément	Rd = Rd + 1	Z N V S	1
DEC	Rd	Décrément	Rd = Rd – 1	Z N V S	1
TST	Rd	Test à Zéro ou Négatif	Rd = Rd ! Rd	Z N V S	1
CLR	Rd	Effacement du Registre	Rd = $00	Z N V S	1
SER	Rd	Mis à 1 du Registre	Rd = $FF	-	1
MUL	Rd, Rr	Multiplication non Signé	R1:R0 = Rd x Rr (UU)	Z C	2 (1)
MULS	Rd, Rr	Multiplication Signée	R1:R0 = Rd x Rr (SS)	Z C	2 (1)
MULSU	Rd, Rr	Multiplication Signée avec non Signé	R1:R0 = Rd x Rr (SU)	Z C	2 (1)
FMUL	Rd, Rr	Multiplication Fractionnaire non Signé	R1:R0=(Rd x Rr)<<1 (UU)	Z C	2 (1)
FMULS	Rd, Rr	Multiplication Fractionnaire Signée	R1:R0=(Rd x Rr)<<1(SS)	Z C	2 (1)
FMULSU	Rd, Rr	Multiplication Fractionnaire Signée avec non Signé	R1:R0=(Rd x Rr)<<1(SU)	Z C	2 (1)

Code	Opérant	Description	Opération	Registre	Cycle
Instructions de Branchement et Saut					
RJMP	k	Saut Relatif	PC = PC + k + 1	-	2
IJMP		Saut Indirect Z	PC(15:0)=Z, PC(21:16)=0	-	2 (1)
EIJMP		Saut Etendu Indirect Z	PC(15:0)=Z, PC(21:16)=EIND	-	2 (1)
JMP	k	Saut	PC = k	-	3 (1)
RCALL	k	Sous-Programme Relatif	PC = PC + k + 1	-	3/4 (4)
ICALL		Sous-Programme Indirect Z	PC(15:0)=Z, PC(21:16)=0	-	3/4(1,4)
EICALL		Sous-Programme Etendu Indirect Z	PC(15:0)=Z, PC(21:16)=EIND	-	4 (4)
CALL	k	Sous-Programme	PC = k	-	4/5(1,4)
RET		Retour de Sous-Programme	PC = STACK	-	4/5 (4)
RETI		Retour d'Interruption	PC = STACK	I	4/5 (4)
CPSE	Rd, Rr	Compare et Saute si Égal	Si Rd=Rr PC=PC+2 ou 3	-	1/2/3
CP	Rd, Rr	Comparer	Rd – Rr	Z C N V S H	1
CPC	Rd, Rr	Comparez avec Retenue	Rd - Rr - C	Z C N V S H	1
CPI	Rd, k	Comparez Immédiat	Rd – K	Z C N V S H	1
SBRC	Rr, b	Sautez si le bit du Registre à 0	Si Rr(b)=0 PC=PC+2 ou 3	-	1/2/3
SBRS	Rr, b	Sautez si le bit du Registre à 1	Si Rr(b)=1 PC=PC+2 ou 3	-	1/2/3
SBIC	A, b	Sautez si le bit du Registre I/O à 0	Si A, b=0 PC=PC+2 ou 3	-	1/2/3
SBIS	A, b	Sautez si le bit du Registre I/O à 1	Si A, b =1 PC=PC+2 ou 3	-	1/2/3
BRBS	s, k	Branche si SREG(s) à 1	Si SREG(s)=1 PC=PC+k+1	-	1/2
BRBC	s, k	Branche si SREG(s) à 0	Si SREG(s)=0 PC=PC+k+1	-	1/2
BREQ	k	Branche si Égal Z à 1	Si Z=1 PC=PC+k+1	-	1/2
BRNE	k	Branche si Non Égal Z à 0	Si Z=0 PC=PC+k+1	-	1/2
BRCS	k	Branche si Retenue C à 1	Si C=1 PC=PC+k+1	-	1/2
BRCC	k	Branche si Retenue C à 0	Si C = 0 PC=PC+k+1	-	1/2
BRSH	k	Branche si Egal ou Plus Haut	Si C=0 PC=PC+k+1	-	1/2
BRLO	k	Branche si Plus bas	si C = 1 PC=PC+k+1	-	1/2
BRMI	k	Branche si Négatif	Si N=0 PC=PC+k+1	-	1/2
BRPL	k	Branche si Positif	Si N = 0 PC=PC+k+1	-	1/2
BRGE	k	Branche si Plus Grand ou Égal Signé	Si N⊕V=0 PC=PC+k+1	-	1/2
BRLT	k	Branche si Moins Que Signé	Si N⊕V=1 PC=PC+k+1	-	1/2
BRHS	k	Branche si Drapeau Moitié H à 1	Si H=1 PC=PC+k+1	-	1/2
BRHC	k	Branche si Drapeau Moitié H à 0	Si H=0 PC=PC+k+1	-	1/2
BRTS	k	Branche si Drapeau T=1	Si T=1 PC=PC+k+1	-	1/2
BRTC	k	Branche si Drapeau T=0	Si T=0 PC=PC+k+1	-	1/2
BRVS	k	Branche si Débordement V=1	Si V=1 PC=PC+k+1	-	1/2
BRVC	k	Branche si Débordement v=0	Si V=0 PC=PC+k+1	-	1/2
BRIE	k	Branche si Interruption active I=1	Si I=1 PC=PC+k+1	-	1/2
BRID	k	Branche si Interruption inactive I=0	Si I=0 PC=PC+k+1	-	1/2

Code	Opérant	Description	Opération	Registre	Cycle
Instructions de Transfert de Donnée					
MOV	Rd, Rr	Déplacement	Rd = Rr	-	1
MOVW	Rd, Rr	Déplacement 16 bits	Rd+1:Rd = Rr+1:Rr	-	1 (1)
LDI	Rd, k	Charge Immédiat	Rd = k	-	1
LDS	Rd, k	Charge Directe en Mémoire	Rd = (k)	-	2 (1,4)
LD	Rd, X	Charge Indirect	Rd = (X)	-	2 (1,4)
LD	Rd, X+	Charge Indirect Post Incrément	Rd = (X) et X = X + 1	-	2 (1,4)
LD	Rd, -X	Charge Indirect Pré Décrément	X = X – 1 et Rd = (X)	-	2 (1,4)
LD	Rd, Y	Charge Indirect	Rd = (Y)	-	2 (1,4)
LD	Rd, Y+	Charge Indirect Post Incrément	Rd = (Y) et Y = Y + 1	-	2 (1,4)
LD	Rd, -Y	Charge Indirect Pré Décrément	Y = Y – 1 et Rd = (Y)	-	2 (1,4)
LDD	Rd, Y+q	Charge Indirect avec Déplacement	Rd = (Y + q)	-	2 (1,4)
LD	Rd, Z	Charge Indirect	Rd = (Z)	-	2 (2,4)
LD	Rd, Z+	Charge Indirect Post Incrément	Rd = (Z) et Z = Z+1	-	2 (2,4)
LD	Rd, -Z	Charge Indirect Pré Décrément	Z = Z – 1 et Rd = (Z)	-	2 (2,4)
LDD	Rd, Z+q	Charge Indirect avec Déplacement	Rd = (Z + q)	-	2 (1,4)
STS	k, Rr	Stock Direct en Mémoire	(k) = Rd	-	2 (1,4)
ST	X, Rr	Stock Indirect	(X) = Rr	-	2 (2,4)
ST	X+, Rr	Stock Indirect Post Incrément	(X) = Rr et X = X + 1	-	2 (2,4)
ST	-X, Rr	Stock Indirect Pré Décrément	X = X – 1 et (X) = Rr	-	2 (2,4)
ST	Y, Rr	Stock Indirect	(Y) = Rr	-	2 (2,4)
ST	Y+, Rr	Stock Indirect Post Incrément	(Y) = Rr et Y = Y + 1	-	2 (2,4)
ST	-Y, Rr	Stock Indirect Pré Décrément	Y = Y – 1 et (Y) = Rr	-	2 (2,4)
STD	Y+q,Rr	Stock Indirect avec Déplacement	(Y + q) = Rr	-	2 (1,4)
ST	Z, Rr	Stock Indirect	(Z) = Rr	-	2 (2,4)
ST	Z+, Rr	Stock Indirect et Incrément postal	(Z) = Rr et Z = Z + 1	-	2 (2,4)
ST	-Z, Rr	Stock Indirect et Pré décroissance	Z = Z – 1 et (Z) = Rr	-	2 (2,4)
STD	Z+q,Rr	Stock Indirect avec Déplacement	(Z + q) = Rr	-	2 (1,4)
LPM		Mémoire Programme de Charge	R0 = (Z)	-	3 (3)
LPM	Rd, Z	Mémoire Programme de Charge	Rd = (Z)	-	3 (3)
LPM	Rd, Z+	Mémoire Programme de Charge Post Incrément	Rd = (Z) et Z = Z + 1	-	3 (3)
ELPM	Extended	Mémoire Programme de Charge	R0 = (RAMPZ:Z)	-	3 (1)
ELPM	Rd, Z	Mémoire Programme de Charge Etendue	Rd = (RAMPZ:Z)	-	3 (1)
ELPM	Rd, Z+	Mémoire Programme de Charge Etendue Post Incrément	Rd=(RAMPZ:Z) et Z=Z + 1	-	3 (1)
SPM		Stock Mémoire Programme	(Z) = R1:R0	-	- (1)
IN	Rd, A	Entrée d'un Port	Rd = I/O(A)	-	1
OUT	A, Rr	Sortie sur Port	I/O(A) = Rr	-	1
PUSH	Rr	Entrée du Registre de Pile	STACK = Rr	-	2 (1)
POP	Rd	Sortie du Registre de Pile	Rd = STACK	-	2 (1)

Code	Opér.	Description	Opération	Registre	Cycle
Instructions de Bit et Bit test					
LSL	Rd	Décalage Logique à Gauche	C=Rd(7), Rd(n+1)=Rd(n), Rd(0)=0	Z C N V H	1
LSR	Rd	Décalage Logique à Droite	C=Rd(0), Rd(n)=Rd(n+1), Rd(7)=0	Z C N V	1
ROL	Rd	Rotation à Gauche	Rd(0)=C, Rd(n+1)=Rd(n), C=Rd(7)	Z C N V H	1
ROR	Rd	Rotation à Droite	Rd(7)=C, Rd(n)=Rd(n+1), C=Rd(0)	Z C N V	1
ASR	Rd	Décalage Arithmétique à Droite	Rd(n) = Rd(n+1), n=0:6	Z C N V	1
SWAP	Rd	Échange demi-parti Réciproque	Rd(3:0)=Rd(7:4) et Rd(7:4)=Rd(0:3)	-	1
BSET	s	Drapeau SREG(s) à 1	SREG(s) = 1	SREG(s)	1
BCLR	s	Drapeau SREG(s) à 0	SREG(s) = 0	SREG(s)	1
SBI	A, b	Registre d'entrée-sortie à 1	I/O(A, b) = 1	-	2
CBI	A, b	Registre d'entrée-sortie à 0	I/O(A, b) = 0	-	2
BST	Rr, b	Drapeau T = position b du Registre	T = Rr(b)	T	1
BLD	Rd, b	Registre position b = Drapeau T	Rd(b) = T	-	1
SEC		Drapeau Retenue à 1	C = 1	C	1
CLC		Drapeau Retenue à 0	C = 0	C	1
SEN		Drapeau Négatif à 1	N = 1	N	1
CLN		Drapeau Négatif à 0	N = 0	N	1
SEZ		Drapeau Zéro à 1	Z = 1	Z	1
CLZ		Drapeau Zéro à 0	Z = 0	Z	1
SEI		Drapeau d'Interruption Permit	I = 1	I	1
CLI		Drapeau d'Interruption Arrêté	I = 0	I	1
SES		Drapeau Signé à 1	S = 1	S	1
CLS		Drapeau Signé à 0	S = 0	S	1
SEV		Drapeau Débordement à 1	V = 1	V	1
CLV		Drapeau Débordement à 0	V = 0	V	1
SET		Drapeau Transfert à 1	T = 1	T	1
CLT		Purifiez Transfert à 0	T = 0	T	1
SEH		Drapeau Moitié de Retenue H à 1	H = 1	H	1
CLH		Drapeau Moitié de Retenue H à 0	H = 0	H	1

Code	Opérant	Description	Registre	Cycle
Instruction de Contrôle MCU				
BREAK		Voir la description de l'instruction de BREAK	-	1 (1)
NOP		Pas d'Opération (Attente d'un cycle)	-	1
SLEEP		Voir la description de l'instruction de SLEEP	-	1
WDR		Voir le chapitre sur le Watchdog ou l'instruction WDR	-	1

Notes : 1. Cette instruction n'est pas disponible dans tous les modèles.

2. L'ensemble des variantes des instructions n'est pas disponible dans tous les modèles.

3. L'ensemble des variantes de l'instruction **LPM** n'est pas disponible dans tous les modèles.

4. Le cycle d'horloge pour les accès mémoire de Données n'est pas valable pour des accès via l'interface externe de **RAM**. Pour **LD**, **ST**, **LDS**, **STS**, **PUSH**, **POP**, ajouter un cycle de plus pour chaque état d'attend. Pour **CALL**, **ICALL**, **EICALL**, **RCALL**, **RET**, **RETI** avec **PC** à 16 bits, l'ajoutent de trois cycles plus deux

cycles pour chaque état d'attente. Pour **CALL**, **ICALL**, **EICALL**, **RCALL**, **RET**, **RETI** avec le **PC** à 22 bits, l'ajoute de cinq cycles plus trois cycles pour chaque état d'attend.

Légende du jeu d'instruction

Registres et Opérateurs

Rd : Registre de Destination (ou de source) Ecrit,
Rr : Registre Source Lu,
k : Adresse en mémoire,
b : Bit du Registre ou Registre d'Entrée/Sortie (3 bits),
s : Bit du Registre de Statut (3 bits),
X, Y, Z : Registre d'Adresse Indirect (X=R27:R26, Y=R29:R28 et Z=R31:R30),
A : Adresse d'Entrée/Sortie,
q : Adressage Indirect (6 bits).

RAMPX, RAMPY, RAMPZ, RAMPD

Les registres sont enchaînés avec **X-, Y-** et **Z-** permettant l'adressage indirect de l'espace de données du **MCU** en débordant l'espace de données de 64 **Ko**. Registre enchaîné **Z-** permettant l'adressage direct de l'espace de données entier du **MCU** en débordant l'espace de données de 64 **Ko**.

EIND

Registre enchaîné avec le mot d'instruction permettant un appel indirect sur l'espace programme entier du **MCU** en débordant l'espace programme de 64 **Ko**.

STACK

La pile pour le retour d'adresse des registres poussés dans **SP**, Indicateur de Pile pour empiler-dépiler.

DRAPEAU

= : Drapeau affecté par instruction
0 : Drapeau purifié par instruction
1 : Drapeau mis par instruction
- : Drapeau non affecté par instruction

Les instructions de branchement

Les instructions de branchements permettent de tester des grandeurs entre elle :

Évaluez	Booléen	Instruction	Remarque
Rd > Rr	Z & (N ⊕ V) = 0	BRLT	Signé
Rd >= Rr	(N ⊕ V) = 0	BRGE	Signé
Rd = Rr	Z = 1	BREQ	Signé
Rd <= Rr	Z ! (N ⊕ V) = 1	BRGE	Signé
Rd < Rr	(N ⊕ V) = 1	BRLT	Signé
Rd > Rr	C ! Z = 0	BRLO	Non signé
Rd >= Rr	C = 0	BRCC	Non signé
Rd = Rr	Z = 1	BREQ	Non signé
Rd <= Rr	C ! Z = 1	BRSH	Non signé
Rd < Rr	C = 1	BRLO	Non signé
Retenu	C = 1	BRCS	Simple
Négatif	N = 1	BRMI	Simple
Déborde	V = 1	BRVS	Simple
Zéro	Z = 1	BREQ	Simple

Le programme assembleur AVR

Ce chapitre décrit l'utilisation de l'assembleur **ATMEL** qui couvre la gamme entière de microcontrôleurs dans la famille **AT90S** et **ATMEGA**. L'assembleur traduit le code source assemblé dans le code d'objet pour être employé avec un simulateur. L'assembleur produit aussi un code **PROMable** et un fichier facultatif **EEPROM** qui peut être programmé directement dans la mémoire programme et la mémoire **EEPROM** d'un microcontrôleur. L'assembleur produit des assignations de code fixées, par conséquent aucune jonction n'est nécessaire. L'assembleur fonctionne sous Windows95 et successeur. De plus, il y a une version de ligne de commande de **MS-DOS**. La version en fenêtres contient une fonction d'aide en ligne couvrant la plupart des commandes ce document.

Début Rapide

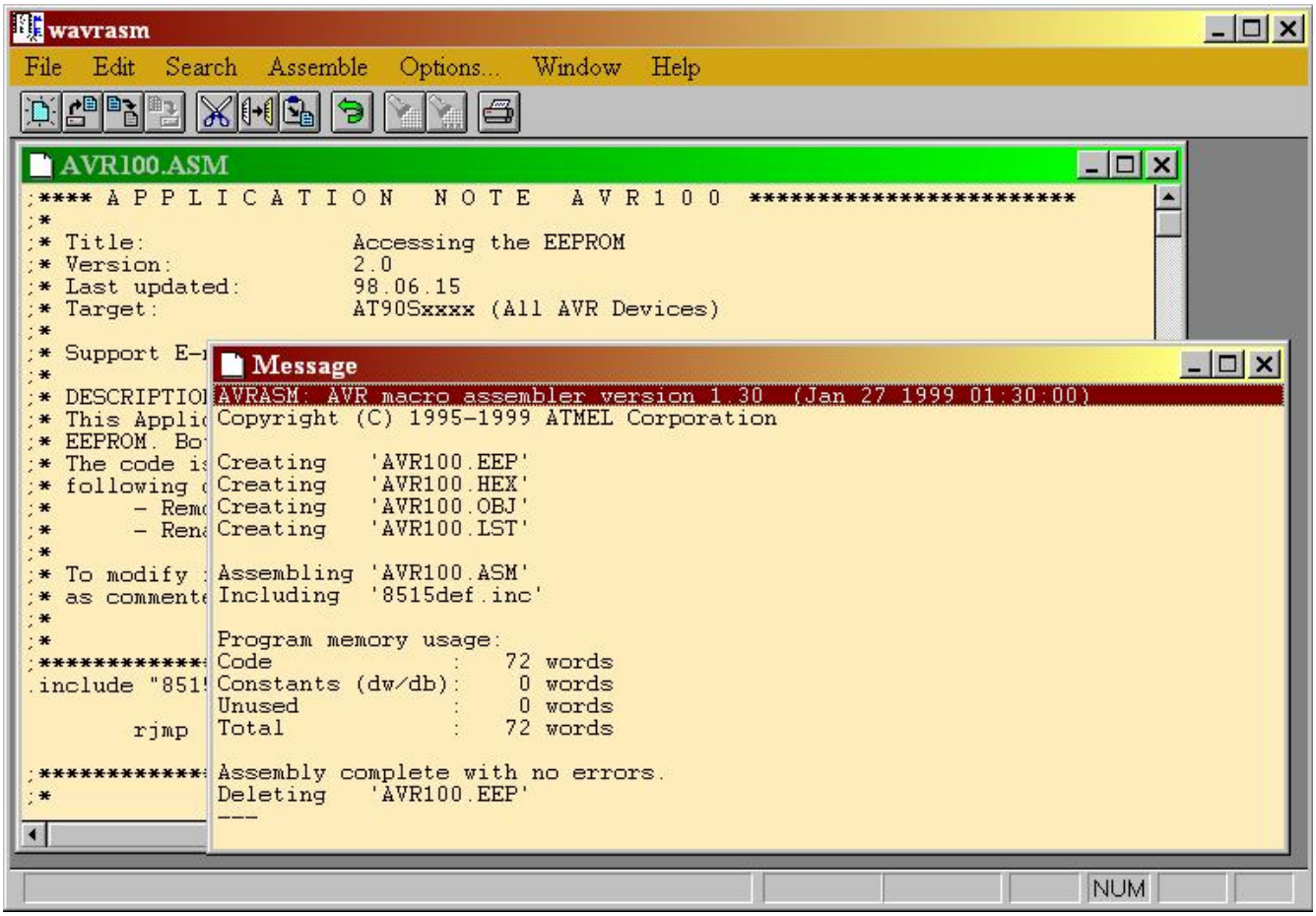
Nous supposons que l'assembleur **AVR** et tous les fichiers de programme qui viennent avec lui sont correctement installés sur votre ordinateur. Référez-vous s'il vous plaît aux instructions d'installation.

Démarrage

Ouvrir le logiciel **WavrAsm.exe** et commencez l'assemblage en choisissant « File » puis « Open » du menu ou en cliquetant sur la barre d'outils, ouvrez un fichier "avr100.asm". Cela charge le fichier en assembleur dans une fenêtre dite fenêtre rédacteur. Lisez l'entête du programme et jetez un coup d'œil dans le programme mais ne faites pas de changements encore.

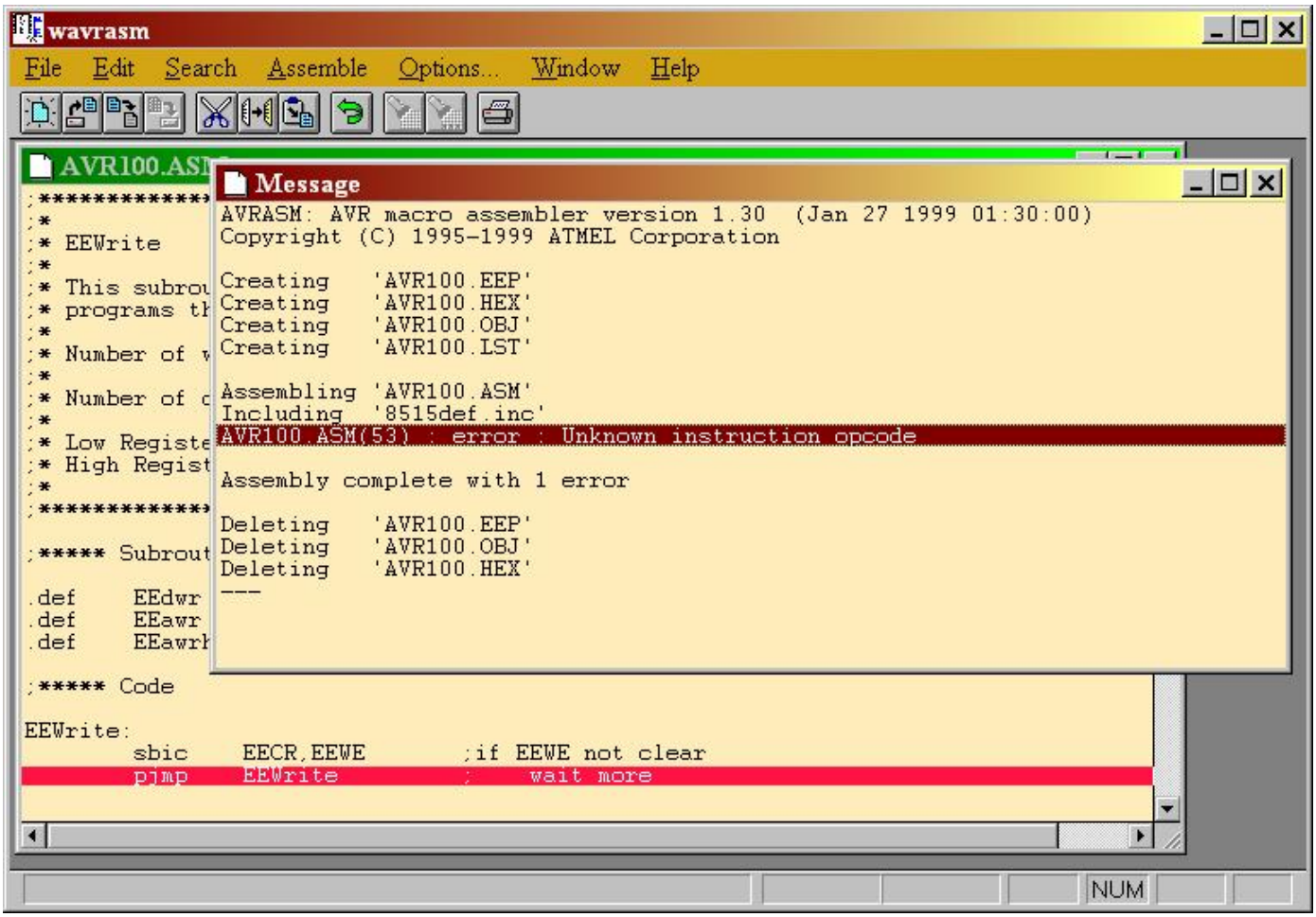
Assemblage de Votre Premier Fichier

Une fois que vous avez regardé le programme, lancé la compilation avec le menu « Assemble ». Une deuxième fenêtre (la fenêtre de Message) apparaît, contenant les messages de compilation et d'erreur. Cette fenêtre chevauchera la fenêtre principale, donc c'est une bonne idée de nettoyer votre espace de travail sur l'écran. Votre écran doit ressembler à cela :



Découverte et Correction d'Erreurs

Dans la fenêtre de Message, il semble que l'assemblage du programme sans bogues (défauts). Mais si on modifie la ligne 53 avec « pjmp », et que l'on assemble de nouveau, il y aura une erreur, elle doit être trouvée et corrigée. Par exemple :



Le premier message d'erreur dans la fenêtre de Message (celui annoncé pour être sur la ligne 53) et pressé sur le bouton gauche de la souris. Remarquez dans le rédacteur la fenêtre, une barre rouge horizontale est montrée sur la ligne 53. Le message d'erreur dit « Unknown instruction opcode ». Voir la figure ci-dessous.

Réassemblage

Une fois toutes les erreurs corrigées, un double-clic sur une erreur (pour activer la fenêtre Rédacteur) ou un clic à l'intérieur de la fenêtre Rédacteur avant de relancer l'assemblage « Assemble » une nouvelle fois. Si vous l'avez fait cela va jusqu'à maintenant, la fenêtre de Message dira que vous êtes couronnés de succès.

Assembler un programme source

L'assembleur travaille sur des fichiers source contenant les mnémoniques d'instruction, les étiquettes et les directives.

Les mnémoniques d'instruction et les directives prennent souvent des opérandes.

Les lignes de code doivent être limitées à 120 caractères.

Chaque ligne d'entrée peut être précédée par une étiquette, qui est une série alphanumérique terminée par un deux-points.

Les étiquettes sont employées comme des cibles pour les sauts et les instructions de branchement et comme des noms de variables dans la mémoire programme et la **SRAM**.

Une ligne d'entrée peut prendre une des quatre des formes suivantes :

1. [étiquette:] directive [opérandes] [; Commentaire]
2. [étiquette:] instruction [opérandes] [; Commentaire]
3. ; Commentaire
4. Ligne vide

Un commentaire à la forme suivante :

```
; [Texte]
```

Les articles placés dans des accolades sont facultatifs. Le texte entre le délimiteur de commentaire « **;** » et la fin de ligne **EOL** est ignorée par l'Assembleur.

Les étiquettes, les instructions et les directives sont décrites plus en détail dans les paragraphes suivants.

Exemple

```
label:      .EQU   var1 = 100      ; Met var1 à 100 (Directive)
            .EQU   var2 = 200      ; Met var2 à 200
test:       rjmp   test            ; Boucle infini (Instruction)
; Ligne de commentaire pur
; Autres commentaires...
```

Note : Il n'y a aucune restriction en ce qui concerne le placement de colonne d'étiquettes, des directives, des commentaires ou des instructions, mais pour facilité la lecture du programme source, il est préférable de placer les éléments similaires sur les mêmes « colonnes », comme le montre l'exemple.

Directives d'Assemblage

L'assembleur soutient quelques directives qui ne sont pas traduites directement en « opcodes ». Au lieu de cela, elles sont employées pour ajuster l'emplacement du programme dans la mémoire, définir des macros, initialiser la mémoire etc. On donne une vue d'ensemble des directives dans la table suivante :

Directive	Description
BYTE	Réserve d'octet pour une variable
CSEG	Segment de Code
DB	Définit le(s) octet(s) constant
DEF	Définit un nom symbolique pour un registre
DEVICE	Définit le modèle pour l'assemblage
DSEG	Segment de Données
DW	Définit le(s) mot(s) constant
ENDMACRO	Fin macro
EQU	Attribut un symbole à une expression
ESEG	Segment EEPROM
EXIT	Sortie du fichier
INCLUDE	Importe la source d'un autre fichier
LIST	Met en route la génération de listfile
LISTMAC	Met en route l'extension macro
MACRO	Commence une macro
NOLIST	Arrêt de la génération de listfile
ORG	Origine du programme en mémoire
SET	Déclare un symbole pour une expression

Nous allons maintenant présenter chaque directive d'assemblage avec des exemples.

BYTE – Reserve bytes to a variable - Réserve d'octet pour une variable

La directive **BYTE** réserve des ressources de mémoire dans la **SRAM**. Pour être capable de se référer à l'emplacement réservé, la directive **BYTE** doit être précédée d'une étiquette. La directive prend un paramètre, qui est le nombre d'octets à réserver. La directive peut seulement être employée dans un segment de données (voir des directives **CSEG**, **DSEG** et **ESEG**). Le paramètre est obligatoire et les octets alloués ne sont pas initialisés.

Syntaxe

LABEL: .BYTE expression

Exemple

```
.DSEG
var1:   .BYTE 1              ; Réserve 1 octet à var1
table:  .BYTE tab_size       ; Réserve tab_size octets
.CSEG
        ldi   r30, low(var1)  ; Charge registre Z bas
        ldi   r31, high(var1) ; Charge registre Z haut
        ld    r1, Z           ; Charge VAR1 dans r1
```

CSEG – Code Segment - Segment de Code

La directive **CSEG** définit le début d'un segment de code. Un fichier assembleur peut contenir plusieurs segments de code, qui sont enchaînés dans un seul segment de code après l'assemblage. La directive **BYTE** ne peut pas être employée dans un segment de code. Le type de segment par défaut est le segment de code. Les segments de code ont leur propre compteur d'emplacement qui est un compteur de mot. La directive **ORG** (voir la description plus tard dans ce chapitre) peut être employée pour placer le code et les constantes aux emplacements spécifiques dans la mémoire programme. La directive ne prend pas de paramètres.

Syntax

.CSEG

Example

```
            .DSEG               ; Début du segment de donnée
vartab:     .BYTE 4             ; Réserve 4 octets dans la SRAM
            .CSEG               ; Début du segment de code
const:      .DW 2               ; Ecrire $0002 dans la mémoire programme
            mov   r1, r0        ; Début du programme
```

DB-Define constant byte(s) in program memory or EEPROM memory - Définit le(s) octet(s) constant

La directive **DB** réserve des ressources de mémoire dans la mémoire programme ou la mémoire **EEPROM**. Pour être capable de se référer aux emplacements réservés, la directive **DB** doit être précédée par une étiquette. La directive **DB** prend une liste d'expressions et doit contenir au moins une expression. La directive **DB** doit être placée dans un segments de code ou un segments **d'EEPROM**. La liste d'expression est un ordre d'expressions, délimitées par des virgules. Chaque expression doit évaluer à un nombre compris entre -128 et 255. Si l'expression évalue un nombre négatif, le complément à 2 des 8 bits du nombre sera placé dans la mémoire programme ou l'emplacement de mémoire **EEPROM**. Si la directive **DB** est employée dans un segment de code et l'expression liste contient plus qu'une expression, les expressions sont réunies deux à deux et placées dans les mots mémoire programme de 16 bits. Si l'expression liste contient un nombre impair d'expressions, la dernière expression sera placée dans un mot mémoire programme propre 16 bits, même si la ligne suivante dans le code d'assemblage contient une directive **DB**.

Syntaxe

LABEL: .DB expression list

Example

```
            .CSEG
consts:     .DB 0, 255, 0b01010101, -128, $AA
            .ESEG
eeconst:    .DB $FF
```

DEF - Set a symbolic name on a register - Définit un nom symbolique pour un registre

La directive **DEF** permet aux registres d'être mentionnée par des symboles. Un symbole ainsi défini peut être employé dans le reste du programme pour se référer au registre auquel il est assigné. Un registre peut prendre plusieurs noms symboliques. Un symbole peut être redéfini plus tard dans le programme.

Syntaxe

.DEF Symbol = Registre

Exemple

```
            .DEF    temp = R16
            .DEF    ior = R0
            .CSEG
            ldi     temp, $F0       ; Charge $F0 dans temp (r16)
            in      ior, $3F        ; Lire SREG dans ior (r0)
            eor     temp, ior       ; Ou exclusif entre temp et ior
```

DEVICE – Define which device to assemble for - Définit le modèle pour l'assemblage

La directive **DEVICE** permet à l'utilisateur de signaler à l'assembleur sur lequel modèle le code doit être exécuté. Si cette directive est employée, un avertissement est affiché si une instruction n'est pas soutenue par le modèle indiqué. Si la taille du segment de code ou le segment **d'EEPROM** est plus grand que l'espace du modèle spécifié, un avertissement est affiché. Si la directive **DEVICE** n'est pas employée, aucun message ne sera affiché et toutes les instructions sont valides, il n'y a aucune restriction de tailles mémoires.

Syntaxe

.DEVICE ATMEGA32

Exemple

```
.DEVICE ATMEGA32      ; Utilise le modèle ATMEGA32
.CSEG
push    r30           ; Place le registre r30 sur la pile
```

DSEG – Data Segment - Segment de Données

La directive **DSEG** définit le début d'un segment de données. Un fichier assembleur peut contenir plusieurs segments de données, qui sont enchaînés dans un seul segment de données dans le code assemblé. Un segment de données seulement contiendra normalement la directive **BYTE** (et les étiquettes).

Les segments de données ont leur propre compteur d'emplacement qui est un compteur d'octet. La directive **ORG** (voir la description plus tard dans ce chapitre) peut être employée pour positionner les variables dans la **SRAM**. La directive ne prend pas de paramètres.

Syntax

.DSEG

Example

```
            .DSEG                ; Début du segment de donnée
var1:       .BYTE 1              ; Réserve 1 octet à var1
table:      .BYTE tab_size       ; Réserve tab_size octets
            .CSEG
            ldi     r30, low(var1)   ; Charge registre Z bas
            ldi     r31, high(var1)  ; Charge registre Z haut
            ld      r1, Z            ; Charge var1 avec r1
```

DW - Define constant word(s) in program memory or EEPROM memory - Définit le(s) mot(s) constant

La directive **DW** réserve des ressources de mémoire dans la mémoire programme ou la mémoire **EEPROM**. Pour être capable de se référer aux emplacements réservés, la directive **DW** doit être précédée par une étiquette. La directive **DW** prend une liste d'expressions et doit contenir au moins une expression. La directive **DB** doit être placée dans un segment de code ou un segment **d'EEPROM**.

La liste d'expression est un ordre d'expressions, délimitées par des virgules. Chaque expression doit évaluer un nombre entre -32768 et 65535. Si l'expression évalue un nombre négatif, le complément à 2 des 16 bits du nombre sera placé dans l'emplacement mémoire programme.

Syntaxe

LABEL: .DW expression list

Exemple

```
            .CSEG
varlist:    .DW    0, $FFFF, 0b1001110001010101, -32768, 65535
            .ESEG
eevar:      .DW    $FFFF
```

ENDMACRO – End macro - Fin macro

La directive **ENDMACRO** définit la fin d'une définition de macro. La directive ne prend pas de paramètres. Voir la directive **MACRO** pour plus d'information sur les macros.

Syntax

.ENDMACRO

Exemple

```
.MACRO    SUBI16              ; Début de définition d'une macro
          subi    r16, low(@0)  ; Soustraction bas
          sbci    r17, high(@0) ; Soustraction haut
.ENDMACRO                     ; Fin de la définition de la macro
```

EQU - Set a symbol equal to an expression - Attribut un symbole à une expression

La directive **EQU** assigne une valeur à une étiquette. Cette étiquette peut alors être employée dans des expressions postérieures. Une étiquette assignée à une valeur selon la directive **EQU** est une constante et ne peut pas être changée ou redéfinie.

Syntax

.EQU label = expression

Example

```
          .EQU    io_offset = 0x23
          .EQU    porta = io_offset + 2
          .CSEG                 ; Début du segment
          clr     r2            ; Efface r2
          out     porta, r2     ; Ecrir le Port A
```

ESEG – EEPROM Segment - Segment EEPROM

La directive **ESEG** définit le début d'un segment **d'EEPROM**. Un fichier assembleur peut consister en plusieurs segments **EEPROM**, qui sont enchaînés dans un seul segment **d'EEPROM** après l'assemblage. La directive **BYTE** ne peut pas être employée dans un segment **d'EEPROM**. Les segments **d'EEPROM** ont leur propre compteur d'emplacement qui est un compteur d'octet. La directive **ORG** (voir la description plus tard dans ce chapitre) peut être employée pour positionner le segment dans la mémoire **EEPROM**. La directive ne prend pas de paramètres.

Syntax

.ESEG

Example

```
          .DSEG                 ; Début de segment de donnée
vartab:   .BYTE 4               ; Réserve 4 octets dans la SRAM
          .ESEG
eevar:    .DW $FF0F             ; Initialise un mot dans l'EEPROM
          .CSEG                 ; Début du segment de code
const:    .DW   2               ; Ecrit $0002 dans la mémoire programme
          mov   r1, r0          ; Début du programme
```

EXIT - Exit this file - Sortie du fichier

La directive **EXIT** dit à l'assembleur d'arrêter d'assembler le fichier. Normalement, l'assembleur court jusqu'à la fin du fichier **EOF**. Si une directive **EXIT** apparaît dans un fichier inclus, l'assembleur continu à la ligne après la directive **INCLUDE** dans le fichier contenant la directive **INCLUDE**.

Syntaxe

.EXIT

Exemple

```
                .EXIT                   ; Sortie du fichier
```

INCLUDE – Include another file - Importe la source d'un autre fichier

La directive **INCLUDE** dit à l'assembleur de commencer à lire le fichier indiqué. L'assembleur assemble alors le fichier indiqué avant que l'on ne rencontre la fin du fichier **EOF** ou une directive **EXIT**. Un fichier inclus peut contenir aussi des directives **INCLUDE**.

Syntax

.INCLUDE "filename"

Example

```
; iodefs.asm:
                .EQU   sreg = $3F       ; Registre Statut
                .EQU   sphigh = $3E     ; Pointeur de Pile haut
                .EQU   splow = $3D      ; Pointeur de Pile bas
                …
; incdemo.asm
                .INCLUDE "iodefs.asm"   ; Inclus I/O definitions
                in      r0, sreg        ; Lire le registre statut
```

LIST - Turn the listfile generation on - Met en route la génération de listfile

La directive **LIST** dit à l'assembleur de mettre en route la génération **listfile**. L'assembleur produit un **listfile** qui est une combinaison de code source assemblé, des adresses et des **opcodes**. La génération **listfile** est active par défaut. La directive peut aussi être employée avec la directive **NOLIST** pour produire seulement des parties choisies d'un fichier de source d'assemblé.

Syntaxe

.LIST

Exemple

```
                .NOLIST                 ; Désactive la génération listfile
                .INCLUDE "macro.inc"    ; Inclus un fichier macro
                .INCLUDE "const.def"    ; Non visible dans listfile
                .LIST                   ; Active la génération listfile
```

LISTMAC – Turn macro expansion on - Met en route l'extension macro

La directive **LISTMAC** dit à l'assembleur quand une macro est appelée, l'extension d'une macro doit être montrée sur **listfile** produit par l'assembleur. Par défaut, on montre la macro appelée avec les paramètres dans le fichier **listfile**.

Syntax

.LISTMAC

Example

```
.MACRO          MACX                    ; Définition d'un exemple de macro
                add     r0, @0
                eor     r1, @1
.ENDMACRO                               ; Fin de la définition de la macro
                .LISTMAC                ; Active l'extension macro
                MACX r2, r1             ; Appel de la macro
```

MACRO – Begin macro - Commence une macro

La directive **MACRO** dit à l'assembleur que c'est le début d'une macro. La directive **MACRO** prend le nom « macro » comme paramètre. Quand le nom de la macro est écrit plus tard dans le programme, la définition macro est employée à la place. Une macro peut prendre jusqu'à 10 paramètres. Ces paramètres sont mentionnés comme **@0-9** dans la définition de la macro. En publiant un appel de macro, on donne les paramètres séparés par une virgule dans la liste. La définition de la macro est terminée avec la directive **ENDMACRO**. Par défaut, on montre l'appel des macros sur le fichier **listfile** produit par l'assembleur.

Pour inclure l'extension macro dans le fichier **listfile**, la directive **LISTMAC** doit être employée. Une macro est marquée avec un symbole **+** dans le champ **opcode** du fichier **listfile**.

Syntax

.MACRO macroname

Example

```
.MACRO      SUBI16                  ; Début de la définition de la macro
            subi    @1, low(@0)     ; Soustraction bas
            sbci    @2, high(@0)    ; Soustraction haut
.ENDMACRO                           ; Fin de la macro
            .CSEG                   ; Début du segment de code
            SUBI16 $1234, r16, r17  ; Soustraction $1234 à r17:r16
```

NOLIST - Turn listfile generation off - Arrêt de la génération de listfile

La directive **NOLIST** dit à l'assembleur d'arrêter la génération du fichier **listfile**. L'assembleur produit normalement un fichier **listfile** qui est une combinaison du code source d'assemblé, des adresses et des **opcodes**. La génération du fichier **listfile** est active par défaut, mais peut être mis hors de service en employant cette directive. La directive peut aussi être employée avec la directive **LIST** pour produire seulement des parties choisies d'un fichier de source d'assemblé.

Syntax

.NOLIST ; Désactivation de listfile

Example

```
            .NOLIST                 ; Désactivation de listfile
            .INCLUDE "macro.inc"    ; Inclus un fichier macro
            .INCLUDE "const.def"    ; non visible dans listfile
            .LIST                   ; Active la génération listfile
```

ORG - Set program origin - Origine du programme en mémoire

La directive **ORG** définit le compteur d'emplacement à une valeur absolue. On donne la valeur comme un paramètre de la directive. Si une directive **ORG** est placée dans un segment de données, c'est le compteur d'emplacement **SRAM** qui est mis, si la directive est dans un segment de code, c'est le compteur de mémoire programme qui est mis et si la directive est dans un segment **d'EEPROM**, c'est le compteur d'emplacement **EEPROM** qui est mis. Si la directive est précédée par une étiquette (sur la même ligne de code source), on donnera à l'étiquette la valeur du paramètre. Les valeurs par défaut du code et des compteurs d'emplacement **EEPROM** sont à zéro, tandis que la valeur par défaut du compteur d'emplacement **SRAM** est de 32 (due aux registres rapides occupant les adresses 0 à 31). L'**EEPROM** et l'emplacement **SRAM** sont en octet tandis que l'emplacement de mémoire programme est en mots à 16 bits.

Syntax

.ORG expression

Example

```
            .DSEG                   ; Début du segment de donnée
            .ORG   $67              ; SRAM à l'adresse $67
variable:   .BYTE 1                 ; Réserve 1 octet dans la SRAM
```

```
; adr.$67
                .ESEG                   ; Début du segment d'EEPROM
                .ORG  $20               ; Localisation de l'EEPROM
; compteur
eevar:          .DW   FEFF              ; Initialise un mot
                .CSEG
                .ORG  $10               ; Compteur de Programme (PC)
; 10
                mov   r0, r1
```

SET - Set a symbol equal to an expression - Déclare un symbole pour une expression

La directive **SET** assigne une valeur à une étiquette. Cette étiquette peut alors être employée dans des expressions postérieures. Une étiquette assignée à une valeur selon la directive **SET** peut être changée plus tard dans le programme.

Syntax

.SET label = expression

Example

```
.SET    io_offset = $23
.SET    porta = io_offset + 2
.CSEG                   ; Début du segment de code
clr     r2              ; Efface r2
out     porta, r2       ; Ecrit dans le Port A
```

Les Expressions de l'Assembleur

L'assembleur incorpore des expressions. Les expressions peuvent consister en des opérandes, des opérateurs et des fonctions. En voici le descriptif complet.

Opérandes

Les opérandes suivants peuvent être employés :

- L'utilisateur a défini les étiquettes que l'on donne à la valeur du compteur d'emplacement à la place où ils apparaissent.
- L'utilisateur a défini des variables selon la directive de **SET**
- L'utilisateur a défini des constantes selon la directive **EQU**
- Entier constant : selon plusieurs formats, incluant :
  a) - (Défaut) décimal : 10, 255
  b) - Hexadécimal (deux notations) : 0x0A, $0A, 0xFF, $FF
  c) - Valeur binaire : 0b00001010, 0b11111111
- PC - la valeur actuelle du compteur programme en mémoire programme

Fonctions

Les fonctions suivantes sont définies :

- LOW (l'expression) rend l'octet bas d'une expression
- HIGH (expression) rend le deuxième octet d'une expression
- BYTE2 (l'expression) est la même fonction que HIGH
- BYTE3 (l'expression) rend le troisième octet d'une expression
- BYTE4 (l'expression) rend le quatrième octet d'une expression
- LWRD (l'expression) rend les particules 0-15 d'une expression
- HWRD (l'expression) rend les particules 16-31 d'une expression
- PAGE (l'expression) rend les particules 16-21 d'une expression
- EXP2 (l'expression) retourne 2 exposant
- LOG2 (l'expression) rend la partie d'entier (d'expression) log2

Opérateurs

L'assembleur soutient quelques opérateurs qui sont décrits ici. Les expressions peuvent être incluses dans des parenthèses et telles expressions sont toujours évaluées avant d'être combiné avec les valeurs à l'extérieur des parenthèses.

Unaire Symbole : !

L'opérateur Unaire retourne 1 si l'expression était zéro et retourne 0 si l'expression était non zéro :

```
Exemple :   Ldi   r16, !$F0          ; Charge r16 avec $00
```

Négation Symbole : ~

L'opérateur Négation rend l'expression d'entrée avec tous les bits inversés :

```
Exemple :   Ldi   r16, ~$F0          ; Charge r16 avec $0F
```

Moins Symbole : -

L'opérateur Moins rend la négation arithmétique d'une expression :

```
Exemple :   Ldi   r16, -2            ; Charge –2 ($FE) dans r16
```

Multiplication Symbole : *

L'opérateur Multiplication binaire que rend le produit de deux expressions :

```
Exemple :   Ldi   r30, label * 2     ; Charge r30 avec label multiplier par 2
```

Division Symbole : /

L'opérateur Division binaire rend le quotient entier de l'expression gauche divisée par l'expression juste :

```
Exemple:    Ldi   r30, label / 2     ; Charge r30 avec label diviser par 2
```

Addition Symbole : +

L'opérateur Addition binaire rend la somme de deux expressions :

```
Exemple :   Ldi   r30, c1 + c2       ; Charge r30 avec c1 plus c2
```

Soustraction Symbole : -

L'opérateur Soustraction binaire rend l'expression gauche moins l'expression juste

```
Exemple :   Ldi   r17, c1 - c2       ; Charge r17 avec c1 moins c2
```

Décalage Gauche Symbole : <<

L'opérateur Décalage à Gauche binaire rend l'expression gauche décalée à gauche plusieurs fois fonction de l'expression juste :

```
Exemple :   Ldi   r17, 1<<bitmask    ; Charge r17 avec 1 décalé à gauche de bitmask fois
```

Décalage Droit Symbole: >>

L'opérateur Décalage à Droite binaire rend l'expression gauche décalée à droite plusieurs fois en fonction de l'expression juste :

```
Exemple :   Ldi   r17, c1>>c2        ; Charge r17 avec c1 décalé à droite de c2 fois
```

Plus Petit Que Symbole : <

L'opérateur Plus Petit Que binaire retourne 1 si l'expression signée à gauche est moins grande que l'expression signée de droit, 0 autrement :

```
Exemple :   Ori   r18, bitmask * (c1<c2) + 1 ; OU avec r18
```

Plus Petit ou Egale Symbole : <=

L'opérateur Plus Petit ou Egale binaire retourne 1 si l'expression signée à gauche est moins grande ou Égale à l'expression signée de droit, 0 autrement :

```
Exemple :   Ori   r18, bitmask * (c1<=c2) + 1      ; OU avec r18
```

Plus Grand Que Symbole : >

L'opérateur Plus Grand Que binaire retourne 1 si l'expression signée à gauche est plus grande que l'expression signée à droit, 0 autrement :

```
Exemple :   Ori   r18, bitmask * (c1>c2) + 1 ; OU avec r18
```

Plus Grand ou Egale Symbole : >=

L'opérateur Plus Grand ou Egale binaire retourne 1 si l'expression signée à gauche est plus grande ou Égale à l'expression signée à droit, 0 autrement :

```
Exemple :   Ori   r18, bitmask * (c1>=c2) + 1 ; OU sur r18
```

Egale Symbole : ==

L'opérateur Egale binaire retourne 1 si l'expression signée à gauche est égale à l'expression signée à droit, 0 autrement :

```
Exemple :   Andi  r19, bitmask * (c1==c2)+1 ; ET sur r19
```

Non Egale Symbole : !=

L'opérateur Non Egale binaire retourne 1 si l'expression signée à gauche n'est pas égale à l'expression signée à droit, 0 autrement :

```
Exemple :   .SET  flag = (c1!=c2)      ; Met le flag avec 1 ou 0
```

ET entre valeur Symbole : &

L'opérateur ET binaire rend le test ET entre deux expressions :

```
Exemple :   Ldi   r18, High(c1&c2)   ; Charge r18 avec un ET logique
```

OU Exclusif Logique Symbole : ^

L'opérateur OU Exclusif binaire retourne le test d'un OU Exclusif entre deux expressions :

```
Exemple :   ldi r18,Low(c1^c2)          ;Charge r18 avec un ou exclusif
```

OU entre deux valeurs Symbole : |

L'opérateur OU binaire rend le test OU entre deux expressions :

```
Exemple :   ldi   r18, Low(c1|c2)      ; Charge 18 avec un OU logique
```

ET Logique Symbole : &&

L'opérateur ET Logique binaire retourne 1 si les expressions sont tous les deux différent de zéro, 0 autrement :

```
Exemple :   ldi   r18, Low(c1&&c2)  ; Charge r18 avec un ET logique
```

OU Logique Symbole : ||

L'opérateur OU Logique binaire retourne 1 si un ou les deux expressions sont différentes de zéro, 0 autrement

```
Exemple :   ldi   r18, Low(c1||c2)      ; Charge r18 avec un OU logique
```

Conclusion

La programmation en assembleur n'est pas plus difficile que le Basic ou le langage C, le contrôle complet du microcontrôleur est par contre total avec ce langage, car vous avez accès à tout les éléments du processeur sans aucune limitation, ce qui peut poser quelques problèmes si vous ne maîtriser pas correctement le fonctionnement interne de la bête.

Ceci étant, avec un peu d'expérience, vous passerez vite à la réalisation de programme complexe et rapide, car l'autre avantage de l'assembleur, c'est que le programme est optimisé par vous, contrairement au basic ou au langage C qui utilise plus de mémoire et plus d'instruction pour faire la même chose.

J'espère que vous avez apprécié ce travail et qu'il vous à servit avec bonheur, bonne programmation à vous et à bientôt.

Jean-Noël

Index