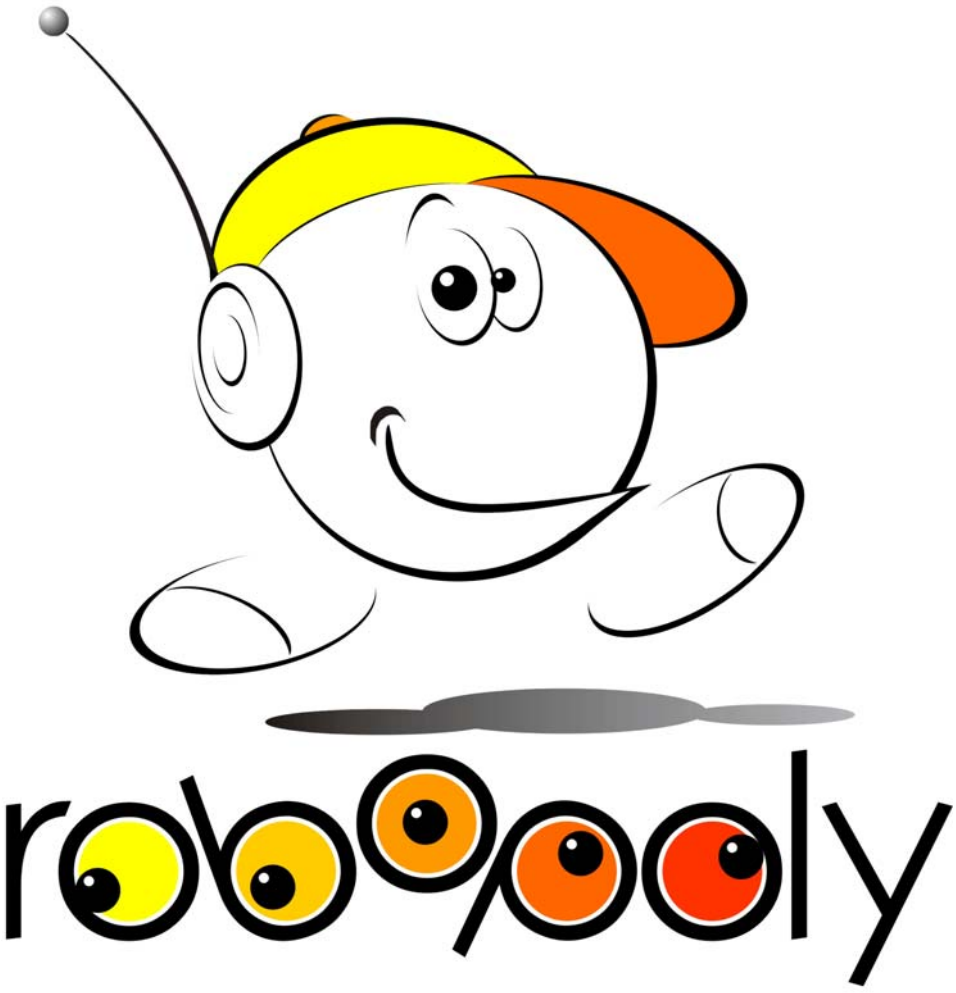
Guide de Programmation en C



Version : 1.0.0
24 octobre 2007

Comité Robopoly (2007-2008)
Ecrit par Fabrizio Lo Conte

Table des matières

1 Introduction

Le microcontrôleur est un composant électronique très largement utilisé dans le monde industriel, académique ou "hobbystique". Ses principales qualités sont celles d'être facilement programmables, petits, peu coûteux et enfin disponible chez tout revendeur de composant électroniques. Il est présents dans de nombreux appareil que vous côtoyez quotidiennement, frigo, four, vitrocéramique, baladeur, téléphone, dans l'automobile, afficheur lumineux dans le TSOL, etc. etc. etc.

Plus précisément le microcontrôleur est utilisé pour afficher des choses, effectuer des calculs ou des opérations en fonction d'une certaine situation, communiquer avec un ordinateur. Et c'est exactement ce que nous vous proposons de faire sur votre robot.

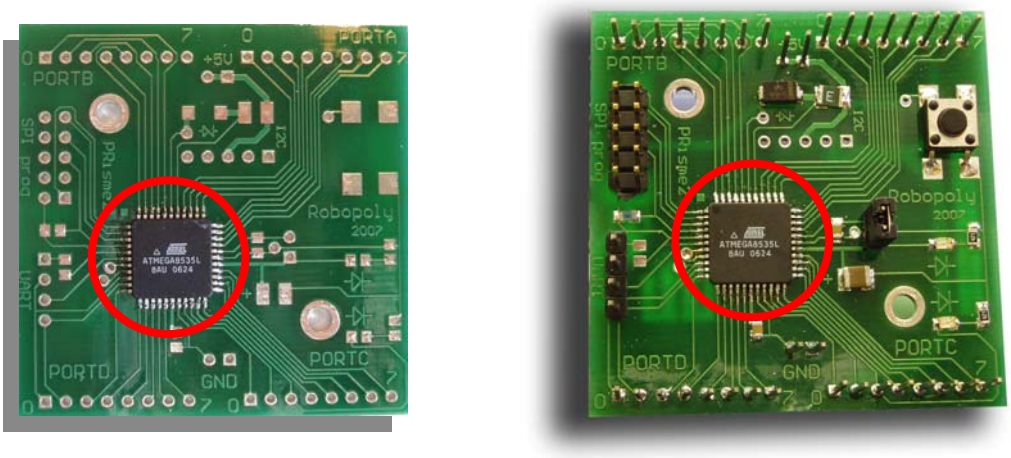


Figure 1 Cerveau du PRisme2[*], encerclé le microcontrôleur, à gauche le cerveau que vous avez reçu, à droite entièrement monté.

Le fonctionnement d'un microcontrôleur est relativement simple, dans notre cas, il possède 4 ports d'entrée/sortie, chaque port ayant 8 pins ou bits. Un port est une interface électronique permettant de faire entrée ou sortir un signal du microcontrôleur, un port est un groupement de 8 pins. Chaque pin est complètement indépendant. Les ports sont nommés (comme vous pouvez le voir sur la Figure 1) PORTA, PORTB, PORTC, PORTD. Les pins sont quant à eux numéroté de 0 à 7. Souvent lorsque nous parlerons du pin #3 du PORTA par exemple nous le noterons PORTA3.

Vous pouvez donc brancher vos moteurs (cf. guide de montage, 2 fils par moteur) sur le PORTD3, PORTD4, PORTD5, PORTD6. Ainsi votre robot, avancera, reculera, tournera en fonction des valeurs que vous écrirez sur ces pins. De même pour les capteurs, en branchant un capteur sur le pin PORTA3, nous pourrons utiliser cette information durant l'exécution du programme. Par exemple lorsque le capteur voit un obstacle le microcontrôleur pourra soit arrêter les moteurs, soit les faire tourner pour éviter cet obstacle.

[*] La partie carrée verte est ce qui s'appelle le PCB (Printed Circuit Board). Il est nommé circuit imprimé car des pistes sont "imprimées" sur une plaque en plastique. On les remarque ici, les pistes sont, les lignes étroites vertes, les textes visibles ainsi que les parties grise métallisées. Les pistes sont en faites simplement des de très fine couche de cuivres. Sur ce PCB divers devront être soudés, pour plus de détail sur leur fonction, référerez vous aux documents que vous trouvez sur le site http://robopoly.epfl.ch

De plus si par exemple l'écran LCD est branché sur le PORTB, il pourra afficher un message pour vous informez de son état ou vous demandez de faire quelque chose, en fonction de ce que vous lui aurez programmé.

Nous vous montrerons dans les pages suivantes, ainsi que dans les divers tutoriaux présents sur le site web que cette opération d'évitement d'obstacle ne nécessite pas plus de 10 lignes de code. Et afficher un texte sur l'écran ne vous demandera que quelques lignes supplémentaires!

Ligne de code les mots sont lancés, qu'est ce que c'est?

1.1 Principe de fonctionnement

Nous l'avons tout juste dit, le microcontrôleur exécute simplement votre programme. Un programme se compose de ligne de code. Ce code peut être écrit dans différents langages, celui que nous vous présenterons dans ce document est le C. Qui est un langage très simple et permettant de réaliser dans la plus part des cas ce que nous souhaitons.

Le microcontrôleur exécute un code, nommé code machine, il se compose uniquement de 1 et de 0. Cette façon de programmer étant comme vous pouvez l'imaginez inutilisable deux type de langage ont été mis en place, ceux de bas niveau et ceux de haut niveau.

Chaque microcontrôleur possède un certain nombre d'opération de base (celui-ci 130) qu'il est capable d'exécuter. Par exemple, additionner 2 nombres, lire un port, écrire sur un port, etc. etc. Chacune de ces opérations est décrite dans les spécifications techniques (en anglais *datasheet* du composant), souvent ces opérations de base sont appelées "instruction". Lors de l'exécution de son code, le microcontrôleur exécutera les instructions les unes après les autres.

1.1.1 Langage bas niveau

Le langage assembleur permet d'écrire le code qu'exécutera le microcontrôleur en utilisant les 130 instructions de base. Ce langage est donc d'une puissance extrême car il permet de contrôler instruction après instruction ce qui se passe dans le microcontrôleur, et d'accéder à tout les registres et espace mémoire que le microcontrôleur possède.

Son principal désavantage, est qu'il étroitement lié à la plate forme sur laquelle le code devra être exécuté. Un code assembleur écrit pour un microcontrôleur Atmel ne pourra absolument pas être exécuté par un microcontrôleur PIC[*]. De plus de part sa syntaxe il n'est parfois pas intuitif de le comprendre.

Sa syntaxe est la suivante: chaque ligne comporte une instruction de base, suivi de 2 variables. Par exemple l'instruction ci-dessous effectue l'opération r1+r2 et stocke le résultat dans r1.

```
add r1,r2
```

[*] Atmel et PIC sont des fabricants de microcontrôleur très connu sur le marché.

Le code s'écrit donc ligne par ligne dans un fichier texte. Nous vous l'avons tout juste mentionné ce code est absolument incompréhensible dans l'état par le microcontrôleur, il faut absolument le traduire en 1 et 0. Le langage assembleur utilisant déjà les instructions de base connue par le microcontrôleur, la traduction est très simple. Il suffira de remplacé l'instruction ainsi que les 2 opérateurs par leurs codes binaires.

Pour d'avantage de détail concernant ce langage de programmation reportez vous au document "Guide de programmation en Assembleur" que vous trouverez sur le site web.

1.1.2 Langage de haut niveau

Ces langages sont la plus part du temps des langages de programmation classique, et utilisé quelque soit la plateforme sur laquelle ils seront exécutés. Leurs syntaxe est souvent très simple et intuitive.

Nous n'entrerons pas dans ce documents dans les détails du langage C, car ceci sort du but pour lequel se document a été écrit. Des exemples ainsi que des commentaires détaillés illustreront ce document.

A nouveau un code écrit en C n'est absolument pas exécutable par un microcontrôleur, pour 2 raisons. Tout d'abord le code n'est pas écrit en 1 et en 0. De plus il n'utilise pas les instructions de base que le microcontrôleur est capable d'exécuter. Il faut donc littéralement traduire le code C en n'utilisant que les instructions de base du microcontrôleur. C'est le rôle de l'outil nommée compilateur. Il lit le code C que vous avez écrit et le traduit en code assembleur! Puis un autre outil se chargera de remplacer tout les instructions de bases et opérateur par leur code binaires.

Bien évidemment le code assembleur générer par le compilateur est moins compact que si vous l'aviez écrit par vous-même en assembleur. Mais dans 95% des cas ceci n'est absolument pas dérangeant et le fait d'écrire votre code en C vous permet de gagné considérablement de temps de développement.

2 Environnement de développement

Pour qu'un code écrit en C puisse être compilé, c'est-à-dire traduit en instructions simples que le microcontrôleur pourra interpréter, il faut ajouter quelques éléments à l'environnement de développement utilisé. Tout d'abord, bien évidemment, un compilateur C est nécessaire, celui-ci serait pratiquement inutilisable sans l'ajout des librairies standards (conversion de type, utilisation de string, opérateur mathématique, etc.) et d'un débuggeur C.

Dans ce guide, ainsi qu'au sein du club, nous vous proposons d'utiliser un "*package*" contenant tout les éléments nécessaires à la programmation en C de la plupart des microcontrôleurs AVR 8 bits-RISC de la maison Atmel.

Ce package est connu sous le nom de " WinAvr ", il est composé du fameux compilateur GCC. Le nom de la librairie C que nous utiliserons est " avr-libc ".

WinAvr tout comme Avr Studio sont en constante évolution. C'est pourquoi parfois il se peut que des incompatibilités entre les différentes versions de ces 2 programmes surviennent. Il est donc primordial de choisir des versions compatibles lorsque vous installez ces deux logiciels.

2.1 Téléchargement et installation

Tout d'abord, il faut que Avr Studio soit installé sur votre PC, un lien est présent sur le site de Robopoly, sinon une simple recherche sur le site d'Atmel vous permettra de trouver ce software.

Puis WinAvr doit être installé, vous le trouverez à nouveau sur le site de Robopoly, ou alors en suivant le lien download depuis le site de SourceFORGE (http://winavr.sourceforge.net/).

Les versions utilisées lors de la rédaction de ce document sont:

Avr Studio 4.13 Build 528
WinAvr 20070122

Maintenant que l'environnement de développement est installé vous pouvez écrire votre premier programme!

3 Programmation

3.1 Début d'un projet

Démarrer AVR Studio 4 et créer un nouveau projet, en fonction de vos préférences cliquer sur *New Project* dans l'écran de bienvenu (Figure 2) ou alors *Project->New Project* depuis le menu.

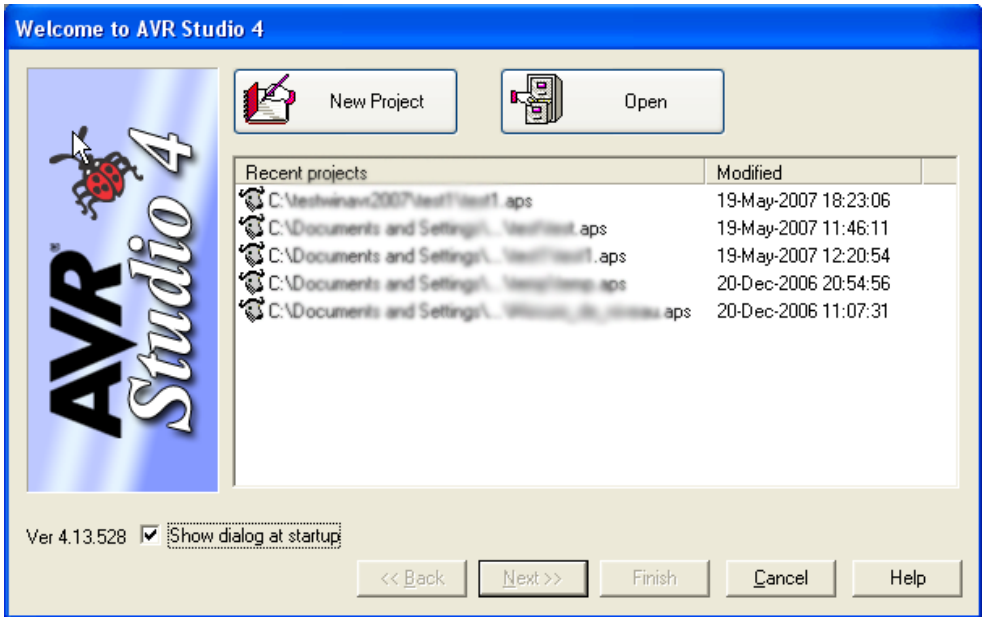


Figure 2 Ecran de bienvenu

Dans la boîte de dialogue "*New Project*" (Figure 3), deux types de projets vous sont proposés.

- Atmel AVR Assembler, qui comme son nom l'indique permet d'écrire des programmes en assembleur.
- AVR GCC, qui utilisera GCC pour compiler votre code, ceci vous permettra donc d'écrire votre code en C.

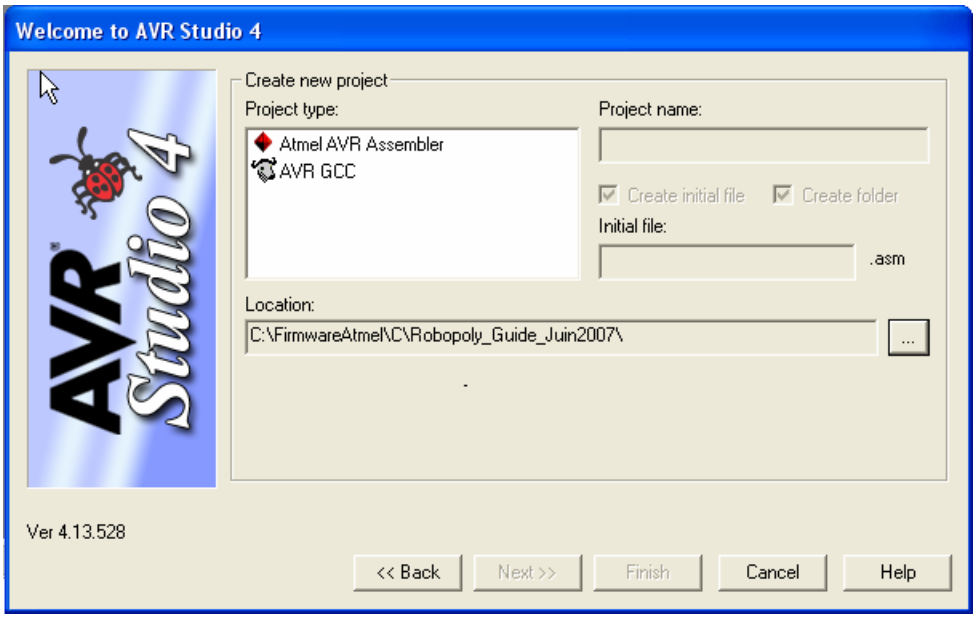


Figure 3 Boîte

Sélectionner le projet AVR GCC, puis compléter le champ *Project name*. Le nom du fichier initial est, par défaut, identique à celui du projet. Vous avez cependant la possibilité de le modifier.

Puis cliquer sur *Next*, une nouvelle boîte de dialogue apparaît, cette dernière vous permet de sélectionner le débuggeur ainsi que le microcontrôleur à utiliser.

Dans notre cas, nous utiliserons la D*ebug platform* " AVR Simulator " et le *Device* Atmega8535. Une fois la sélection faite, cliquer sur *Finish*.

L'environnement de "travail" apparaît (Figure 4), à gauche se trouve l'arborescence des fichiers utilisés dans votre projet, alors que sur la droite sont présentés les différents éléments du microcontrôleur.

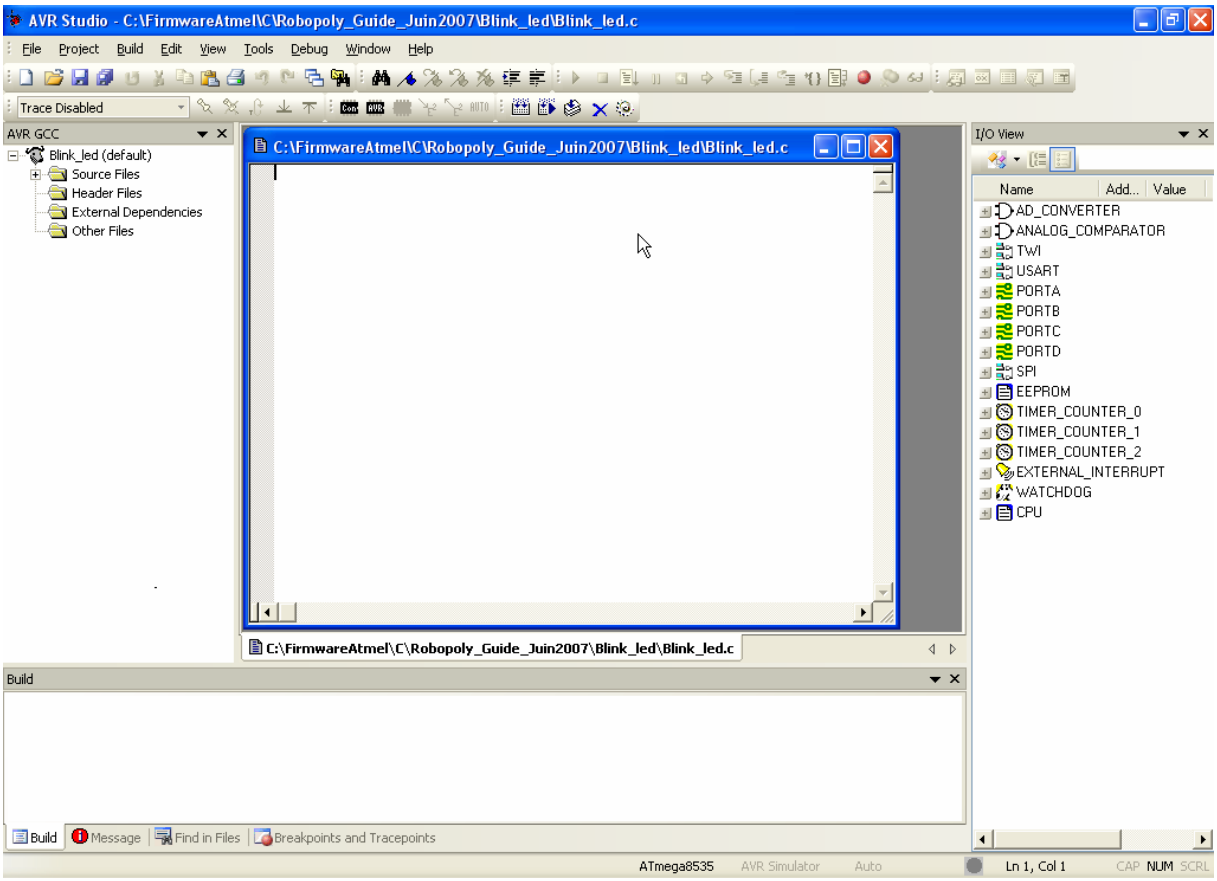


Figure 4 Environnement de développement

3.2 Base de la programmation en C

3.2.1 Structure de base du code

Les lignes qui suivent montrent comment utiliser le C pour la programmation d'un microcontrôleur. Nous supposons ici que le lecteur connaisse les bases du langage C.

Au sein de la librairie avr-libc de nombreux *include* sont disponibles, permettant de générer des délais, utiliser des interruptions, effectuer des opérations mathématiques, traiter des chaînes de caractères, etc… Une description détaillée de ces divers *include* se trouve:

- Depuis les menus de Avr Studio 4 sous *Help->avr-libc Reference Manual*
- Sur Internet à l'adresse http://www.nongnu.org/avr-libc/user-manual/

Pour pouvoir utiliser ces fonctions, il faudra auparavant avoir inclus à son code l'*include* correspondant.

Pour un bon fonctionnement du programme, un *include* est indispensable. Il faut en effet définir les divers éléments propres au microcontrôleur, ainsi que les différents types de variable que le C utilise. Cet *include* est le fichier io.h*

Voici donc la structure de base d'un code en C pour microcontrôleur Atmel

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

int main(void)
{
    //Début du Programme!

    return 0;
}
```

Code 1 Structure de base d'un code C

(Les lignes ci-dessus ont été volontairement incluses en tant qu'image car, nous pensons que le fait de devoir les réécrire vous permettra de les mémoriser.)

3.2.2 Premier Build

Après avoir écrit ces quelques lignes dans le fichier principal, nous pouvons effectuer le premier *build* du projet, depuis les menus *Build->Build* ou en cliquant sur le bouton encerclé en rouge (Figure 5).

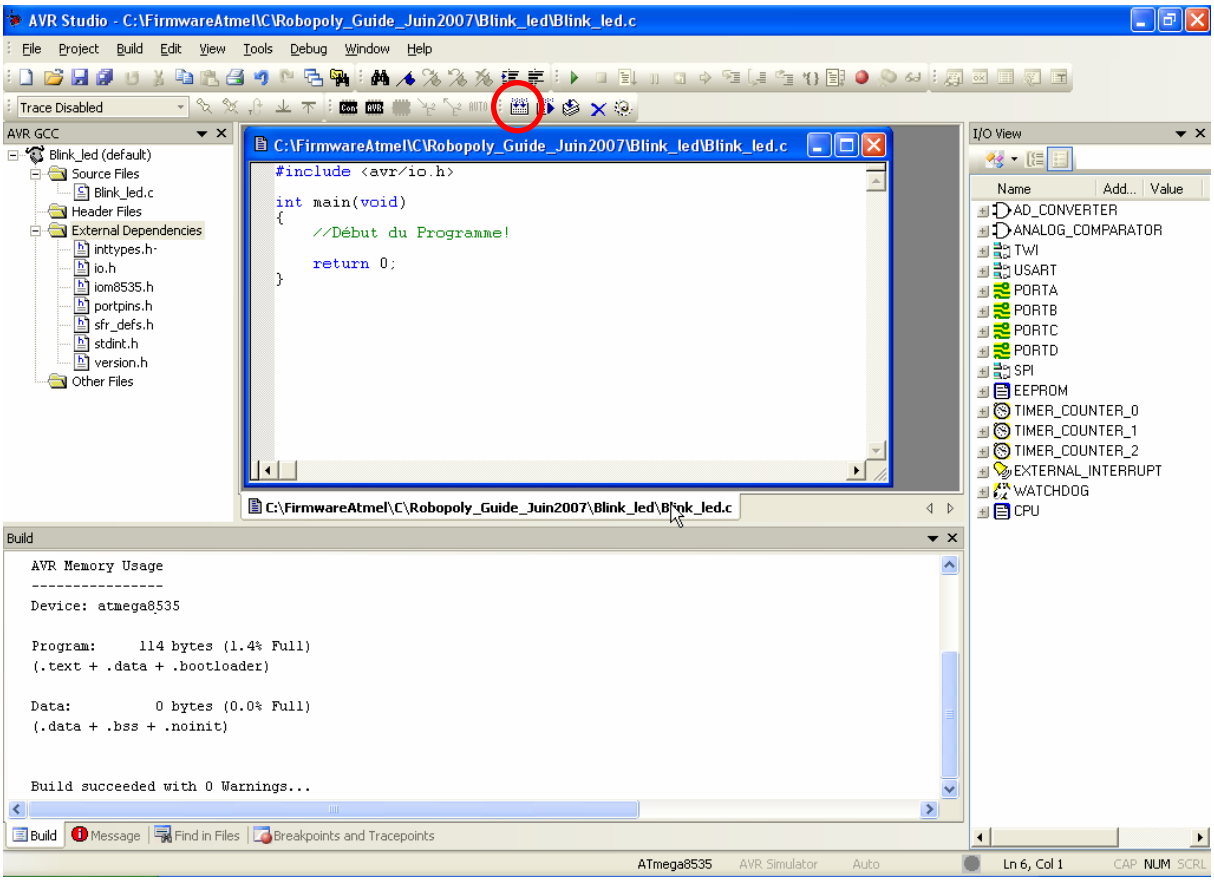


Figure 5 Premier Build

*Fichier contenant tous les noms et adresses des registres nécessaire au fonctionnent du microcontrôleur.

De nombreuses informations sont maintenant contenues dans l'environnement de développement.

Remarquons que le fichier io.h que nous avons inclus apparaît dans les *External Dependencies*. De nombreux autre fichiers ont aussi été inclus, voici pour les lecteurs les plus curieux une très brève description des principaux fichiers.

- stdint.h : contient la définition des différents types de variables entières propres au C (int, long, short, etc.)
- inttype.h : contient une extension de définition de type, surtout utilisée avec les fonctions *printf()* et *scanf()*.
- iom8535.h : contient la définition des adresses mémoire de tous les registres du microcontrôleur utilisé (ici le Atmega8535)
- portpins.h : fait la relation entre nom du pin et son numéro de bits au sein du registre (p.ex _PORTB2 est le bit 2 dans le registre PORTB)

La fenêtre *Build* nous informe sur la taille de notre code (*Program)*, ainsi que la taille de mémoire RAM que nos variables utilisent (*Data)*. Nous pouvons y lire, en pourcentage, l'espace libre qui nous reste dans le microcontrôleur.

ATTENTION: Lorsque le code devient important, très souvent lorsque des chaînes de caractères sont utilisées, il n'est pas rare de lire (Figure 6) un pourcentage plus grand que 100%, sans que ceci pose problème à la compilation. Bien évidemment un tel code fonctionnera au mieux de manière aléatoire ou alors pas du tout. Il est donc judicieux de contrôler de temps à autre la taille de votre code.

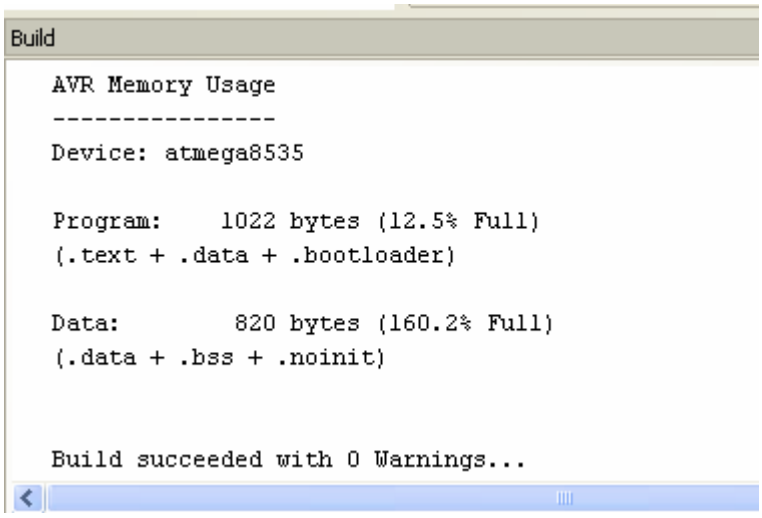


Figure 6 Sur-occupation de la mémoire

3.2.3 Description générale de quelques include

Inclure les fichiers suivants:

- stdlib.h : si des fonctions de conversion tels que string to long (strtol()), ascii to int (atoi()) doivent être utilisées.
- stdio.h : si des fonctions tels que scanf() ou printf() doivent être utilisées.
- string.h : si des opérations sur des chaînes de caractères doivent être faites.

- ctype.h : si des tests logiques sur des caractères doivent être utilisés (isdigit()).
- delay.h : si des délais en milliseconde, ou en microseconde doivent être effectués (pour d'avantage d'explication concernant cette librairie cf. ci-dessous).
- interrupt.h : si les interruptions sont utilisées. Voir section 3.3 pour d'avantage d'explications.
- math.h : si des opérations mathématiques comme cosinus, sinus, etc. sont utilisées (cf. ci-dessous pour des explications concernant la compilation).

3.2.4 delay.h

3.2.4.a Utilité des délais

Les fonctions contenues dans cet include permettent de faire attendre le microcontrôleur. En effet, de par sa conception interne, nous l'avons vu (cf. 1.1) les instructions, dont votre code est composé, seront exécutées les unes après les autres à la fréquence de fonctionnement de votre microcontrôleur.

Il est parfois nécessaire, après qu'une instruction ait été exécutée d'attendre un certain temps avant d'exécuter l'instruction suivante. L'exemple le plus simple est le clignotement d'une lumière, imaginons la structure de code suivante:

(Attention: les fonctions utilisées ici ne sont pas comprises par le compilateur, mais sont uniquement des exemples. Le Code 5 présenté à la section 3.2.4.c est un exemple de code compilable faisant clignoter le PortB)

```
while(1)
{
        ALLUMER ();
        ETEINDRE ();
 }
```

Code 2 Clignotement très rapide

Où ALLUMER (), et ETEINDRE () sont des fonctions qui permettent d'allumer et éteindre respectivement une lumière. Quel sera le résultat d'un tel code s'il est exécuté par un microcontrôleur (comme celui contenu dans le kit) travaillant à 8Mhz ?

Pour répondre à cette question, il suffit d'exécuter mentalement le code. La lumière s'allume puis l'instruction suivante* l'éteint, l'instruction d'après l'allume à nouveau. Or nous savons que le microcontrôleur exécute 8 millions d'instructions par seconde (car il a une fréquence de travail de 8Mhz). Ce qui en d'autres termes signifie que la lumière va s'allumer et s'éteindre 4 millions de fois par seconde! Notre œil n'est pas capable de voir plus de 24

* Pour simplifier le raisonnement nous supposons que le microcontrôleur soit capable d'exécuter l'appel de la fonction ALLUMER () ou ETEINDRE () en une seule instruction (un seul coup d'horloge). Ceci est faux, il lui faudra au moins une dizaine de cycle. Malgré cette petite approximation le raisonnement garde tout son sens.

images par seconde! Avec un tel code vous ne verrez donc pas la lumière clignoter, vous aurez l'impression qu'elle est constamment allumée![*]

Une fois l'instruction ALLUMER () exécutée il faut donc attendre un certain temps, avant d'exécuter l'instruction qui éteindra la lumière. Nous l'avons vu, il est pratiquement impossible d'arrêter le microcontrôleur pendant un certain temps, le moyen le plus simple, et le plus utilisé pour créer un délai entre une instruction et une autre, est de simplement faire autre chose pendant ce temps. On peut utiliser ce temps pour faire des calculs qui nous servirons plus tard, ou plus simplement faire compter le microcontrôleur jusqu'à une certaine valeur. On sait qu'il exécute une instruction toutes les 125ns (1/8Mhz = 125^{E}-9) donc par exemple en faisant compter le microcontrôleur de 0 à 10'000 par pas de 1 il lui faudra exécuter 10'000[**] instructions et donc 1.25ms vont s'écouler.

Imaginons une fonction ATTENDRE (100ms) qui permette, en utilisant la technique vue ci-dessus, d'attendre 100ms.

Modifions donc le code de la façon suivante:

```
while(1)
{
        ALLUMER ();
        ATTENDRE (100ms)
        ETEINDRE ();
 }
```

Code 3 Clignotement asymétrique

Vous l'aurez certainement remarqué, ce code ne fait toujours pas clignoter la lumière. Détaillons son fonctionnement, tout d'abord il allume la lumière, puis il attend 100ms, enfin il éteint la lumière! Très bien! Sauf que l'instruction suivante le microcontrôleur allume la lumière, notre œil n'aura donc pas eu le temps de voir la lumière s'éteindre. Apportons une dernière modification au code, qui devient donc fonctionnel. Il allume la lumière, attend 100ms, éteint la lumière, attend 100ms et allume à nouveau la lumière!

```
while(1)
{
        ALLUMER ();
        ATTENDRE (100ms)
        ETEINDRE ();
        ATTENDRE (100ms)
 }
```

Code 4 Clignotement correct

Voyons maintenant comment utiliser les fonctions contenues dans delay.h pour effectuer ces attentes.

[*] Petite question, un appareil électroménager que vous regarder souvent utilise exactement le même principe, qui est-ce? …. ….. C'est le téléviseur!

[**] On suppose à nouveau que pour compter le microcontrôleur n'emploi qu'une instruction. En réalité, il en emploiera plus d'une.

3.2.4.b Les fonctions de délais

Il y a deux fonctions qui permettent de réaliser un délai. _delay_ms (float time) et _delay_us (float time).

La première _delay_ms permet de réaliser des délais dont l'unité est la milliseconde alors que la seconde des délais en microseconde. La durée du délai dépend de la valeur du paramètre. Il suffira donc pour réaliser un délai de 2ms d'appeler la fonction _delay_ms (2). Alors que si l'on avait voulu réaliser un délai de 50us il aurait fallu appeler la fonction _delay_us (50).

Lorsqu'on utilise ces fonctions il faut faire attention à 2 choses:

- ***Le délai maximal***

  Nous l'avons vu, le principe utilisé par ces fonctions pour réaliser un délai, est de faire compter le microcontrôleur. Or bien évidemment le nombre de bit que le microcontrôleur va utiliser est un nombre fini, en d'autres termes, il ne pourra compter que jusqu'à une certaine valeur. Ce qui correspondra au délai le plus long que l'on peut réaliser par appel de fonction. Si l'on demande à la fonction de réaliser un délai plus grand, le résultat sera tout simplement aberrant et se traduira par un délai plus court dont la durée sera difficilement prévisible.

  La durée maximale que peut réaliser la fonction se trouve dans le fichier delay.h, et se résume par ces 2 phrases:

  _delay_ms:
  The maximal possible delay is 262.14 ms / F_CPU[*] in MHz
  _delay_us:
  The maximal possible delay is 768 us / F_CPU in MHz

  On remarque que le délai dépend de la fréquence de travail du microcontrôleur. Ce qui est bien évidemment normal. Il faut donc connaître à quelle fréquence travail le microcontrôleur. Par défaut, lorsque vous programmer la première fois votre prisme sur le programmateur de table, l'oscillateur interne au microcontrôleur est réglé sur 8Mhz.

  Le plus long délai que la fonction **_delay_ms** pourra réaliser est de:

  262.14/8 =32.7675 ms

  Uniquement les deux premiers chiffres après la virgule sont significatifs, le plus long délai réalisable avec une fréquence de 8Mhz est donc de 32.76 ms. Alors que le plus court est de 0.01 ms[†].

[*] Où F_CPU est la fréquence de travail du microcontrôleur. Dans notre cas, par défaut il est réglé à 8Mhz le délai maximal réalisable sera donc 262.14ms/8 ≈ 32.76ms. Idem pour la fonction delay_us le délai maximal sera de 768µs/8 = 96µs

[†] Bien sûr pour des questions de précision nous préférerons utiliser la fonction _delay_us(10) plutôt que _delay_ms(0.01)

Le même raisonnement peut être fait pour la fonction **_delay_us**, à la seule différence qu'ici les chiffres après la virgule ne sont pas significatifs. Donc le plus long délai réalisable est de 96us, alors que le plus court est de 1us.

Vous remarquerez que le plus long délai réalisable reste tout de même relativement court, nous verrons dans l'exemple de code ci-dessous, comment réaliser des délais de l'ordre de la seconde.

- ***La configuration du Compilateur***

Pour que les délais puissent être implémentés correctement, deux réglages sont à faire au sein de l'environnement de développement.

1) Tout d'abord il est essentiel que le compilateur connaisse la fréquence de travail du microcontrôleur, pour qu'il puisse calculer les constantes utilisées à l'intérieur des fonctions _delay.

Ceci peut sembler bizarre, mais l'environnement de développement (le compilateur y compris) ne connaît pas à quelle fréquence travail le microcontrôleur. Nous l'avons vu, celle-ci dépend de la configuration des fuses bit (qui ne sont pas accessible par le compilateur). De plus si le quartz externe est activé, la fréquence dépend directement du type de quartz installé.

Pour ce faire, allez dans les menus sous *Project->Configuration Options*

Puis entrer, en Hertz, la fréquence de travail de votre microcontrôleur. Dans l'exemple utilisé pour la Figure 7, nous avions une horloge de 8Mhz.

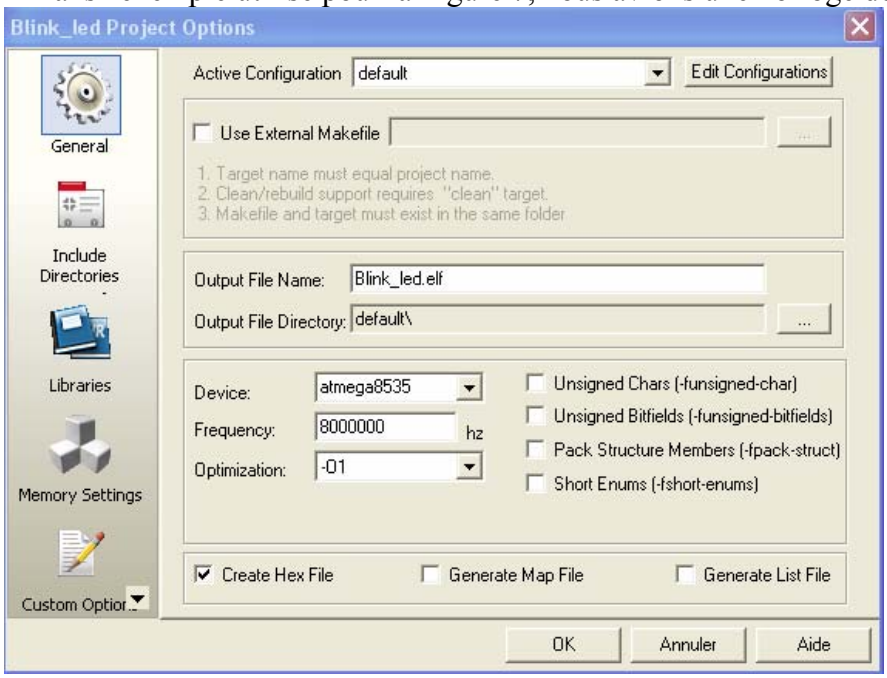


Figure 7 Option du projet

2) Le second point à configurer est le facteur d'optimisation, il se trouve aussi dans les options de projet (Figure 7). Par défaut il est mis à O0, et ceci

provoque un disfonctionnement des fonctions "_delay". Il faut activer l'une des options d'optimisation (Ici O1).

En effet avec l'option O0 aucune optimisation n'est effectuée par le compilateur, donc tous les calculs sont laissés au microcontrôleur, y compris les calculs des constantes! Or ce calcul utilise des nombres flottants, ce qui prend relativement beaucoup de temps à faire sur ce microcontrôleur. Une fois ces constantes calculées la fonction "_delay" est exécutée. Vous l'aurez compris, le fait de devoir recalculer à chaque fois les constantes provoque un délai supplémentaire[*], qui est difficilement prévisible.

Le fait d'activer une option d'optimisation oblige le compilateur à calculer toutes les constantes utilisées au sein du programme, ceci évite au microcontrôleur de le faire et résout donc notre problème. Ceci explique pourquoi il est essentiel d'activer une option d'optimisation et d'appeler les fonctions _delay_ UNIQUEMENT avec des **constantes**! Si l'on souhait créer un délai variable, on utilisera la méthode présentée dans l'exemple de code ci-dessous.

Les codes, que devra exécuter le microcontrôleur, sont relativement simples donc le choix du type d'optimisation n'est pas critique. Mais ceci peut le devenir lorsque la taille du code commence à atteindre les limites de la mémoire disponible, ou alors, lorsque les performances en vitesse deviennent un facteur critique. Nous conseillons donc au lecteur se trouvant dans ce cas, de consulter la documentation du compilateur GCC pour savoir exactement l'effet de chaque type d'optimisation.

3.2.4.c Exemple de code

Voici ci-dessous, un code qui fait clignoter tous les pins du PortB. Parcourons-le sommairement. Sa structure est identique à la structure de base présentée à la section 3.2.1.

[*] Ce délai supplémentaire est de l'ordre de quelques centaines de microseconde, il se remarque donc clairement lors de l'utilisation de la fonction _delay_us ()

Tout d'abord, les deux "*includes*"; le premier est celui de base et le second nous permettra d'utiliser les fonctions de délai.

Puis à l'intérieur de la fonction principale nommée "*main*", une variable temporaire est déclarée et initialisée à 0. La direction du Port est définie (ici en sortie). Nous rentrons ensuite dans une boucle infinie. Ceci peut sembler étrange de prime abord, mais c'est en fait exactement ce que nous souhaitons que le code fasse. Il doit mettre à 0 puis à 1 tous les pins du PortB, et répéter ceci un nombre de fois infini (c'est-à-dire dès le moment où on l'allume jusqu'au moment où on l'éteint)*

L'intérieur de cette boucle est une implémentation du Code 4, tout d'abord le PortB est mis à 0. Puis, afin que notre œil puisse voir le clignotement, il a été choisi d'attendre 100ms. Le délai maximum réalisable avec la fonction _delay_ms, en utilisant un quartz à 8Mhz, est de 32ms (cf.: section 3.2.4.b), c'est pourquoi pour réaliser le délai de 100ms il a été choisi d'attendre 4 fois 25ms. Puis de mettre les pins du PortB à 1, et d'attendre, avant de recommencer la boucle, à nouveau 100ms.

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

int main(void)
{
	//Variable Temporaire
	int i = 0;
	//Début du Programme!
	DDRB=0xFF; //Port B en sortie

	while(1)
	{
		PORTB=0x00; //Tout les pins du PortB à 0

		//Début du délai 4*25ms = 100ms
		for(i=0; i<=3;i++)
		{
			_delay_ms(25);
		}
		//Fin du délai de 100ms

		PORTB=0xFF; //Tout les pins du PortB à 0

		//délai 4*25ms = 100ms
		for(i=0; i<=3;i++)
		{
			_delay_ms(25);
		}
		//Fin délai 100ms
	}

	return 0;
}
```

Code 5 Clignotement du Port B

3.2.5 math.h

* La boucle infinie est très largement utilisée, car il est très rare que l'on demande à un microcontrôleur de faire quelque chose au moment où il est allumé et une fois cette opération terminée de s'arrêter!

Cette librairie est standard dans le langage C. Elle est utilisée lorsque des opérations mathématiques relativement complexes doivent être utilisées. Des fonctions trigonométriques, exponentielles ainsi que des opérateurs de puissance sont contenus dans cette librairie.

Pour l'utiliser, il suffit d'ajouter l'*include* <math.h>

Des explications détaillées sont disponibles dans les références indiquées à la section 3.2.1. Nous ne détaillerons pas d'avantage cette librairie, car son utilisation est souvent très spécifique.

Une configuration importante doit cependant être faite au sein du compilateur pour qu'il puisse compiler le code correctement. Il faut en effet dire au compilateur d'ajouter, durant une phase de compilation, la librairie libm.a qui contient l'implémentation des fonctions de math.h

Dans la fenêtre, Project Option (*Project->Configuration Options*), cliquez sur l'icône *Librairies*, une fenêtre semblable à la Figure 8 apparaît.

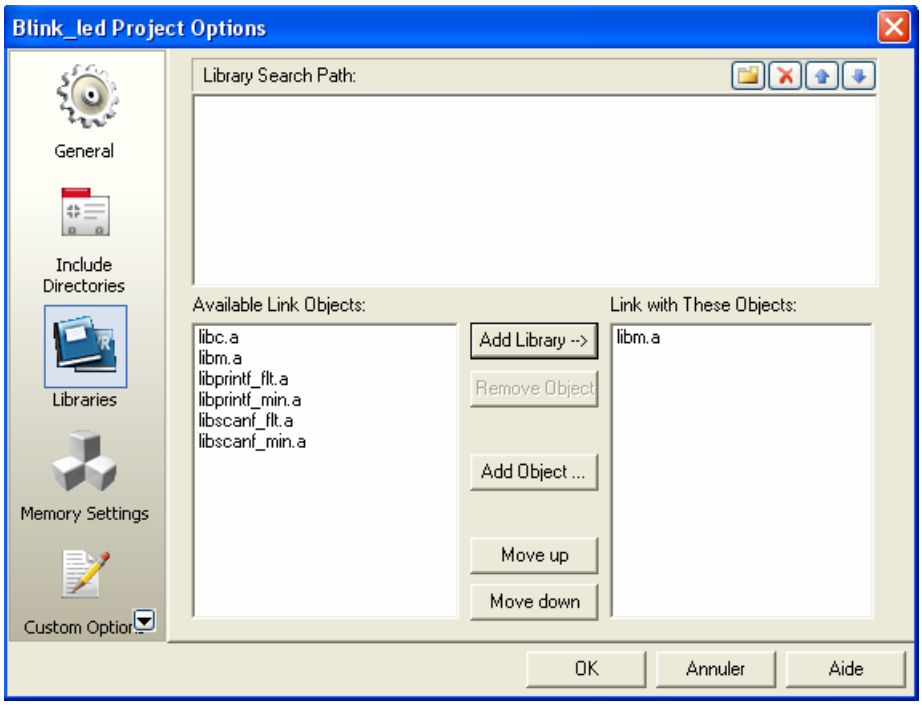


Figure 8 Configuration du compilateur pour utiliser la librairie math.h

Il suffit donc de cliquer sur libm.a et enfin de cliquer sur Add Library.

Durant la compilation, le message suivant apparaît dans la fenêtre build:

```
avr-gcc.exe -mmcu=atmega8535  Blink_led.o    -lm  -o Blink_led.elf
```

L'option –lm indique que la librairie libm a été ajoutée au projet lors de la compilation, ce qui est essentiel lorsque des fonctions contenues dans l'include math.h sont utilisées.

3.3 Les Interruptions

3.3.1 Principe de fonctionnement

Lorsque aucune interruption n'est utilisée, le microcontrôleur va exécuter le code assembleur final de manière séquentielle (Cf.: section 1.1). En fonction du résultat d'un test (Condition), nous pouvons exécuter un morceau de code spécifique. C'est-à-dire que l'on pourra par exemple exécuter le code A **SI** le PORTA est à 1.

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

int main(void)
{
	//Variable Temporaire
	//Début du Programme!
	DDRA=0x00; //Port A en entrée

	while(1)
	{
		//…
		//Code
		//…

		if(PORTA==0xFF)
		{
			//…
			//Code A
			//…
		}

		//…
		//Code
		//…
	}

	return 0;
}
```

Code 6 Exécution séquentielle

Ceci implique que l'on test régulièrement l'état de la condition et en fonction du résultat nous décidons quel code sera exécuté. Le temps de réponse est limité en utilisant cette méthode, car il nous faut toujours attendre que le microcontrôleur teste la condition.

Pour résoudre ce problème il faudrait pouvoir exécuter un code **LORSQUE** la condition devient satisfaisante, **c'est exactement à ceci que sert une interruption**.

En C, l'interruption s'écrit comme une fonction classique, cette fonction sera "appelée"* automatiquement par le microcontrôleur.
Pour qu'une interruption puisse avoir lieu, il faut auparavant l'autoriser (cf. 3.2.4.c). Ceci se fait une seule fois en début de programme, généralement dans la section destinée à l'initialisation.

Une fois cette configuration effectuée le programme s'exécute normalement jusqu'au moment où l'événement (pour lequel l'interruption a été autorisée) survient. A ce moment le

* Ceci est un abus de langages, l'appel de l'interruption au sein du microcontrôleur ne se fait pas comme celui d'une fonction, si ce sujet vous intéresse n'hésitez pas contacter un membre du comité pour de plus amples explications.

microcontrôleur interrompt son exécution, il exécute immédiatement le code de l'interruption puis, une fois celle-ci terminée, il reprend l'exécution du code principal.

L'Atmega8535 possède diverses interruptions, qui sont documentées dans le datasheet du composant. Dans ce document nous ne décrirons que l'interruption sur le signal externe. Les autres interruptions (timer, uart, ADC,…) seront décrites dans un autre document.

3.3.2 Exemple d'application

L'Atmega8535 possède 3 interruptions externes nommées INT0 -> INT2. C'est-à-dire 3 pins sur lesquels nous pouvons déclencher une interruption dès qu'ils changent d'état. Donc par exemple en branchant un capteur sur cette ligne d'interruption on pourrait, au moment ou le robot rencontre un obstacle, exécuter le code qui permettra de l'éviter. Afin de simplifier cet exemple, au moment où l'interruption se déclenche nous allumerons/éteindrons une LED.

Depuis la figure 1 du datasheet nous trouvons que :

- INT0 se trouve sur le pin 2 du port D
- INT1 est sur le pin 3 du port D
- INT2 est sur le pin 2 du port B.

La section "*External Interrupt"* dans le datasheet décrit les bytes de configuration.

La configuration de cette interruption se fait en 3 étapes:

3.3.2.a Configuration du flanc de déclenchement:

Nous voulons que l'interruption soit déclenchée chaque fois que le pin 2 du port D passe de la valeur 1 à la valeur 0, c'est-à-dire chaque fois qu'un flanc[*] descendant est détecté.

Le datasheet (Tableau 36 page 69) nous informe que pour ceci le bit ICS01 du registre MCUCR doit être mis à 1[†]

3.3.2.b Autorisation de l'interruption INT0

Cette étape autorisera le microcontrôleur à appeler la fonction d'interruption chaque fois que l'interruption est détectée[‡]. A nouveau le datasheet (Page 70) nous permet de savoir qu'il faut mettre le bit INT0 à 1 dans le registre GICR

[*] Le mot flanc en programmation/électronique est le moment où le signal change de valeur. Par exemple on appellera flanc montant le moment ou le signal passe de la valeur 0 à 1, et flanc descendant le moment ou le signal passe de 1 à 0

[†] Les noms des registres et des bits peuvent à prime abord sembler hostile et incompréhensible. Ce sont au final simplement des noms. Un registre (ici MCUCR) contient 8 bits, chacun ayant un nom. Un de ceux-ci se nomme ICS01. Et pour configurer le flanc descendant il doit être mis à 1

[‡] Rappelons ici que l'interruption sera détectée lors le signal présent sur le pin 2 du portD passe de 1 à 0. Physiquement cela signifie que chaque fois que la tension sur le pin PD2 passera de 5V à 0V l'interruption sera déclenchée

3.3.2.c Autorisation générale des interruptions

Les microcontrôleurs possèdent un bit particulier qui bloque toutes les interruptions. Il faut donc enclencher le bit Global Interrupt afin que celles-ci puissent être effectuées. Ceci se fait grâce à la commande "sei();".Il est parfois utile d'empêcher l'exécution des interruptions, pour se faire il faut mettre à 0 le bit Global Interrupt, et ceci se fait en appelant la fonction cli();

Pour rendre le code fonctionnel, il nous reste à inclure le fichier avr/interrupt.h dans le code, et à écrire la fonction d'interruption. Nous l'avons vu cette dernière est nommé ISR et reçoit en paramètre le "vecteur d'interruption".
La liste complète des vecteurs d'interruption se trouve dans le "User Guide" de WinAvr (cf.: lien section 3.2.1), ou alors directement sur :

http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

L'exemple de code ci-dessous illustre ce que nous venons de présenter.

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

ISR(INT0_vect)
{
	//Si la led est allumée on l'éteint et vice-versa
	//Pour de plus ample information sur les lignes qui suivent
	//cf. guide de programmation en C section 3.5.1 )

	//Test de l'état de la led
	if( (_PORTC2 == 1)
		//led allumée, il faut l'éteindre, le portC2 est donc
		//mis à 0
		_PORTC2 =0);
	else
		//led éteinte, il faut l'allumée, le portC2 est donc
		// mis à 1
		_PORTC2=1;

}

int main(void)
{
	//Variable Temporaire

	//Début du Programme!
	DDRC=0xFF; //Port C en sortie  PC2:Led
	DDRD=0x00; //Port D en entrée  PD2:Capteur IR
	DDRB=0xFF;//Port B en sortie
	//Config Interrupt
	MCUCR|=(1<<ISC01);  //falling Edge of INT0
	GICR|=(1<<INT0); //Enable INT0 Interrupt

	sei(); //Enable Global Interrupt
```

Code 7 Utilisation de l'Interruption INT0

Attention, ici nous testons volontairement le PORTC2 et non le PINC2 car, la led à été configurée en sorite, mais nous ne connaissons pas son état.

3.4 Opération bit à bit, description générale

La section 3.5 explique cette partie de façon légèrement différente. Cette partie étant un peu technique nous avons jugé utile de donnée 2 explications présentant le masquage avec un point de vue différent.
Nous ne pouvons que vous conseiller de lire les 2 sections. Si les explications vous semblent répétitives, c'est que vous avez compris le sujet et qu'il ne vous reste plus qu'à le mettre en pratique en faisant les challenges qui vous sont proposés sur le site web http://robopoly.epfl.ch

3.4.1 Les 3 types de bases

Vous le saurez déjà certainement, en programmation 3 bases numériques sont utilisées, la base 10 qui est celle quotidiennement utilisée (chiffre de 0 à 9), la base hexadécimale (chiffre de 0 à F) et enfin la base binaire (chiffre de 0 à 1). La calculatrice de Windows permet de rapidement passer d'une base à l'autre.

Ouvrez la calculatrice, puis sous le menu affichage sélectionnée la vue scientifique. En cliquant sur le bouton, dec, Hex, bin il est possible de changer de base.

Afin de ne pas confondre les 3 bases une notation particulière sera adoptée. Le chiffre décimaux se noterons naturellement 0, 1, .., 10, 100, etc. Les nombres en hexadécimale seront notés 0x0, 0x1,…0xA, 0xB, 0x12, 0xA4, 0xFF, etc. Les nombres binaires seront quant à eux noté 0b0, 0b01, 0b11, 0b001001, 0b111011, etc. Ces différents bases sont utilisées uniquement pour simplifiée la vie du développeur, pour le microcontrôleur, la manière dont le nombre est notée n'aura aucune influence sur sont fonctionnement.

Pour passer par exemple le 3ème pin du PORTD à 1 nous pourrons écrire indifféremment

- PORTD=0b00000100*;
- PORTD=4;
- PORTD=0x04;

De même il est possible de changer 2 pins d'un port simultanément. Pour faire passer le 8ème pin ainsi que le 3ème pin du PORTD à 1, nous pourrons écrire

- PORTD=0b10000100;
- PORTD=132;
- PORTD=0x84;

Cet exemple montre que l'utilisation des nombres en décimal n'est parfois pas du tout intuitif, la version binaire étant nettement plus parlante. Au contraire dans d'autres situations les nombres binaires seraient absolument incompréhensibles, par exemple les opérations mathématiques.

3.4.1.a Petites astuces

Pour faciliter la conversion du format binaire vers hexadécimal et vice-versa. 1 chiffre hexadécimal est toujours codé en utilisant 4 chiffres binaires

*Convention de notation. Afin de clairement identifier la base dans laquelle le nombre est écrit il a été choisi de placer une syntaxe avant le nombre. Donc si vous voulez écrire un nombre en binaire la syntaxe est 0b####, si vous voulez écrire le nombre en hexa la syntaxe est 0x##. Ainsi il n'y a aucune confusion possible! Lorsque nous écrirons Variable= 1010 (Variable = mille-dix), Variable=0b1010 (Variable = un zéro un zéro en binaire ce qui équivaux à 10 en décimal) et de même Variable=0x0A (Variable = zéro A en héxadécimal ce qui vaut 10 en décimal)
Notez le lien direct qui existe entre l'écriture binaire et l'accès physique au port que vous avez sur votre PRisme. Le chiffre le plus à droite correspond au pin numéroté 0 sur votre PRisme, et le chiffre à droite du 0b correspond au pin numéro 7.

décimal	binaire	hexa	décimal	binaire	hexa
1	0b00000001	0x01	16	0b00010001	0x11
2	0b00000010	0x02	17	0b00010010	0x12
3	0b00000011	0x03	18	0b00010011	0x13
4	0b00000100	0x04	19	0b00010100	0x14
5	0b00000101	0x05	20	0b00010101	0x15
6	0b00000110	0x06	21	0b00010110	0x16
7	0b00000111	0x07	22	0b00010111	0x17
8	0b00001000	0x08	23	0b00011000	0x18
9	0b00001001	0x09	24	0b00011001	0x19
10	0b00001010	0x0A	25	0b00011010	0x1A
11	0b00001011	0x0B	26	0b00011011	0x1B
12	0b00001100	0x0C	27	0b00011100	0x1C
13	0b00001101	0x0D	28	0b00011101	0x1D
14	0b00001110	0x0E	29	0b00011110	0x1E
15	0b00001111	0x0F	30	0b00011111	0x1F

Tableau 1 Exemple base numérique

Exemples supplémentaires

décimal	binaire	hexa	décimal	binaire	hexa
17	0b00010001	0x11	81	0b01010001	0x51
33	0b00100001	0x21	97	0b01100010	0x62
49	0b00110001	0x31	115	0b01110011	0x73
65	0b01000001	0x41	132	0b10000100	0x84

Tableau 2 Exemple 2 base numérique

Le passage binaire décimal est relativement compliqué et une calculatrice est souvent indispensable, au contraire le passage de binaire vers hexadécimal est relativement simple. En effet le premier chiffre du nombre en hexadécimal correspond au 4 premiers chiffres du nombre binaire, le second chiffre du nombre hexadécimal correspond quant à lui aux 4 derniers chiffres du nombre binaire, regardez attentivement le tableau ci-dessus. Ainsi pour faire la conversion de binaire vers hexadécimal il suffit de connaître la représentation binaire des chiffres hexadécimaux (c'est-à-dire savoir compter jusqu'à 15)

3.4.2 Le masquage

Souvent un port est utilisé simultanément par divers capteurs ou actionneurs. Imaginer par exemple que les moteurs de votre PRisme soient connectés sur les pins du PORTD, et que sur le même PORTD vous ayez branché une LED. Enfin, supposons que la LED se trouve sur le pin 0, et que les moteurs se trouvent sur les pins 3 à 6. Si nous voulons allumer la LED il suffira donc d'écrire PORTD=1, ou PORTD=0x01, ou encore PORTD=0b00000001. De même la commande PORTD=0 éteindra la LED. En utilisant ces commandes, nous avons modifié le pin0, mais simultanément nous avons affecté une valeur à tous les pins. Le résultat de la commande PORTD=1 fait bien allumé la LED, mais force le robot à s'arrêter. Comment allumer la LED sans affecté la trajectoire du robot, qui est elle commandée par une autre partie du code? C'est pour ce but que le masquage est utilisé!
Le masquage est aussi utilisé lorsque plusieurs capteurs sont branchés sur le même port. Comment savoir si le capteur numéro 3 est à 1 sans être influencé par les autres?

Exemple:

Supposons que le PORTA contiennent 8 capteurs. Nous aimerions savoir si le capteur 3 est à 1. La condition PORTA==0b00000100 sera vrai uniquement si le capteur 3 est à 1 **ET** que tout les autres capteurs sont à 0. Imaginez que les capteurs 3 et 1 soient à 1, la condition PORTA==0b00000100 sera fausse, en effet PORTA vaut 0b00000101.

Le masquage utilise les fonctions logique bit à bit & (ET logique) et | (OU logique). Ci-dessous vous trouvez la table de vérité de ces 2 fonctions

ET logique:	OU logique:
&	|
0&0 = 0 0&1 = 0 1&0 = 0 1&1 = 1	0|0 = 0 0|1 = 1 1|0 = 1 1|1 = 1
Le résultat d'un & est 1 uniquement si le premier bit ET le second valent 1	Le résultat d'un | est 1 uniquement si le premier bit OU le second valent 1 (si les 2 valent 1 le résultat sera 1)

Tableau 3 Table de vérité

La fonction & sera utilisée pour faire passer le bit que nous voulons modifier à 0, alors que la fonction | sera utilisée quand nous voudront faire passer 1 pin à 1. Le tableau ci-dessous contient un exemple d'application.

Pin 0 passe à 0	Pin 0 et 3 passe à 0	Pin 0 passe à 1	Pin 0 et 3 passe à 1
0b01010101 & 0b1111111**0** 0b0101010**0**	0b01010101 & 0b1111**0**1**0** 0b0101**000**0	0b01010010 | 0b0000000**1** 0b0101001**1**	0b01010010 | 0b0000**10**1 0b0101**10**1

Tableau 4 Exemple Masquage

Afin de se convaincre du fonctionnement du masquage, essayez d'appliquer les lois logiques vues au Tableau 3 par vous-même.

Le nombre qui est utilisé à la deuxième ligne du Tableau 4 est appelé masque.

3.4.3 En résumé :

- Pour faire passer des pins à 0, effectuez un & (ET logique) entre le PORT et un masque contenant des 0 sur les pins que l'on désire changer, le reste du masque doit contenir des 1
    - Valeur actuelle du PORTD (inconnue lors de l'écriture du code) est 0b01010000
    - PORTD = PORTD & 0b11111110; //Code
        - La nouvelle valeur du PORTD est 0b01010000

        - nous venons de passer le pin0 du PORTD à 0 dans notre exemple ce pin était connecté à une LED, la LED s'est donc éteinte sans que le robot ait changé de trajectoire
- Pour faire passer des pins à 1, effectuez un | (OU logique) entre le PORT et un masque contenant des 1 sur les pins que l'on désire changer, le reste du masque doit contenir des 0
        - Valeur actuelle du PORTD (inconnue lors de l'écriture du code) est 0b01010001
    - PORTD = PORTD & 0b00000001; //Code
        - La nouvelle valeur du PORTD est 0b01010001
        - nous venons de passer le pin0 du PORTD à 1 dans notre exemple ce pin était connecté à une LED, la LED s'est donc allumée sans que le robot ait changé de trajectoire

Cette partie peut sembler un peut compliquée lors de la première lecture, vous verrez après quelques utilisations, elle deviendra très simple car logique et répétitive. Des exemples de codes sont disponibles dans les divers tutoriaux présents sur le site http://robopoly.epfl.ch

3.5 Opération sur les bits*

Le langage C offre la possibilité de travailler directement sur la valeur binaire d'une variable. Imaginons un entier 8bits non signé. Les valeurs possibles sont contenues entre 0 et 255 en représentation décimale. En représentation binaire on a une fourchette entre 0b00000000 et 0b11111111 et en hexadécimal entre 0x00 et 0xFF.

Il arrive parfois que le travail en binaire soit plus agréable. Prenons, par exemple, un port, le PORTC. On peut le percevoir comme un entier 8bits. Si l'on veut mettre à 1 la patte numéro 3 du PORTD, nous pouvons le faire, soit en décimal en donnant la valeur 4, soit en lui donnant la valeur 0b00000100. Il n'y a aucune différence entre les deux manières d'écrire.

Imaginons maintenant que le PORTD contient déjà une valeur arbitraire 0b11001010 qui est définie en cours d'utilisation mais que nous ne connaissons à priori pas lors de l'exécution du code. Si nous désirons mettre à 1 le 3ème bit du port, nous ne pouvons plus utiliser la même manière que précédemment. En effet le fait d'écrire PORTD = 4 changerait la valeur de tous les bits et nous aurions le résultat 0b00000100 au lieu de 0b11001110 ! Nous allons donc introduire le concept de masquage.

3.5.1 Masquage

Le masquage consiste à effectuer une opération logique (ET, OU, XOR, etc…) entre deux bytes afin de pouvoir mettre à 1 ou à 0 un ou une série de bit bien précis SANS modifier la valeurs des autres bits de la variable choisie.

Voici les tables de vérité des opérations les plus courantes :

ET ≡ &

&	1	0
1	1	0
0	0	0

OU ≡ |

|	1	0
1	1	1
0	1	0

XOR ≡ ^

^	1	0
1	0	1
0	1	0

On peut remarquer que l'opération ET permet de forcer à 0 un bit (en faisant un ET entre n'importe quelle valeur et 0 le résultat est toujours 0). L'opération OU permet de forcer

* Section rédigée par Christophe Winter

à 1 un bit (un OU entre n'importe quelle valeur et 1 donne toujours 1) et l'opération XOR d'inverser la valeur d'un bit (un XOR entre n'importe quelle valeur et 1 donne toujours l'inverse de la valeur de départ).

Passons maintenant à la mise en pratique du masque. Reprenons l'exemple sur le PORTD. Comme nous voulons mettre à 1 le bit 3, nous devons utiliser l'opération OU. Nous allons tout d'abord définir une sorte de byte de commande qui contiendra comme information les bits sur lesquels nous voulons travailler. Dans notre cas il s'agira de 0b00000100 à savoir un 1 sur le 3ème bit puisque ce n'est que lui que nous voulons modifier ici. En calculant PORTD | 0b00000100 nous obtenons le résultat recherché 0b11001110 (à savoir un OU bit à bit entre 0b11001010 et 0b00000100).

Si maintenant nous souhaitons faire revenir à 0 le bit 3 du PORTD, il faudra utiliser l'opération ET. Comme précédemment nous créons le byte de "commande" à savoir 0b11111011. Cette fois on place un 0 sur les bits que nous voulons changer (ceci est dû à la table de vérité de l'opération ET). En calculant PORTD & 0b11111011 nous obtiendrons la valeur de départ 0b1101010 et le tout sans jamais avoir modifié la valeur des autres bits que le 3ème ! Ce principe est utile si l'on veut par exemple allumer une LED qui se trouve sur le même port que les moteurs, le tout sans que le robot s'arrête.

Si maintenant le but est de faire clignoter une LED ou d'inverser une valeur sans devoir au préalable aller vérifier l'état actuel pour choisir l'état futur, on peut utiliser l'opération XOR. Dans ce cas, le byte de résultat verra son bit inversé à l'endroit où se trouve un 1 dans le byte de commande. (0b11001010 ^ 0b00000100 = 0b11001110 et 0b11001110 ^ 0b00000100 = 0b11001010).

4 Historique

Date	Version	Auteur	Description
24.10.2007	1.0.0	F. Lo Conte	Version de base