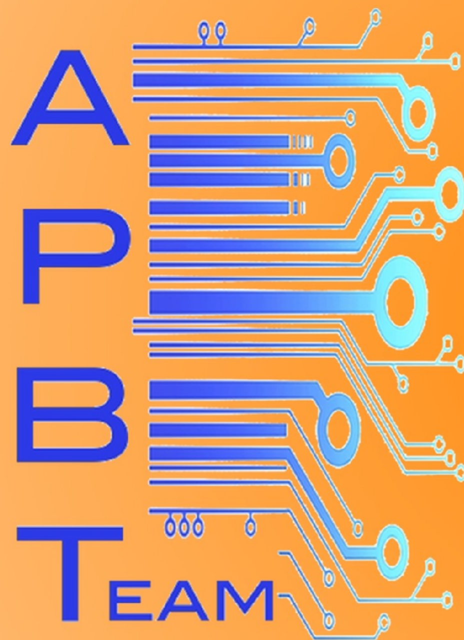
TUTORIEL

Introduction à la programmation de micro-contrôleur AVR 8-bits



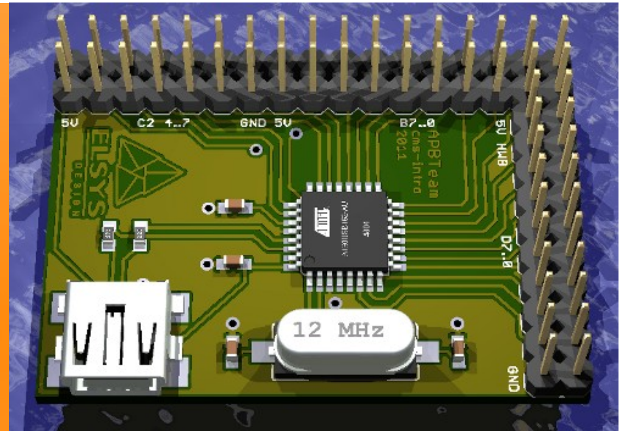
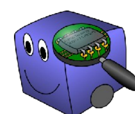


Table des matières

Introduction....3
Présentation d'Elsys-Design....3
Qu'est-ce qu'un micro-contrôleur?....4
Matériel, logiciel et documentations....4
Nos premiers pas avec l'AVR AT90USB162....5
Présentation de l'AVR....5
Mapping de la carte....5
Fonctionnement des I/O....6
Notre premier programme....7
Montage....7
Software (1ère version)....7
Les bibliothèques....8
Le démarrage....8
La compilation....10
dfu-programmer....11
Pour un programme plus propre et plus modulaire....12
Réutiliser son code....12
La compilation....13
Delay AVR 13
Une (petite) amélioration de my_delay()....14
Un programme plus complexe : un compteur électrique....14
L'afficheur 7 segments....14
« Multiplexage » des sorties....15
Afficher les nombres....15
Création de notre bibliothèque....16
Ecriture du main: 18



Introduction

Ce tutoriel réalisé par APBTeam en partenariat avec Elsys Design, a pour but d'initier à la programmation des micro-contrôleur.

En effet, notre équipe propose depuis maintenant deux ans un stand « Introduction à la soudure CMS ». Ceux qui participent à cette formation repartent avec une mini carte de développement (soudée par leurs soins) qui contient un micro-contrôleur. Ce tutoriel a donc pour but de les aider à continuer de travailler dessus. Nous utiliserons donc la mini carte de développement conçue par APBTeam & Elsys-Design et expliquerons comment programmer dessus aussi bien avec un environnement Linux qu'avec un environnement Windows.

Ce tutoriel part du principe que vous avez quelques notions de programmations avec n'importe quel langage, mais c'est à peu prés tout. Une fois ce tutoriel terminé, vous n'aurez « plus qu'à » vous plonger dans le livre de votre choix pour apprendre à coder correctement :).

Si vous êtes déjà un codeur en C de bon niveau, certains paragraphes vous sembleront moins intéressant.

Présentation d'Elsys-Design

ELSYS Design est le spécialiste en conception de systèmes électroniques (métiers du hardware, du logiciel embarqué et des systèmes électroniques). Entreprise française, elle a donné naissance à ADVANS Group, l'un des leaders européens de la spécialité (850 ingénieurs, 63M€ de chiffre d'affaires).

ELSYS Design, de par son positionnement métier unique, permet à ses ingénieurs d'exercer leurs compétences dans de nombreuses industries – aérospatiale, défense, énergie, médical, multimédia, télécom, transports… Les modes d'interventions sont variés : ELSYS Design est la seule société de cette envergure à présenter une activité répartie de façon équilibrée entre l'assistance technique, les forfaits et les centres de développement dédiés.

Fondée par des ingénieurs, managée par des ingénieurs, ELSYS Design rassemble les meilleurs talents, dans un environnement propice à l'enrichissement de leurs compétences. L'entreprise et son groupe offrent de nombreuses opportunités d'évolution, en France et à l'international – management de projet, expertise technique, fonctions commerciales...

Positionnée à la pointe des technologies, ELSYS Design participe également à des projets de recherche et favorise l'essaimage de sociétés innovantes.

http://www.elsys-design.com

ELSYS Design : Imaginer et Concevoir les Systèmes Électroniques du Futur

✉: recrutement@elsys-design.com

Qu'est-ce qu'un micro-contrôleur?

Nous reprenons ici l'introduction de l'article Wikipedia sur ce sujet :

Un micro-contrôleur est un circuit intégré qui rassemble les éléments essentiels d'un ordinateur : processeur, mémoires (mémoire morte pour le programme, mémoire vive pour les données), unités périphériques et interfaces d'entrées-sorties.
Les micro-contrôleurs se caractérisent par un plus haut degré d'intégration, une plus faible consommation électrique, une vitesse de fonctionnement plus faible (quelques mégaHertz à quelques centaines de mégaHertz) et un coût réduit par rapport aux microprocesseurs polyvalents utilisés dans les ordinateurs personnels.

Contenu soumis à la licence CC-BY-SA 3.0. Source : Article Microcontrôleur de Wikipédia en français

La page Wikipedia est d'ailleurs une bonne chose à lire en introduction à ce sujet.

Matériel, logiciel et documentations

En plus de cette carte, on utilisera quelques éléments :

- une plaquette labdec (plaque à bidouille),
- quelques diodes,
- quelques résistances,
- 2 afficheurs 7 segments.

De plus, voici une petite liste de ce qu'il faut avoir installé sur votre ordinateur avant de pouvoir programmer le micro-contrôleur :

- votre éditeur de texte préféré (afin de pouvoir coder),
- avr-gcc (qui permet de compiler votre programme pour une cible AVR),
- avr-libc (qui contient les bibliothèques que vous allez utiliser),
- dfu-programmer (qui permet de communiquer avec notre AVR via le port USB),
- Un câble USB A mâle d'un coté et micro USB B mâle de l'autre (typiquement un connecteur téléphone portable-PC).

Enfin la documentation indispensables à ceux qui veulent se lancer dans l'aventure :

- la datasheet du composant,
- AVR libc user manual.

Nos premiers pas avec l'AVR AT90USB162

Une fois tout cela installé, préparé, téléchargé, etc... on peut enfin commencer à attaquer les choses sérieuses.

D'abord quelques petites remarques rapides.

La carte est alimenté via le port USB, et donc en 5V.

La carte ne possède rien, mises à part : la puce, quelques capacités de découplage et les contacts d'entrée-sortie. D'où la nécessité d'avoir une plaque à bidouille pour pouvoir faire un petit projet.

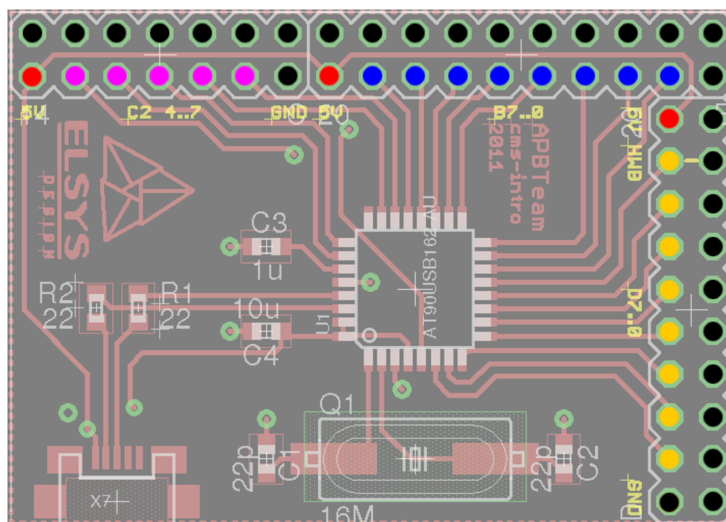
Ce micro-contrôleur possède un oscillateur interne, cependant nous avons tout de même ajouté un oscillateur externe afin d'augmenter sa rapidité et sa précision (ce qui est de toute façon obligatoire pour utiliser l'USB).

Présentation de l'AVR

Afin de pouvoir utiliser notre AVR, il nous faut évidemment des entrées sorties. Voici comment elles fonctionnent sur les AVR.

Mapping de la carte

Grâce aux contacts situés sur les côtés, l'utilisateur peut utiliser la carte plus facilement.



- +5V (Rouge)
- GND (Noir)
- C2, C4, C5, C6, C7 (Violet)
- B7, B6, B5, B4, B3, B2, B1, B0 (Bleu)
- D7, D6, D5, D4, D3, D2, D1, D0 (Orange)
- HWB (sur D7)

La patte RESET, n'est pas câblé sur les contacts. Il est donc un peu plus difficile d'y accéder.

Comme vous pouvez le constater, la carte possède plusieurs ports d'entrée-sorties (B, C & D). Certaines d'entre elles permettent l'utilisation de périphériques ou mode de fonctionnement spéciaux (ex : interruptions, PWM…), d'autres sont des I/O standards.
Se référer à la datasheet pour plus de précision (rapide résumé donné en page 5&6).

Fonctionnement des I/O

Pour contrôler chaque port d'entrée/sortie, on dispose de registres de huit bits (chaque port étant composé de 8 broches).

Voici une description de chaque registre :

PIN	DDR	PORT
Read only	Read/write	Read/write
Valeur lue présente sur l'entrée	Permet de choisir le sens d'utilisation du port 0: input (par défaut) 1: output	Si DDR: 1 permet de modifier la valeur d'une sortie Si DDR: 0 Met en place un pull-up sur cette broche

DDR	PORT	Résultat
0	0	Input mode No pull-up
0	1	Input mode Pull-up enabled
1	0	Output mode Output set to 0
1	1	Output mode Output set to 1

Autrement dit, un GPIO, se contrôle de la manière suivante :

Admettons que nous voulons mettre la broche '3' du port 'B' à 1 :

```
DDRB = 0xFF;  /* Toutes les broches du PORTB seront des sorties. */
PORTB = 0x04; /* En binaire cela donne 00000100, et donc la 3eme broche est à '1'. */
```

Cependant, spécifier ainsi le comportement d'une broche n'est pas très lisible. Surtout lorsque le code commence à devenir plus complexe. Il existe donc une autre façon de modifier une broche donnée (cette notation est très pratique lorsqu'on ne souhaite modifier qu'une seule broche sur un port donnée) :

```
PORTB = PORTB | _BV(7); /* Mise à '1' de la 7eme broche du port B (et uniquement celle ci). */
PORTB = PORTB & ~_BV(7); /* Mise à '0' de cette même broche (et uniquement celle ci). */
```

Ce qui en C se condense ainsi:

```
PORTB |= _BV(7);
PORTB &= ~_BV(7);
```

_BV(n) est une macro fournie par AVR libc qui calcule la valeur du bit n (soit 1 << n).

Dans la suite de ce tutoriel, nous utiliserons les deux notations indifféremment.

Notre premier programme

Maintenant que l'on en sait un peu plus sur notre micro-contrôleur, il est temps de réaliser notre premier projet.

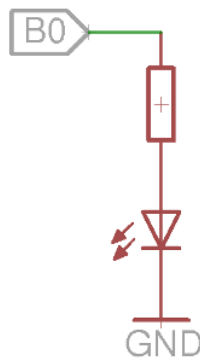
Quand on commence à programmer sur un ordinateur, on a toujours le droit au fameux « Hello world ». Sur un micro-contrôleur, l'équivalent consiste à faire clignoter une LED. Et pour cela il suffit de faire varier l'état d'une broche donnée de '1' à '0'.

Montage

Comme vous l'avez probablement deviné, il faut d'abord brancher une LED (ne pas oublier de la protéger avec une résistance).

On va brancher la LED sur la sortie B, broche 0 (il n'y a pas de raison spéciale, il fallait juste en choisir une).

Bon, ce montage ne devrait pas poser trop de problème.



Software (1ère version)

Maintenant, on peut commencer à coder :

On va (de manière très originale) nommer notre premier programme : blink_led.c.

```
int main (void)
{
        /* Set PORTB for output. */
        DDRB = 0xFF;
        while (1) {
                /* Set PORTB high. */
                PORTB = 0xFF;
                my_delay ();
                /* Set PORTB low. */
                PORTB = 0x00;
                my_delay ();
        }
```

```
	return 0;
}
```

Comme vous pouvez le constater, ce programme est vraiment très simple.

Il utilise le port B comme une sortie, et toutes ses broches font la même chose (même si la LED n'est connectée qu'à une seule d'entre elles).

Il faut aussi créer une fonction « delay ». Sinon la LED clignoterai beaucoup trop vite !

Cette fonction est juste quelque chose qui ferait « perdre son temps » à la puce entre deux instructions.

Voici un exemple que l'on pourrait écrire :

```
void my_delay (void)
{
	volatile int n = 0;
	int i;
	int tps = 32000;
	for (i = 0; i < tps; i++)
	{
		n = n + 1;
	}
}
```

C'est vraiment très simple. On se contente de compter de 0 à 32000 ce qui devrait prendre un peu de temps.

Ceci dit attention : Le “volatile” devant le « int n » est là pour forcer le compilateur à ne pas chercher à optimiser le code. Sans lui, lors de la compilation, cette fonction serait optimisée et la puce ne ferait pas tous les calculs. Cette fonction serait donc inutile.

On est maintenant presque prêt à tester notre premier programme, mais il reste encore quelques petites chose.

Les bibliothèques

Vous avez peut-être remarqué qu'on a inclus aucune bibliothèque dans notre programme. Le compilateur n'a donc aucun moyen de savoir ce que DDRB et PORTB veulent dire et il ne pourra donc pas faire son travail correctement.

Pour l'instant, on utilise juste les entrées/sorties basiques. Il nous faut donc inclure la bibliothèque d'entrée sortie de l'AVR-Libc :

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

Le démarrage

Avant d'aller plus loin, il faut connaître un petit peu comment est organisée la mémoire de notre AVR et ce qui se passe au démarrage.
La mémoire programme (Program memory) est divisée en deux parties :

- Le bootloader qui met l'AVR dans un mode spécial permettant notamment de charger un programme.
- Le reste de la mémoire qui permet de charger les programmes utilisateur.

Après un reset, le microcontrôleur a pour ordre :
« Si la patte 13 (HWB) est à 0, alors je vais au Bootloader. Si non, j'exécute le programme utilisateur. »

Cependant, ce test n'est pas effectué lorsqu'on allume le microcontrôleur. Or le reset **N'EST PAS CABLE** sur la plaquette APBTeam, et il n'est donc pas aisé d'en faire un : il faut aller toucher la pin 24 avec un fil relié à la masse, avec des risques de court-circuit notamment.

Il est donc beaucoup plus aisé d'ajouter dans notre programme une première instruction qui teste la valeur de la patte HWB et va sur le bootloader (à l'adresse 0x30000) si sa valeur est à '0' :

```
/* Jumps to the bootloader if port D7 is low. */
void try_bootloader (void)
{
	DDRD = 0x00;
	PORTD |= _BV (7);
	if (!(PIND & _BV(7)))
		((void (*) (void)) 0x3000) ();
}
```

Cette fonction n'est pas forcément aisée à comprendre. Pour les non-experts en C, inutile de passer trop de temps dessus. Il suffit de savoir pourquoi elle est là et ce qu'elle permet de faire.

Maintenant, il suffit de mettre la patte HWB à 0 lors du démarrage de l'AVR pour pouvoir modifier le programme utilisateur.

Voici donc le programme final que nous allons devoir compiler :

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

void my_delay (void)
{
	volatile int n = 0;
	int i;
	int tps = 32000;
	for (i = 0; i <= tps; i++)
	{
		n = n + 1;
	}
}

/* Jumps to the bootloader if port D7 is low. */
void try_bootloader (void)
{
```

```
	DDRD = 0x00;
	PORTD |= _BV(7);
	if (!(PIND & _BV(7)))
		((void (*) (void)) 0x3000) ();
}

int main (void)
{
	try_bootloader ();
	/* Set PORTD for output. */
	DDRD = 0xFF;
	while (1) {
		/* Set PORTD high. */
		PORTD = 0xFF;
		my_delay ();
		/* Set PORTB low. */
		PORTD = 0x00;
		my_delay ();
	}
	return 0;
}
```

La compilation

Il nous faut maintenant compiler ce code. Pour cela on va devoir utiliser avr-gcc.

Gcc (Gnu compiler collection) permet de compiler du code pour cible AVR, cependant dans un soucis d'optimisation, le binaire installé par défaut dans la plupart des distributions Linux ne permet pas de le faire.

On peut donc soit télécharger les sources complètes de gcc, et les compiler soi-même, soit ajouter le paquet AVR-GCC.

Maintenant on compile en écrivant dans un terminal :

```
avr-gcc -mmcu=at90usb162 -Wall -o blink_led.elf blink_led.c
avr-objcopy -j .text -j .data -O ihex blink_led.elf blink_led.hex
```

Explication de cet obscur charabia:

1ere ligne :

- avr-gcc : c'est le programme appelé,
- -mmcu : c'est la cible souhaité (notre micro contrôleur quoi),
- -Wall : c'est une option de compilation qui demande d'activer tous les warnings,
- -o blink_led.elf : c'est une option de compilation qui spécifie la sortie,
- blink_led.c : applique tout ce qu'on a vu précédemment à ce fichier.

Autrement dit, on demande de compiler blink_led.c avec avr-gcc pour une cible AT90USB162 avec tous les warnings, et la sortie s'appellera blink_led.elf.

2eme ligne :

Le fichier .elf généré par le compilateur contient beaucoup d'informations non utilisées par dfu-

programmer. La deuxième ligne va traduire le fichier ELF dans un format plus simple qui ne contient que ce qui est nécessaire pour programmer l'AVR.

- avr-objcopy : c'est le programme appelé, qui permet de convertir un fichier entre plusieurs formats,
- -j .text -j .data : demande à copier uniquement le programme (.text) et ses données (.data),
- -O ihex : choisi le format de sortie IntelHex.

dfu-programmer

Et voilà, maintenant que votre programme a été compilé, on est prêt à le mettre dans la mémoire de votre AVR.

Pour cela :

- mettre la patte HWB à la masse (cf le mapping de la carte),
- mettre la patte RESET à la masse (après vous n'aurez plus besoins de le faire grâce à l'ajout de la fonction « trybootloader » dans vos programme, c'est aussi inutile si l'AVR n'a jamais été programmé),
- brancher le microcontrôlleur au PC (via le port USB),
- relâcher la patte RESET, puis ensuite la patte HWB,
- entrer les lignes suivantes dans votre terminal (en mode super-utilisateur) :

```
dfu-programmer at90usb162 erase
dfu-programmer at90usb162 flash blink_led.hex
dfu-programmer at90usb162 start
```

Afin d'accélérer cette procédure, il est conseiller d'écrire un petit script bash que l'on nommera load_hex par exemple :

```
#!/bin/bash
dfu-programmer at90usb162 erase
dfu-programmer at90usb162 flash $1
dfu-programmer at90usb162 start
```

N'oubliez pas de le rendre exécutable

```
chmod +x load_hex
```

Maintenant il vous suffira d'écrire la ligne suivante :

```
sudo ./load_hex VOTRE_FICHIER
```

Et voilà normalement votre LED devrait clignoter, et un immense sentiment de satisfaction devrait vous envahir. Ceci dit vous devriez vite vous apercevoir que votre entourage trouvera votre enthousiasme quelque peut exagéré pour une lampe qui clignote...

Pour un programme plus propre et plus modulaire

Réutiliser son code

La première chose à faire, est de regarder d'un peu plus prés comment réutiliser nos programmes. En effet, dans le programme précédent nous avons écris une fonction (try_bootloader) que nous allons devoir réutiliser partout.

On va donc créer une bibliothèque « à nous » ou l'on mettra les fonctions générales que nous développerons. Nous l'appellerons : myavrlib.

Pour l'instant, on va y mettre nos deux fonctions :

- try_bootloader()
- my_delay()

Pour cela, nous allons créer un nouveau fichier « myavrlib.c » qui contient ceci :

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

void my_delay (void)
{
	volatile int n = 0;
	int i;
	int tps = 32000;
	for (i = 0; i <= tps; i++)
	{
		n = n + 1;
	}
}

/* Jumps to the bootloader if port D7 is low. */
void try_bootloader (void)
{
	DDRD = 0x00;
	PORTD |= _BV(7);
	if (!(PIND & _BV(7)))
		((void (*) (void)) 0x3000) ();
}
```

Maintenant nous allons créer le fichier.h (le header) dans un nouveau fichier : « myavrlib.h »

```
#ifndef HEADER_MYAVRLIB
#define HEADER_MYAVRLIB
void my_delay (void);
void try_bootloader (void);
#endif
```

Et voilà, on peut maintenant réécrire notre programme en utilisant notre bibliothèque personnelle :

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

```
int main (void)
{
	try_bootloader();

	/* Set PORTD for output. */
	DDRD = 0xFF;
	while (1) {
		/* Set PORTD high. */
		PORTD = 0xFF;
		my_delay ();
		/* Set PORTB low. */
		PORTD = 0x00;
		my_delay ();
	}
	return 0;
}
```

La compilation

Il faut maintenant compiler et linker les fichiers ensembles :

```
avr-gcc -mmcu=at90usb162 -Wall -o blink_led.elf blink_led.c myavrlib.c
avr-objcopy -j .text -j .data -O ihex blink_led.elf blink_led.hex
```

Et on peut charger le hex obtenu dans la carte (via dfu-programmer) ce qui devrait donner le même résultat que précédemment.

Delay AVR

La fonction my_delay() bien que pratique, est assez cavalière : en effet il existe déjà des fonctions de delay dans la bibliothèque AVR contenues dans delay.h avec lesquelles on peut directement spécifier le temps d'attente :

_delay_us()
_delay_ms()

dans lesquelles on spécifie directement un temps (en micro ou en millisecondes)

On ne peut cependant pas les utiliser telles quelles, car les délais maximums possibles pour ces deux fonctions sont :

_delay_ms() : 262.14 ms / F_CPU en MHz. (soit dans notre cas 16 ms)
_delay_us() : 768 us / F_CPU en MHz. (soit dans notre cas 48 us)

Il nous faut donc boucler autour de ces fonctions pour pouvoir les utiliser dans notre application.

Pour utiliser ces deux fonctions il suffit d'inclure la bibliothèque delay.h, ainsi que de définir la

vitesse du CPU :

```
#define F_CPU 16000000UL
#include <util/delay.h>
```

Une (petite) amélioration de my_delay()

Nous allons donc améliorer notre fonction my_delay() afin de la rendre plus efficace. On pourra maintenant spécifier le temps d'attente (en ms), et cela grâce à une magnifique petite boucle for :

Attention, ces fonctions n'acceptent qu'une constante en paramètre.

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>

void my_delay (int tps)
{
      for (; tps; tps--)
            _delay_ms(1);
}
```

N'oubliez pas de modifier aussi le fichier .h :

```
#ifndef HEADER_MYAVRLIB
#define HEADER_MYAVRLIB
#include <avr/io.h>

void my_delay (int tps);
void try_bootloader (void);

#endif
```

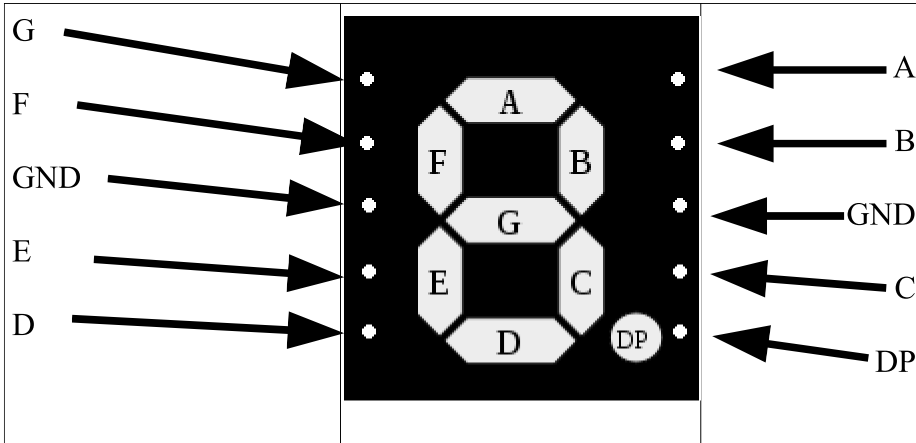
Un programme plus complexe : un compteur électrique

Maintenant que nous savons faire clignoter une LED (bel effort), nous allons tenter d'aller un petit peu plus loin en réalisant un compteur à l'aide de deux afficheurs 7 segments (compteur qui nous permettra donc de compter de 0 à 99).

L'afficheur 7 segments

Le modèle d'afficheur 7 segments que nous utilisons dans ce tutoriel, est un modèle à ANODE COMMUNE. C'est à dire que tous les segments sont reliés à la même masse.

Chaque patte est relié à un segment : ainsi pour illuminer le segment du milieu, il faut que la première patte en haut à gauche soit a 5V, et que la patte du milieu (droite ou gauche) soit à GND



« Multiplexage » des sorties

Afin de garder un code plus compact et d'économiser nos entrée-sorties, on va brancher les deux afficheurs sur LES MEME GPIO (ici le PORTB).
Seules les pattes reliées à la masse seront branchés sur des sorties différentes.
Voilà comment l'ensemble devrait fonctionner. Soit un afficheur A et un afficheur B :

Pour écrire un chiffre sur l'afficher A :

- On met la masse de l'afficheur A à 0 (on « allume » l'afficheur A)
- On met les segments souhaités à 1
- On met la masse de l'afficheur B à 1 (on « éteint » l'afficheur B)

Pour écrire un chiffre sur l'afficheur B :

- On met la masse de l'afficheur B à 0
- On met les segments souhaités à 1
- On met la masse de l'afficheur A à 1

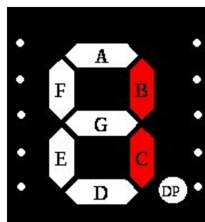
Or si l'on passe de l'un à l'autre suffisamment rapidement, avec la persistance rétinienne on à l'impression que les deux afficheurs sont constamment allumés.

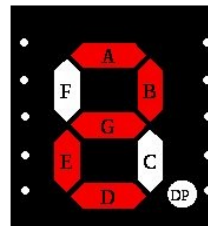
Afficher les nombres

Il est maintenant temps de s'attaquer au cœur du problème :

Pour afficher un '1', il faut allumer les segments B & C

Pour afficher un '2', il faut allumer les segments A, B, G, E, D





Comme nous l'avons dit, nous allons brancher les afficheurs sur le PORTB (le plus accessible) pour les segments de la façon suivante :

PB0 → G A ← PB7

PB1 → F B ← PB6

XX → GND GND ← XX

PB2 → E C ← PB5

PB3 → D DP ← PB4

Les pattes GND sont pour l'instant relié à XX car on les branchera sur d'autres ports.

Notre afficheurs étant actif à l'état haut, voici donc comment afficher les chiffres (de 0 à 9) selon le branchement spécifié ci-dessus :

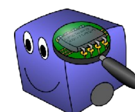
Chiffre	Segment actifs	PORTB								Équivalent (Hexa)
		7	6	5	4	3	2	1	0	
0	F A B C D E	1	1	1	0	1	1	1	0	EE
1	B C	0	1	1	0	0	0	0	0	60
2	A B G E D	1	1	0	0	1	1	0	1	CD
3	A B G C D	1	1	1	0	1	0	0	1	E9
4	F G B C	0	1	1	0	0	0	1	1	63
5	A F G C D	1	0	1	0	1	0	1	1	AB
6	A F E D C G	1	0	1	0	1	1	1	1	AF
7	A B C	1	1	1	0	0	0	0	0	E0
8	A B C D E F G	1	1	1	0	1	1	1	1	EF
9	A B C D F G	1	1	1	0	1	0	1	1	EB

Maintenant, si on a bien câblé l'afficheur sur les bons ports de notre AVR, si on veut afficher, par exemple un '5' il suffira d'écrire : PORTB = OxAB;

Création de notre bibliothèque

Nous avons maintenant tous les éléments pour écrire le fichier double7seg.c qui contient les fonctions nécessaires.

Avant cela, ajoutons juste que nous allons câbler la patte GND du premier afficheur à la patte PORTC 6 et la patte GND du second afficheur au PORTC 7



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

void base10digit (int n, int* tab_digits);
int display7seg (int display, int digit);

unsigned int code7seg[] ={0xEE,0x60,0xCD,0xE9,0x63,0xAB,0xAF,0xE0,0xEF,0xEB};

/* Display the digit on the selected display.
 * Returns 1 if the parameters are incorrect.
 * Returns 0 otherwise.
 */
int display7seg(int display, int digit)
{
    if( digit<0 || digit>9)
        return 1;

    switch(display)
    {
        case 1: //first 7seg
            PORTC &= ~_BV(7);
            PORTB = code7seg[digit];
            PORTC |= _BV(6);
            return 0;
            break;

        case 2: //second 7seg
            PORTC &= ~_BV(6);
            PORTB = code7seg[digit];
            PORTC |= _BV(7);
            return 0;
            break;

        default:
            return 1;
            break;
    }
}

/* Puts in tab_digits the digits of n in base 10
 * in a little endian fashion.
 * n must be < 100
 */
void base10digit(int n, int *tab_digits)
{
    tab_digits[0] = n%10;
    n /= 10;
    tab_digits[1] = n%10;
}
```

Petit analyse du code

1. Le tableau **code7seg[]** : contient les valeurs en hexa du tableau de la partie précédente.

2. La fonction **display7seg(int display, int digit) : p**ermet d'afficher un nombre sur un afficheur.
   Le int display permet de sélectionner l'afficheur (1 ou 2).
   Le int digit permet de choisir le nombre à afficher.

Ensuite les lignes à l'intérieur de chaque case, ne sont que la version « C » de ce que nous avons explicité plus tôt :

Pour écrire un chiffre sur l'afficher A:

- On met la masse de l'afficheur A à 0
- On met les segment souhaités à 1
- On met la masse de l'afficheur B à 1

Pour écrire un chiffre sur l'afficheur B:

- On met la masse de l'afficheur B à 0
- On met les segment souhaités à 1
- On met la masse de l'afficheur A à 1

3. Enfin la fonction **base10digit(int n, in *tab_digits)** : Permet de diviser un nombre à deux chiffres avec une composante dizaine et une composante unité.

On peut maintenant créer le header correspondant:

```
#ifndef HEADER_DOUBLE7SEG
#define HEADER_DOUBLE7SEG

void base10digit(int n, int* tab_digits);
int display7seg(int display, int digit);

#endif
```

Ecriture du main:

Il ne nous « reste plus qu'à » écrire le main qui permet de réaliser notre compteur

```
#include "myavrlib.h"
#include "double7seg.h"
#define T_MAX 20

int main (void)
{
    int tab[2]= {0, 0};
    int n=0;
    int t=0; //time_counter
    try_bootloader();

    /* Set PORTB, PC6 and PC7 for output. */
    DDRB = 0xFF;
    DDRC |= 0xC0;

    PORTC &= ~_BV(7);
    PORTC &= ~_BV(6);

    while (1)
    {
        display7seg(1, tab[1]);
        my_delay(1000);
```

```
        display7seg(2, tab[0]);
        my_delay(1000);

        t++;
        if (t>T_MAX)
        {
            t = 0;
            base10digit(n, tab);
            n = (n+1)%100;
        }
    }
    return 0;
}
```

Le code est très simple à comprendre.
La dernière partie de la boucle « while » sert juste à gérer la vitesse de notre compteur puis à mettre la valeur de ce que l'on souhaite afficher dans notre tableau.
Si on veut le faire compter plus vite, il suffirait de modifier la valeur de T_MAX.