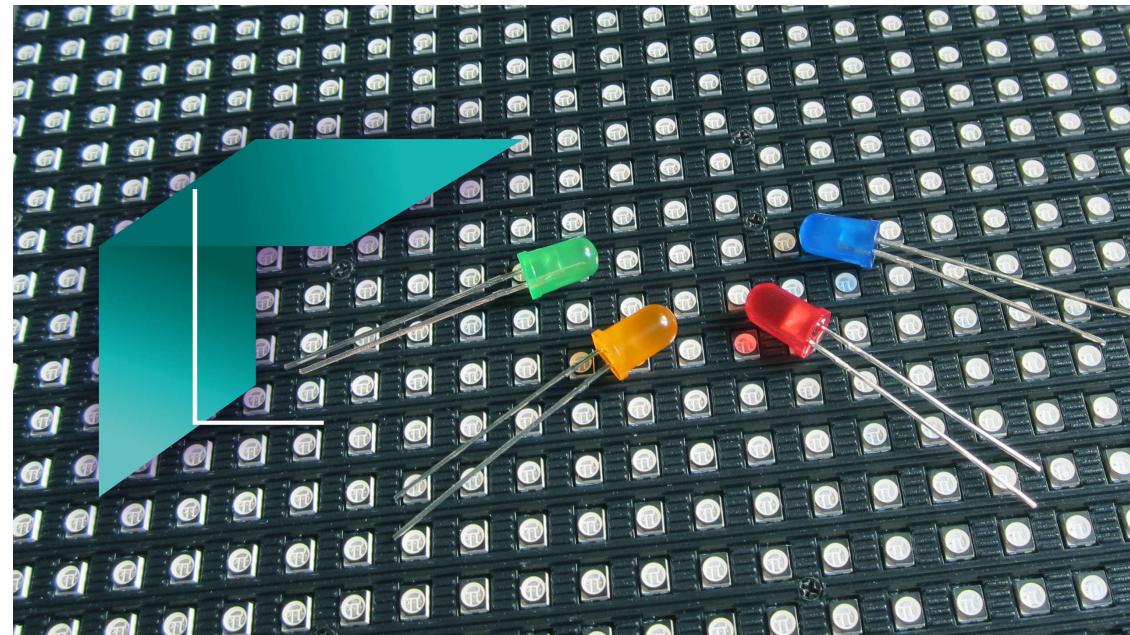


MOOC

Enseignes et afficheurs à LED



Pierre-Yves Rochat

EPFL

DRAFT
2016-03-20
01:16:01

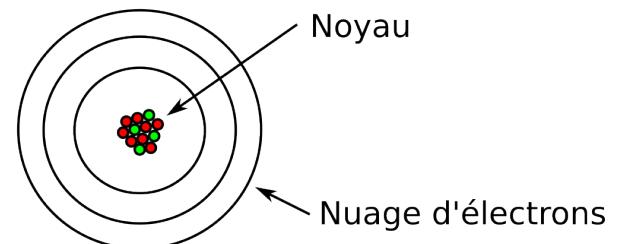
CHAPITRE 1

CIRCUITS ÉLECTRIQUES

Pierre-Yves Rochat, EPFL
rév 2016/01/20

1.1 ÉLECTRONS LIBRES ET COURANT ÉLECTRIQUE

On se souvient que la matière est composée d'**atomes**. Les atomes ont un noyau, composé de protons et de neutrons, ainsi qu'un nuage d'électrons, constitués en couches successives.



Structure d'un atome

Dans certains matériaux, les électrons de la dernière couche sont capables de se déplacer d'un atome à l'autre. On parle d'**électrons libres**. Ce mouvement est généralement désordonné. Lors qu'il est ordonné, on parle de **courant électrique**.

Les électrons ayant une charge négative, ils se **déplacent** du *moins* vers le *plus*. Ils se déplacent lentement, de l'ordre de quelques centimètres par heure.

Le courant électrique se déplace conventionnellement du *plus* vers le *moins* : on a découvert le courant électrique avant de connaître l'existence des atomes et

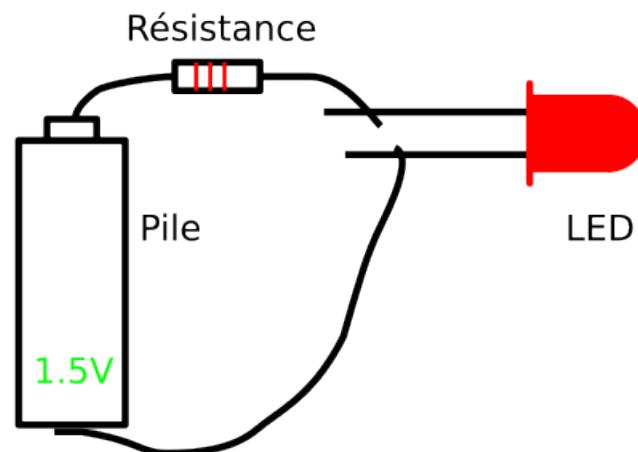
des électrons ! Le courant électrique se **propage** très rapidement, à une vitesse proche de la vitesse de la lumière, qui est de 300'000 km par seconde.

Les matériaux qui permettent ce phénomène du courant électrique sont appelés **conducteurs**. Il s'agit principalement des métaux, dont le cuivre et l'aluminium sont les plus couramment utilisés.

1.2 TENSION ÉLECTRIQUE

En présence d'une *force électromotrice*, appelée aussi **tension électrique**, un courant va se produire dans un conducteur. Par exemple, on trouve une tension entre les deux bornes d'une pile électrique. En établissant un **circuit électrique**, du courant va pouvoir circuler.

Nous verrons plus tard ce qu'est une LED (Light-Emitting Diode, diode électroluminescente ou *diode lumineuse*). En attendant, réalisons le montage ci-dessous. Le courant électrique produit par la pile va circuler à travers la résistance et la LED, qui va émettre alors de la lumière.



Circuit électrique

Les électriciens et les électroniciens ont l'habitude de dessiner des **schémas** en utilisant des symboles. Ils ne ressemblent pas toujours à la forme des composants utilisés ! Voici le schéma correspondant au montage ci-dessus :

Résistance

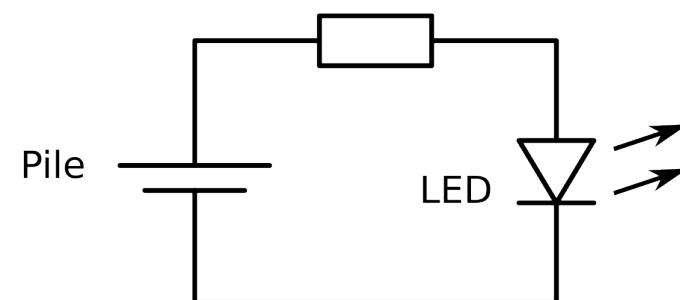


Schéma électrique

1.3 UNITÉS ÉLECTRIQUES

La **tension électrique** s'exprime en **Volt**, noté V. Une pile ordinaire a une tension d'environ 1,5 V. Dans beaucoup de pays, la tension fournie dans les maisons est de 230 V. Elle est de 115 V aux USA. Notons qu'il ne s'agit pas d'une tension continue, mais d'une tension alternative, dont la fréquence est de 50 alternances par seconde (50 Hz, 60 Hz aux USA).

En dessus de 25 V, une tension électrique est **dangereuse** pour le corps humain. En effet, l'eau se trouvant le corps est légèrement conductrice, vu qu'elle n'est pas pure. Or le cœur est très sensible aux courants électriques qui le traverse.

Dans les enseignes et afficheurs à LED, la tension est généralement inférieure à 25 V, sauf bien entendu au niveau de l'alimentation électrique. Toutes les précautions doivent donc être prises dans ce cas.

La tension électrique se mesure toujours entre deux points. Souvent, un point est considéré comme un point de référence. Il s'appelle la **masse** (*ground*). On peut alors dire, par abus de langage, que la tension *en un point* à telle valeur : c'est en fait la tension entre ce point et la masse.

Comme la tension se mesure entre deux points, c'est facile de la mesurer sans modifier le circuit. L'appareil s'appelle un **voltmètre**. Les multi-mètres proposent tous ces fonctions.

Le **courant électrique** s'exprime en **Ampère**, noté A.

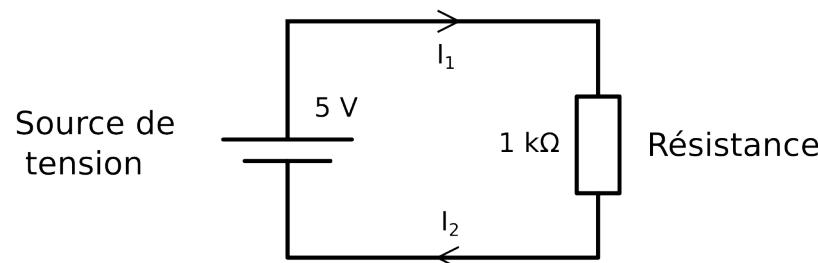
Le courant qui traverse une LED ne dépasse généralement pas 10 mA. Mais une enseigne composée de plusieurs milliers de LED peut nécessiter des courants importants de quelques dizaines d'Ampère.

Le diamètre d'un fil électrique ne dépend pas de la tension, mais du courant qui le traverse. Il faudra donc dimensionner correctement les fils. Par exemple, un fils de cuivre de 1 mm^2 de section n'est pas prévu pour un courant dépassant 16 A.

La mesure du courant nécessite de couper le circuit et d'y insérer l'appareil de mesure. C'est la raison pour laquelle la mesure directe du courant est beaucoup plus rare. On préférera une mesure indirecte.

1.4 RÉSISTANCE ÉLECTRIQUE

Dans le schéma ci-dessous, on voit un circuit électrique réalisé avec une source de tension et une **résistance**.



Source de tension et résistance

Le courant électrique est le même en tout point de ce circuit : $I_1 = I_2$. Ce courant est d'autant plus grand que la résistance est petite : c'est la loi d'Ohm.

$$I = U / R, \text{ avec :}$$

- U : la tension électrique, exprimée en Volt [V]
- I : le courant électrique, exprimé en Ampère [A]
- R : la résistance électrique, exprimée en Ohm [Ω]

Voici les deux autres expressions de la loi d'Ohm :

- $U = R \times I$
- $R = U / I$

On utilisera très souvent cette loi, par exemple pour calculer les résistances qui sont presque toujours associées aux LED.

1.5 ÉNERGIE ET PUISSANCE

On comprend intuitivement qu'une pile contient de l'**énergie** électrique. Une certaine quantité de cette énergie va être transformée en chaleur dans la résistance. Notons que le rendement est alors de 100% ! La **pouissance** est la quantité d'énergie par unité de temps :

$$P = U \times I, \text{ avec } P : \text{la puissance, exprimée en Watt [W]}$$

Dans notre premier montage avec la LED, une partie de l'énergie électrique est transformée en énergie lumineuse, le reste étant transformé en chaleur. Mais alors qu'une ampoule à incandescence a un rendement qui dépasse rarement 2%, une LED peut avoir un rendement qui approche les 30%.

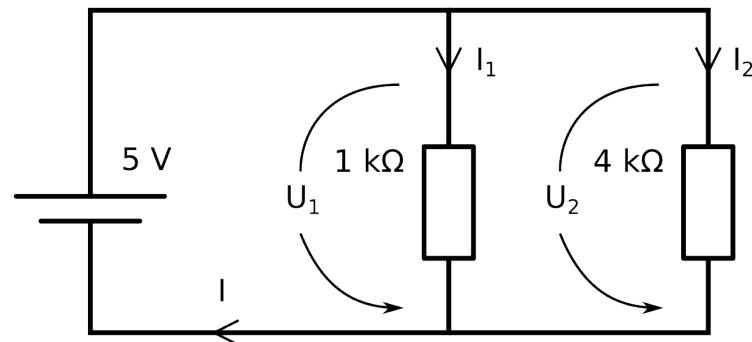
1.6 MONTAGE EN PARALLÈLE

Dans le schéma ci-dessous, deux résistances ont été montée *en parallèle*. La tension aux bornes de chaque résistance est la tension de la pile :

$$U = U_1 = U_2 = R_1 \times I_1 = R_2 \times I_2$$

Quel est alors la valeur du courant I ? On admet facilement que les électrons ne peuvent pas *sortir* du fil. On a donc $I = I_1 + I_2$

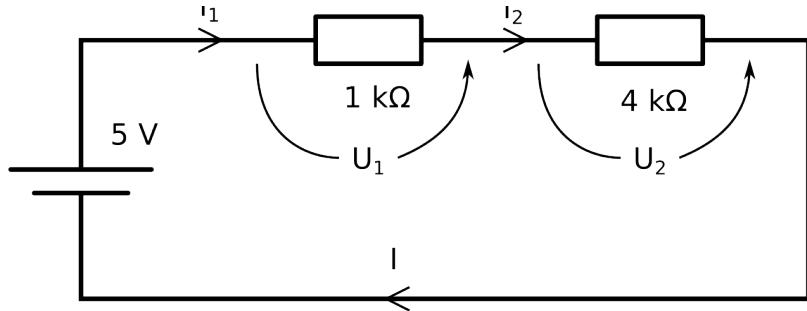
...



Résistances en parallèle

1.7 MONTAGE EN SÉRIE

Dans le schéma ci-dessous, deux résistances ont été montée *en série* : le courant passe par R_1 , puis par R_2 .



Résistances en série

Le courant à travers les deux résistances est le même. $I_1 = I_2 = I$. On peut calculer la tension aux bornes de chaque résistance :

$$U_1 = R_1 \times I_1 \text{ et } U_2 = R_2 \times I_2$$

...

CHAPITRE 2

NOMBRES ET CHAMPS DE BITS

Pierre-Yves Rochat, EPFL
et Yves Tiecoura, INP-HB
rév 2015/12/25

Ce document n'est pas à jour, il n'a pas encore été adapté aux modifications des diapositives pour la vidéo.

2.1 BASCULES ET REGISTRES

Une LED peut être, à un instant donné, complètement éteinte ou allumée à un degré d'intensité ajustable. Cet état est généralement mémorisé par une bascule.

Les enseignes et afficheurs à LED utilisent beaucoup de registres, composés de bascules. Ces registres sont souvent de 8 bits ou de 16 bits, mais on en trouve aussi avec des valeurs beaucoup plus grandes.

L'état de chaque LED est aussi fréquemment mémorisé dans la mémoire d'un microcontrôleur. Le processeur du microcontrôleur reçoit des données, il les traite et diffuse les résultats sur ses sorties. Tous les systèmes informatiques travaillent en binaire. Historiquement, des systèmes ternaires (fonctionnant en base 3) ont été développés, mais ils sont extrêmement rares et pratiquement introuvables sur le marché.

....

2.2 CHAMP DE BIT

On appelle “bit” un symbole binaire. Il peut prendre les valeurs 0 et 1, qui peuvent aussi s'appeler *vrai* et *faux*, *allumé* et *éteint*, etc. C'est un mot-valise composé de la fusion des mots de la locution anglaise *binary digit* ou *chiffre binaire* en français.

On désigne par mot binaire, ou champ de bits, un ensemble de bits. Des opérations logiques peuvent s'appliquer à ces champs de bits (NON, ET, OU, etc.). Elles seront étudiées plus loin dans ce cours.

2.3 NOMBRES BINAIRES

Un champ de bit peut aussi représenter un nombre. La numération binaire est bien connue :

Codage binaire		Poids des bits		
Binaire	Décimal	Rang	Valeur	Décimal
0	0	0	1	$1 = 2^0$
1	1	1	2	$= 2^1$
1 0	2	2	4	$= 2^2$
1 1	3	3	8	$= 2^3$
1 0 0	4	4	16	$= 2^4$
1 0 1	5	5	32	$= 2^5$
1 1 0	6	6	64	$= 2^6$
1 1 1	7	7	128	$= 2^7$
1 0 0 0	8	8	256	$= 2^8$
1 0 0 1	9	9	512	$= 2^9$
1 0 1 0	10	10	1024	$= 2^{10}$
1 0 1 1	11			
1 1 0 0	12			
1 1 0 1	13			
1 1 1 0	14			
1 1 1 1	15			
1 0 0 0 0	16			

Figure : Numération binaire

... poids...

Par exemple, 2345 (en décimal) s'exprime par 100100101001 en nombre binaire. Preuve en est :

$$\begin{aligned} 1 \times 1 &+ 0 \times 2 + 0 \times 4 + 1 \times 8 + 0 \times 16 + 1 \times 32 \\ &+ 0 \times 64 + 0 \times 128 + 1 \times 256 + 0 \times 512 + 0 \times 1024 + 1 \times 2048 = 2345 \end{aligned}$$

Pour coder un nombre décimal en binaire, on effectue des divisions entières successives par 2 jusqu'à ce que le quotient soit nul. Le premier reste est le poids faible, le dernier est le poids fort.

2.4 ARITHMÉTIQUE MODULAIRE

Lorsqu'un nombre est matérialisé dans un circuit électronique, il a forcément une taille limitée.

On peut utiliser ces nombres pour des calculs. Mais il faut être attentif au fait qu'ils ont une limite dans leur taille. En étudiant les mathématiques, on prend l'habitude d'utiliser des nombres immatériels, qui peuvent être aussi grands que nécessaire. Lorsqu'un nombre doit être matérialisé dans un dispositif physique, dans notre cas dans un registre ou une mémoire d'ordinateur, sa taille est forcément limitée. On se trouve alors en face d'une arithmétique différente, l'*arithmétique modulaire*.

Pour bien la comprendre, prenons l'exemple des nombres représentés par 3 bits. Ils peuvent prendre 8 valeurs ($8 = 2^3$).

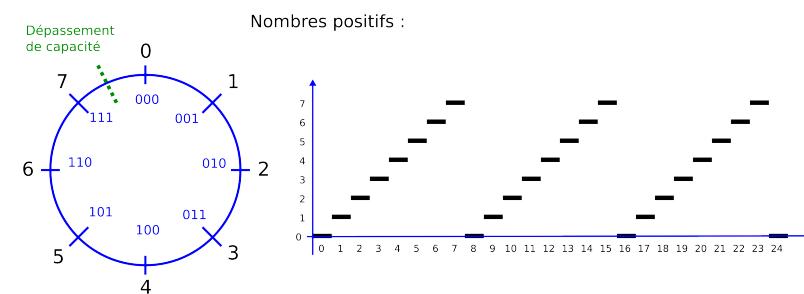


Figure : Nombres positifs sur 3 bits

On voit qu'il n'est possible de représenter qu'un nombre limité de valeurs. S'il s'agissait de nombres de 8 bits, on aurait un choix de 256 valeurs (de 0 à 255). Pour des nombres de 16 bits, on aurait 65'536 valeurs (de 0 à 65'535).

Sur le cercle qui représente l'ensemble des valeurs possibles, l'incrémentation (addition de 1) correspond à une avance dans un sens. Lorsqu'on dépasse la valeur la plus grande (7 dans le cas de 3 bits), on retrouve la valeur 0. On a franchi la limite du dépassement de capacité (*overflow* en anglais).

La décrémentation (soustraction de 1) correspond au sens contraire. Un dépassement de capacité se produit aussi lors du passage de 0 à la valeur la plus grande.

Les opérations arithmétiques classiques sur les nombres entiers doivent donc tenir compte du dépassement de capacité. Il s'agit de l'arithmétique modulaire. Dans le cas de 3 bits le résultat est donné *modulo 8*. L'opération Modulo correspond aussi au reste de la division entière.

Prenons quelques exemples :

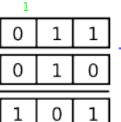
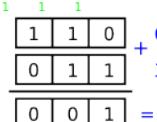
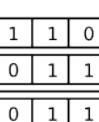
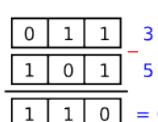
			
---	---	---	---

Figure : Opérations sur des nombres de 3 bits

2.5 NOMBRES SIGNÉS

Dans l'usage courant, les nombres peuvent être positifs ou négatifs. Est-ce possible de les représenter en binaire ? Il existe beaucoup de manières de le faire et plusieurs d'entre elles ont été utilisées au cours de l'histoire de l'informatique. Mais c'est la représentation appelée *en complément à 2* qui est de loin la plus utilisée actuellement.

Voici une figure qui en explique le principe, appliqué à des nombres de 3 bits :

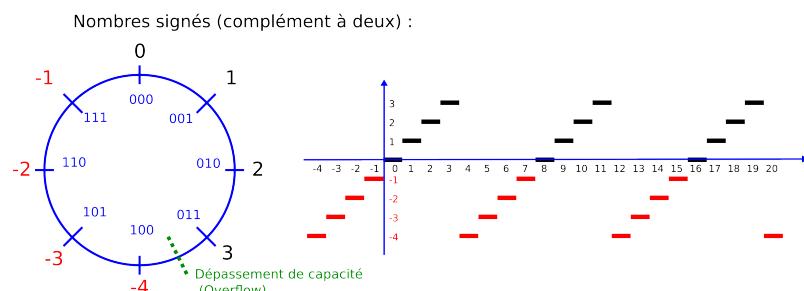


Figure : Nombres positifs et négatifs sur 3 bits

On remarque que le nombre est négatif lorsque le bit de poids fort (celui de gauche) a pour valeur 1.

2.6 TYPES EN C

Les langages de programmation définissent aussi des types avec des nombres entiers d'une taille limitée. Les types permettent d'allouer l'espace mémoire optimal à chaque format.

Les types "historiques" du langage C sont :

Type	Description
char	mot de 8 bits *
signed char	mot de 8 bits signé
unsigned char	mot de 8 bits positif
int	mot généralement de 16 bits *
signed int	mot de 16 bits signé
unsigned int	mot de 16 bits positif
long int	mot généralement de 32 bits *
signed long int	mot de 32 bits signé
unsigned long int	mot de 32 bits positif

* (signé ou non signé, selon les réglages du compilateur)

Ces notations sont souvent ambiguës. On préfère maintenant une notation plus claire, standardisée depuis la version C99 de 1999 :

Type	Description
int8_t	mot de 8 bits signé
uint8_t	mot de 8 bits positif
int16_t	mot de 16 bits signé
uint16_t	mot de 16 bits positif
int32_t	mot de 32 bits signé
uint32_t	mot de 32 bits positif

C'est cette notation que nous utiliserons dans ce cours. Les opérations arithmétiques disponibles pour ces types sont :

Opération	Symbole
l'addition	+
la soustraction	-
la multiplication	*
la division entière	/
le reste de la division entière, appelée aussi modulo	%

2.7 HEXADÉCIMAL

Dans notre exemple précédent, le nombre 2345, qui est composé de quatre chiffres en décimal, nécessite déjà 12 bits en binaire. L'écriture dans cette base est fastidieuse pour l'être humain !

En utilisant une autre base qui est aussi une puissance de 2, on bénéficie d'une conversion très simple en base 2. La base la plus couramment utilisée est la base 16, appelée "hexadécimal".

0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

Figure : Hexadécimal

Pour convertir un nombre binaire en hexadécimal, on le sépare en tranches de 4 bits de la droite vers la gauche et on complète à gauche avec des zéros non significatifs.

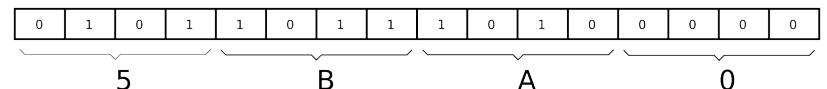


Figure : Conversion binaire-hexadécimal

Pour convertir un nombre hexadécimal en binaire, il faut simplement écrire les 4 valeurs binaires de chaque chiffre hexadécimal.

2.8 CODAGE DES CARACTÈRES

Parmi les données traitées par les systèmes informatiques (par exemple un microcontrôleur), on trouve souvent des caractères. Pour représenter les caractères, on utilise des tables de transcodage vers le binaire.

Le codage ASCII (*American Standard Code for Information Interchange*) sur 7 bits a été standardisé dans les années 1960.

Table de codage ASCII

	00 _h	01 _h	02 _h	03 _h	04 _h	05 _h	06 _h	07 _h	08 _h	09 _h	0A _h	0B _h	0C _h	0E _h	0E _h	0F _h	
	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	
00 _h	0.	NUL	SOH	STX	ETX	EOT	ENQ	ENQ	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10 _h	16.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20 _h	32.	!	"	#	\$	%	&	'	()	*	,	+	,	-	.	/	
30 _h	48.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40 _h	64.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50 _h	80.	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	
60 _h	96.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
60 _h	112.	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Pour trouver le code d'un caractère, ajouter la valeur de la ligne et celle de la colonne *en décimal* ou *en hexadécimal*

Figure : Caractères ASCII

Malheureusement, les caractères accentués n'étant pas standardisés par la table ASCII, un grand nombre de tables sont apparues, qui cohabitent encore à notre époque de l'internet.

Une des tables les plus souvent utilisées est l'UTF-8.

CHAPITRE 3

CIRCUITS LOGIQUES COMBINATOIRES

3.1 MAMADOU LAMINE NDIAYE, ESP DAKAR

Document en cours de relecture, version du 2016/01/25

3.2 ÉLÉMENTS DE BASE DES SYSTÈMES LOGIQUES

Dans les circuits logiques, le codage des informations utilise deux niveaux de tension et un état logique est associé à chaque niveau de tension. Les systèmes numériques (logiques) fonctionnent en binaire c'est à dire que les variables discrètes (entrées ou sorties) du système ne prennent que deux valeurs 0 ou 1. L'état d'une variable logique peut être *faux* ou *vrai*. On représente l'état faux par 0 et l'état vrai par 1.

Par exemple, pour la technologie électrique TTL, le 0 logique correspond à une tension entre 0 et 0,8V, alors que le 1 logique correspond à une tension entre 2,4 et 5V.

Si nous prenons l'exemple d'un moteur électrique, le 0 logique peut correspondre à l'état *Arrêt*, alors que le 1 logique peut correspondre à l'état *Marche*.

3.3 ALGÈBRE DE BOOLE

La logique binaire peut être représentée par l'Algèbre de BOOLE, qui permet de décrire dans un modèle mathématique le traitement et le fonctionnement des systèmes binaires. L'Algèbre de BOOLE est conçue autour d'opérations logiques de base :

- le complément logique : NON, représenté par un surligné
- l'addition logique : OU, représenté par le signe +

- produit logique : ET, représenté par le signe •
- et leurs dérivés (XOR, NAND, NOR, etc)

3.4 PORTES LOGIQUES

Les éléments de base des systèmes logiques sont des portes logiques que l'on peut assembler pour réaliser des fonctions logiques qui peuvent à leur tour être assemblées pour construire des machines numériques.

La seule porte intéressante à une entrée est l'inverseur :

Fonction	Symbol	Equation	Table de vérité						
NON		$S = \bar{A}$	<table border="1"> <tr> <th>A</th> <th>S</th> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	A	S	0	1	1	0
A	S								
0	1								
1	0								

Figure : Inverseur

La table de vérité qui est un tableau qui indique la valeur de la sortie pour toutes les combinaisons des entrées. Chaque ligne correspond à une combinaison des variables. Ici, elle n'a que deux lignes.

Les deux portes de base à 2 entrées sont la porte OU et la porte ET :

Fonction	Symbol	Equation	Table de vérité															
OU		$S = A + B$	<table border="1"> <tr> <th>A</th> <th>B</th> <th>S</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	A	B	S	0	0	0	0	1	1	1	0	1	1	1	1
A	B	S																
0	0	0																
0	1	1																
1	0	1																
1	1	1																

Figure : Porte OU

Fonction	Symbol	Equation	Table de vérité															
ET		$S = A \cdot B$	<table border="1"> <tr> <th>A</th> <th>B</th> <th>S</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	A	B	S	0	0	0	0	1	0	1	0	0	1	1	1
A	B	S																
0	0	0																
0	1	0																
1	0	0																
1	1	1																

Figure : Porte ET

Les tables de vérité ont alors 4 lignes.

Les autres portes logiques à deux entrées sont des dérivées des éléments de base troisième élément de base :

Fonction	Symbol	Equation	Table de vérité															
NOR		$S = \overline{A + B}$	<table border="1"> <tr> <th>A</th> <th>B</th> <th>S</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	A	B	S	0	0	1	0	1	0	1	0	0	1	1	0
A	B	S																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

Figure : Porte NOR

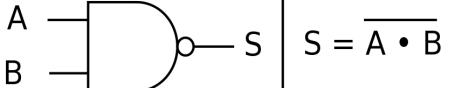
Fonction	Symbole	Equation	Table de vérité															
NAND		$S = \overline{A \cdot B}$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th>A</th> <th>B</th> <th>S</th> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	A	B	S	0	0	1	0	1	1	1	0	1	1	1	0
A	B	S																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

Figure : Porte NAND

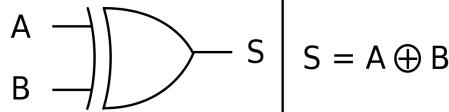
Fonction	Symbole	Equation	Table de vérité															
XOR		$S = A \oplus B$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <th>A</th> <th>B</th> <th>S</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table>	A	B	S	0	0	0	0	1	1	1	0	1	1	1	0
A	B	S																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

Figure : Porte Ou exclusif

3.5 EXPRESSION MATHÉMATIQUE D'UNE FONCTION LOGIQUE

Tout système logique peut être défini à l'aide d'une fonction logique (ou expression logique) qui représente les relations entre les variables de sortie en fonction des variables d'entrée.

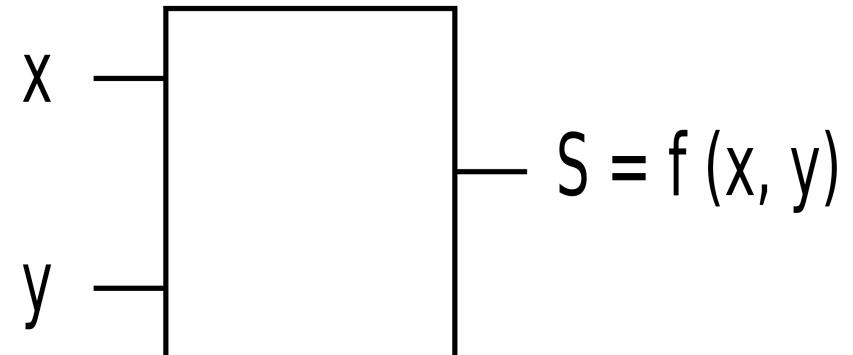


Figure : Fonction logique

La représentation des fonctions logiques se fait:

- de manière algébrique (représentation mathématique), déjà utilisé la représentation algébrique avec les équations logiques. Exemple :
- de manière tabulaire (table de vérité), une table de vérité est un tableau à 2^N lignes et $N+1$ colonnes. Chaque ligne correspond à une combinaison des N variables. Les N premières colonnes contiennent les variables et la colonne $N+1$ contient la valeur de la fonction
- de manière graphique avec un logigramme qui est un schéma illustrant l'expression d'une fonction logique avec les portes logiques (Exemple du schéma de la porte XOR).
- avec des diagrammes, plusieurs diagrammes existent, nous présenterons celui de Karnaugh lors de la leçon sur la synthèse des circuits logiques combinatoires.

L'expression en algèbre de Boole d'une fonction logique peut être faite sous deux formes : la forme *somme de produits* et la forme *produit de sommes*. L'expression sous la forme somme de produit est obtenue à partir de la somme de tous les lignes de la table de vérité où la fonction vaut 1. On appelle ces lignes *mintermes*.

A	B	C	S	Minterme	Maxterme
0	0	0	0	$A \cdot B \cdot C = 0$	$A + B + C = 0$
0	0	1	0	$\bar{A} \cdot B \cdot C = 0$	$\bar{A} + B + \bar{C} = 0$
0	1	0	1	$\bar{A} \cdot B \cdot \bar{C} = 1$	$\bar{A} + \bar{B} + C = 1$
0	1	1	1	$\bar{A} \cdot B \cdot C = 1$	$\bar{A} + \bar{B} + \bar{C} = 1$
1	0	0	0	$A \cdot \bar{B} \cdot \bar{C} = 0$	$\bar{A} + B + C = 0$
1	0	1	1	$A \cdot \bar{B} \cdot C = 1$	$\bar{A} + \bar{B} + \bar{C} = 1$
1	1	0	1	$A \cdot B \cdot \bar{C} = 1$	$\bar{A} + B + C = 1$
1	1	1	0	$A \cdot B \cdot C = 0$	$\bar{A} + \bar{B} + \bar{C} = 0$

Somme de produits :

$$S = \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot \bar{C}$$

Produit de somme :

$$S = (A + B + C) \cdot (\bar{A} + B + C) \cdot (\bar{A} + \bar{B} + C) \cdot (\bar{A} + \bar{B} + \bar{C})$$

Figure : Fonction à 3 entrées

L'expression sous la forme *produit de sommes* est obtenue en multipliant tous les *maxtermes* pour lesquels la fonction vaut 0.

3.6 PROPRIÉTÉS DE L'ALGÈBRE DE BOOLE

Il est souvent utile de simplifier ces expressions pour réduire le nombre de portes logiques et par conséquent le coût du matériel ou le temps de câblage. On utilise alors les propriétés de l'Algèbre de BOOLE.

Commutativité :

$$\mathbf{\Lambda}(A \bullet B = B \bullet A)$$

$$\mathbf{\Lambda}(A + B = B + A)$$

Idempotence :

$$\mathbf{\Lambda}(A \bullet A = A)$$

$$\mathbf{\Lambda}(A + A = A)$$

Constantes :

$$\mathbf{\Lambda}(A \bullet 0 = 0)$$

$$\mathbf{\Lambda}(A \bullet 1 = A)$$

$$\mathbf{\Lambda}(A + 0 = A)$$

$$\mathbf{\Lambda}(A + 1 = 1)$$

Complémentation :

$$\mathbf{\Lambda}(A \bullet \bar{A} = 0)$$

$$\mathbf{\Lambda}(A + \bar{A} = 1)$$

Distributivité :

$$\mathbf{\Lambda}(A \bullet (B + C) = (A \bullet B) + (A \bullet C))$$

$$\mathbf{\Lambda}(A + (B \bullet C) = (A + B) \bullet (A + C))$$

Associativité :

$$\mathbf{\Lambda}(A \bullet (B \bullet C) = (A \bullet B) \bullet C = A \bullet B \bullet C)$$

$$\mathbf{\Lambda}(A + (B + C) = (A + B) + C = A + B + C)$$

De Morgan :

$$\mathbf{\Lambda}(\overline{A \bullet B} = \overline{A} + \overline{B})$$

$$\mathbf{\Lambda}(\overline{A + B} = \overline{A} \bullet \overline{B})$$

Ces propriétés sont utilisées pour la simplification des fonctions logiques. Elles permettent d'obtenir une expression logique comportant un nombre minimal de termes, ainsi qu'un nombre minimal de variables dans chaque terme dans le but de simplifier la réalisation matérielle.

CHAPITRE 4

SYNTHÈSE DE CIRCUITS COMBINATOIRES

4.1 MAMADOU LAMINE NDIAYE, ESP DAKAR

Document en cours de relecture, version du 2015/01/25

4.2 SYSTÈME COMBINATOIRE

C'est un système où chacune des sorties est une combinaison logique des entrées à l'instant seulement. Une sortie ne dépend que des combinaisons d'entrée. Pour une combinaison des variables d'entrée, l'état de la sortie est unique.

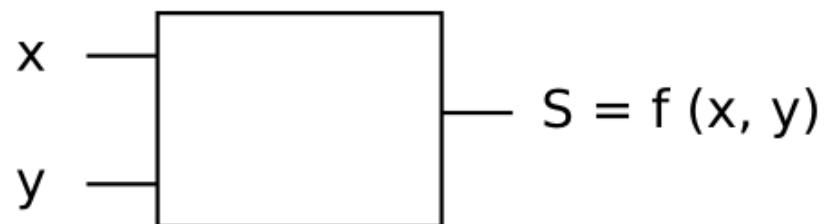


Figure : Système combinatoire

4.3 MÉTHODOLOGIE DE SYNTHÈSE DES CIRCUITS COMBINATOIRES

La méthodologie de synthèse des circuits combinatoires se fait en plusieurs étapes:

- Analyse du cahier des charges
- Identification des variables d'entrées et de sorties
- Représentation du problème sous forme de table de vérité
- Établissement de la ou des fonctions résultantes
- Simplification et établissement de logigramme
- Prototypage d'essai et réalisation finale

A partir d'un cahier des charges, il faut analyser le problème pour une bonne compréhension permettant sa traduction en circuit logique. L'étude du cahier des charges permet d'identifier les variables du système (entrée, sortie). Lorsque le nombre de variables est faible, on peut utiliser directement la table de vérité. Dans le cas contraire le circuit est décomposé en différents blocs fonctionnels que l'on peut étudier séparément. Dans tous les cas il est nécessaire d'établir la ou les fonctions logiques et éventuellement procéder à leur simplifications avant d'envisager le prototype et la réalisation finale.

4.4 SIMPLIFICATION DES FONCTIONS LOGIQUES

La simplification des fonctions logiques cherche à obtenir une expression logique comportant un nombre minimal de termes, ainsi qu'un nombre minimal de variables dans chaque terme dans le but de simplifier et de réduire le coût de la réalisation matérielle. Plusieurs méthodes existent :

4.5 SIMPLIFICATION ALGÉBRIQUE

Elle utilise les théorèmes généraux de l'Algèbre de BOOLE et aux identités remarquables (formules élémentaires) que nous avons vu lors de la leçon précédente.

Le principe consiste à :

- Regroupement des termes ayant des variables communes et mises en facteur
- Réplication de termes existants
- Suppression de termes superflus

Exemple

$$S = A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C$$

En factorisant et en utilisant les propriétés de BOOLE on a :

$$S = A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$$

$$S = \overline{C} (A \cdot \overline{B} + A \cdot B) + B \cdot C (A + \overline{A})$$

$$S = \overline{C} \cdot A + C \cdot B$$

4.6 SIMPLIFICATION PAR TABLEAU DE KARNAUGH

Le tableau de Karnaugh est une forme particulière de la table de vérité. Il comprend 2^N cases, N étant le nombre de variables d'entrées de la fonction considérée. Dans chaque case est inscrite la valeur de la sortie. Les variables sont disposées selon le code GRAY ou code binaire réfléchi. Lorsque l'on passe d'une case à la case adjacente, une seule variable change. Simplification : Pour exprimer la sortie S:

- Effectuer des groupements de cases adjacentes successivement contenant des 1.
 - La taille d'un groupe est une puissance de 2^k (8,4,2,1...). On cherche toujours le groupement maximal.
 - Le résultat d'un groupement est le produit des variables constantes (qui ne changent pas).
 - Le résultat final est la somme des résultats des groupements.
 - Une même case peut appartenir à deux groupements différents.
- Exemple: Considérons l'exemple de la table de vérité suivante :

Entrées			Sorties
C	B	A	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$S = A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$$

Simplification forme somme de produit

$$S = A \cdot \bar{C} + B \cdot C$$

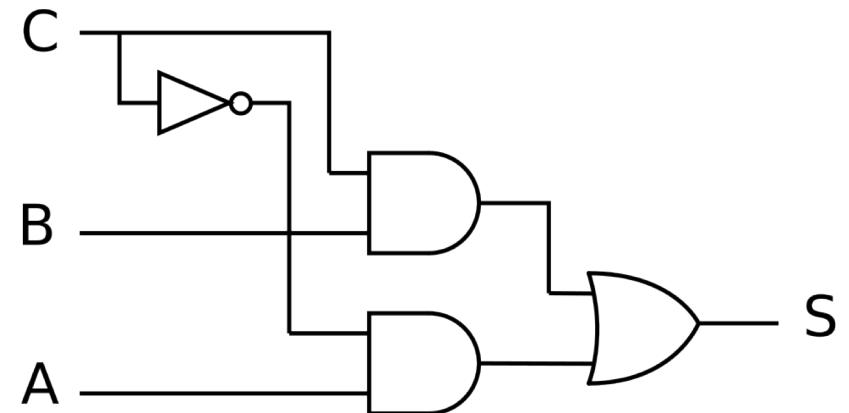
Tableau de Karnaugh

BA C	00	01	11	10
0	0	1	1	0
1	0	0	1	1

En regroupant les 1, on a $S = A \cdot \overline{C} + B \cdot C$

BA \ C	00	01	11	10
0	0	1	1	0
1	0	0	1	1

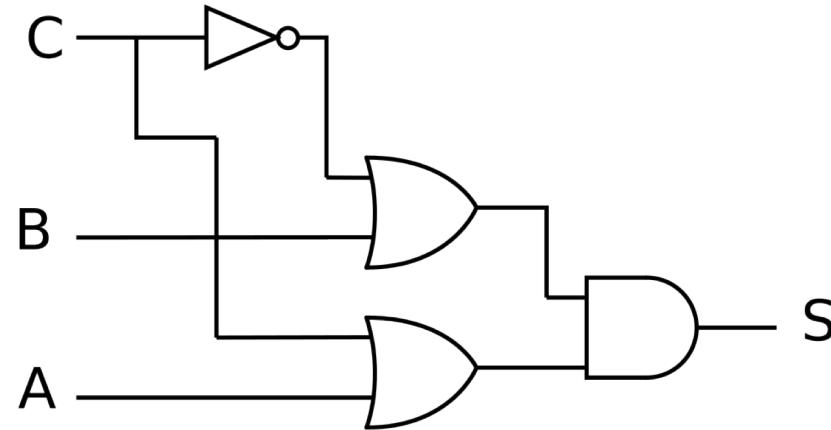
Schéma logique



En utilisant l'expression sous forme de produit de somme on regroupe les 0 et on a :

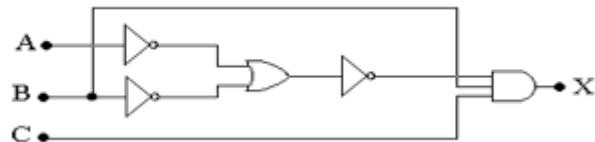
$$S = (A + C) \cdot (B + \bar{C})$$

Schéma logique

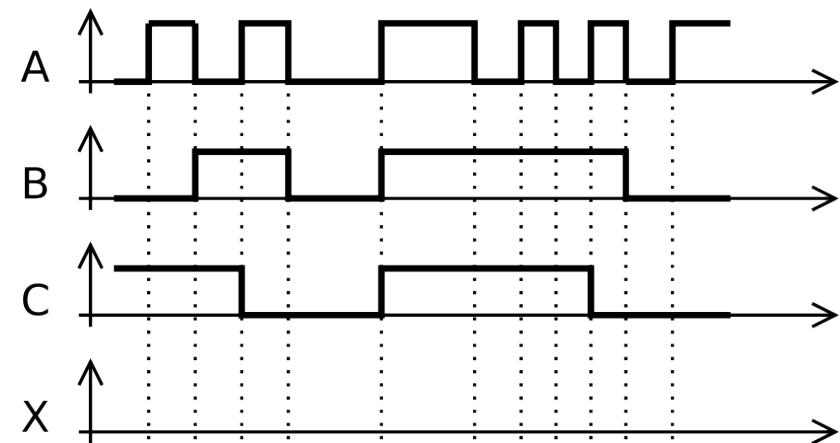


4.7 DIAGRAMME TEMPOREL

Tracez les formes d'onde de sortie pour la sortie X pour la figure ci-dessous.



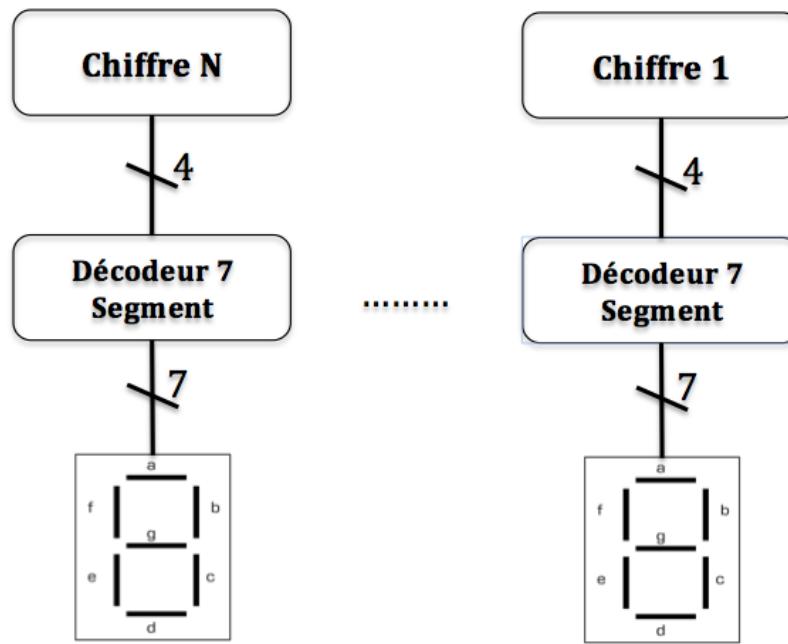
$$X = (\bar{A} + \bar{B}) \cdot B \cdot C = A \cdot B \cdot C$$



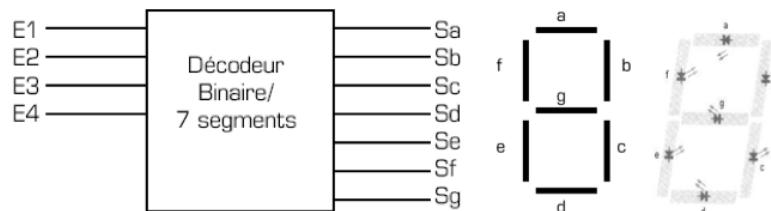
Dans un système numérique de traitement d'informations plusieurs opérations sont nécessaires : le codage et le décodage (transposition de données d'un code à un autre), le multiplexage (aiguillage des données) etc., Ces opérations sont effectuées par des circuits combinatoires spécialisés. Nous allons maintenant regarder comment synthétiser un décodeur 7 segments.

4.7.1 Décodeur 7 segments

Un décodeur est un circuit qui permet de passer d'un code à un autre. Il comporte N entrées et M sorties. Le décodeur 7 segments permet de transcoder chaque chiffre séparément : unité, dizaine centaine etc. Il s'appuie sur le codage BCD (Décimal codé en Binaire) qui transforme les nombres décimaux en remplaçant chacun des chiffres décimaux par 4 chiffres binaires. Chaque chiffre (de 0 à 9) est codé en binaire sur 4 bits (E1, E2, E3 et E4). Pour un nombre à 3 chiffres (ex : 123), on aura 4 valeurs binaires pour chaque chiffre : centaines, dizaines et unités : 0001 0010 0011. Le décodeur permet de transformer ces 4 bits provenant des 4 fils en entrée (E1, E2, E3, E4) du décodeur en 7 bits à la sortie du décodeur. Chaque fil de la sortie du décodeur est connecté à une LED de l'afficheur (a, b, c, d, e, f, g).



Chaque afficheur alimente ses 7 segments (leds) en fonction du code binaire à la sortie du décodeur correspondant aux chiffres de 0 à 9 ainsi que les lettres de A à F.



4.7.1.1 Table de vérité du Décodeur 7 Segments en décimal

Avec les 4 variables d'entrée on a 24 soit 16 cases dans la table de vérité mais nous n'avons besoin que des dix premières valeurs (0 à 9). Il y a donc 6 valeurs à

éliminer 10, 11, 12, 13, 14 et 15. On mettra un x ($x = 0$ ou 1) au niveau des sorties pour ces différentes valeurs.

Valeur	Entrées				Sorties							
	Code	E4	E3	E2	E1	Sa	Sb	Sc	Sd	Se	Sf	Sg
0	0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	0	1	1	0	0	0	0
2	0	0	1	0	0	1	1	0	1	1	0	1
3	0	0	1	1	0	1	1	1	1	0	0	1
4	0	1	0	0	0	0	1	1	1	0	0	1
5	0	1	0	1	0	1	0	1	1	0	1	1
6	0	1	1	0	0	1	0	1	1	1	1	1
7	0	1	1	1	0	1	1	1	0	0	0	0
8	1	0	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	0	1	1	1	1	0	1	1
10	1	0	1	0	0	x	x	x	x	x	x	x
11	1	0	1	1	0	x	x	x	x	x	x	x
12	1	1	0	0	0	x	x	x	x	x	x	x
13	1	1	0	1	0	x	x	x	x	x	x	x
14	1	1	1	0	0	x	x	x	x	x	x	x
15	1	1	1	1	0	x	x	x	x	x	x	x

4.7.1.2 Equations logiques des 7 sorties en décimal

Segment Sa

E2E1	00	01	11	10
E4E3				
00	1	0	1	1
01	0	1	1	1
11	1	1	1	1
10	1	1	1	1

E2E1 E4E3	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	1	1	1	1
10	1	1	1	1

$$Sa = \overline{E1.E3} + E4 + E2 + E1.E3$$

4.7.1.3 Simplification des équations logiques des 7 segments

Segment Sa

E2E1 E4E3	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	1	1	1	1
10	1	1	1	1

$$Sa = \overline{E1.E3} + E4 + E2 + E1.E3$$

Segment Sb

E2E1 E4E3	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	1	0	1	0
10	1	1	1	1

$$Sb = \overline{E3} + E1.E2 + \overline{E1.E2}$$

Segment Sc

E2E1 E4E3	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	0

$$Sc = E3 + E1 + \overline{E2}$$

Segment Sd

E2E1 E4E3	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	1	1	1	1
10	1	1	1	1

$$Sd = \overline{E1.E3} + E4 + \overline{E1.E2} + E2.\overline{E3} + E1.\overline{E2}.E3$$

Segment Se

E2E1 E4E3	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	0	0	0	1
10	1	0	0	1

$$Se = \overline{E1.E3} + \overline{E1.E2}$$

Segment Sf

E2E1 E4E3	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	1	1	1	1
10	1	1	1	1

$$Sf = E4 + \overline{E1.E2} + \overline{E1.E3} + \overline{E2.E3}$$

Segment Sg

E2E1 E4E3	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	1	1	1	1
10	1	1	1	1

$$Sg = E4 + \overline{E1.E2} + E2.\overline{E3} + \overline{E2.E3}$$

Le logigramme du décodeur pourra être réalisé à partir de ces équations.

CHAPITRE 5

INTRODUCTION AU TRANSISTOR

*Yves Tiecoura, INP-HB Yamoussoukro
rév 2015/11/11*

5.1 UNE INVENTION DE PREMIER PLAN

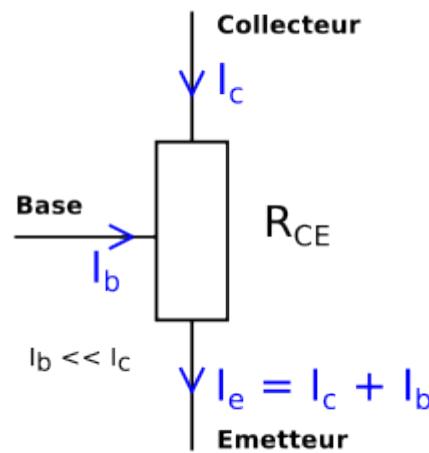
Pour réaliser des enseignes et afficheurs à LED, il est très souvent nécessaire d'utiliser l'élément le plus emblématique de l'électronique : le transistor.

La mise au point de ce dispositif à semi-conducteur a marqué une étape importante dans l'évolution de l'électronique et de l'humanité en général. Son invention date de 1948 et ses inventeurs, William Shockley, John Bardeen et Walter Houser Brattain ont reçu le prix Nobel de physique pour leurs travaux en 1956.

Le transistor est utilisé en électronique comme amplificateur ou comme interrupteur. Les premiers transistors étaient fabriqués avec du germanium, mais par la suite, c'est le silicium qui a été principalement utilisé. D'autres matériaux semi-conducteurs sont utilisés pour certaines applications, tel l'arsénure de gallium (GaAs).

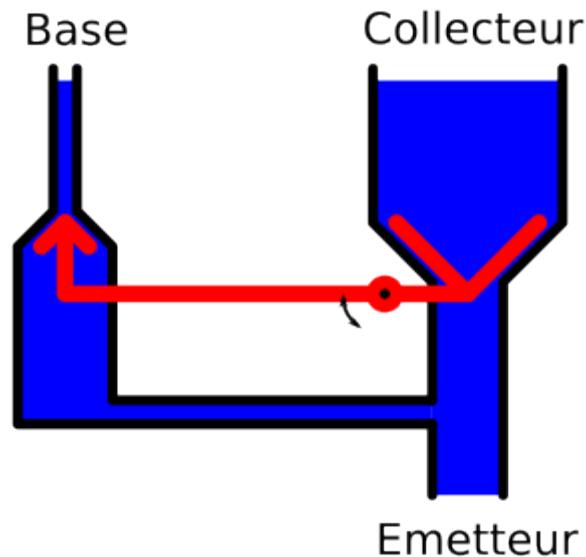
5.2 PRÉSENTATION

Le transistor se présente sous la forme d'un composant à trois broches, désignées par **base**, **collecteur** et **émetteur**. *Transistor* est un mot-valise formé par la fusion des mots de la locution anglaise *transfert resistor* ou *transrésistance* en français. Il est appelé ainsi, car il transfère un courant à travers une résistance. La figure suivante montre le transistor comme un tripôle.



Dans un transistor, la valeur de la résistance R_{CE} entre le collecteur et l'émetteur varie en fonction du courant I_b qui circule entre la base et l'émetteur. Une augmentation du courant de base provoque une diminution de la résistance R_{CE} , ce qui permet une augmentation du courant I_c qui entre dans le collecteur. C'est ce qu'on appelle l'*effet transistor*.

Cette similitude avec un système hydraulique aide à comprendre le principe :



Analogie hydraulique

On sait que $U = R \times I$ (Loi d'Ohm). Une variation du petit courant I_b provoque une variation du grand courant I_c .

Dans certaines conditions, cette variation est linéaire :

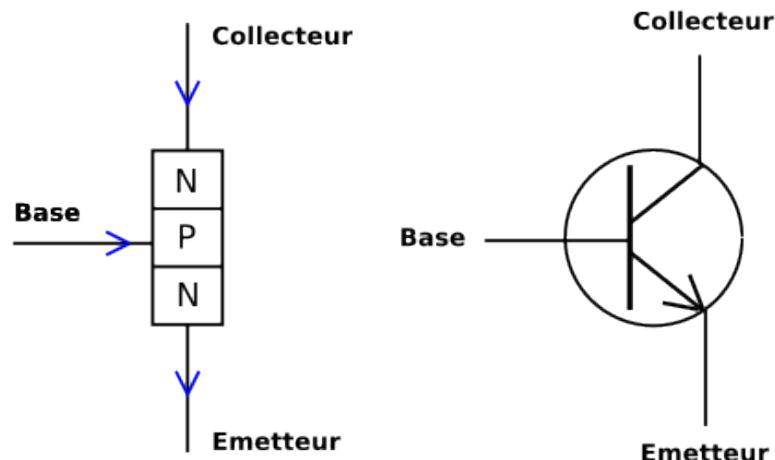
$$I_c = \beta \times I_b$$

Ce coefficient β (bêta) est appelé *facteur d'amplification du transistor*. Sa valeur est largement supérieure à 1, souvent quelques centaines. C'est une grandeur sans dimension, c'est-à-dire d'unité un.

Il faut noter que le courant de base doit toujours rester petit pour éviter la destruction du transistor. Une résistance sur la base est généralement utilisée pour limiter ce courant.

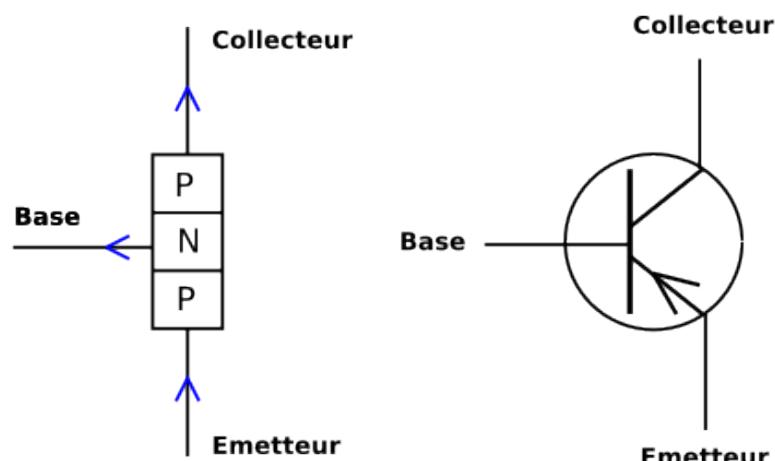
5.3 STRUCTURE D'UN TRANSISTOR

Pour mieux comprendre le fonctionnement du transistor, regardons sa structure interne. Un transistor est construit à partir d'un morceau de silicium de type N (dopé négativement) dans lequel sont diffusées des impuretés de type P. Il s'agit là d'un transistor de type NPN. La figure suivante montre le principe de la constitution d'un transistor NPN, ainsi que le symbole utilisé.



Transistor NPN, principe et symbole

Pour obtenir un transistor de type PNP, ce sont des impuretés de type N qui sont diffusées dans une lame de silicium de type P.



Transistor PNP, principe et symbole

5.4 LES JONCTIONS ET MODES DE FONCTIONNEMENT

On appelle *jonction* le contact entre une zone de silicium dopé N et une zone de silicium dopé P. Deux *jonctions* sont créées dans un transistor, une jonction base-émetteur J_{be} et une jonction base-collecteur J_{bc} .

Une jonction est dite *polarisée en direct* lorsque la tension entre zone P et la zone N est supérieure à 0.7 V (tension de seuil). Dans le cas contraire, on dit qu'elle est polarisée en inverse.

Le transistor a trois modes de fonctionnement intéressants :

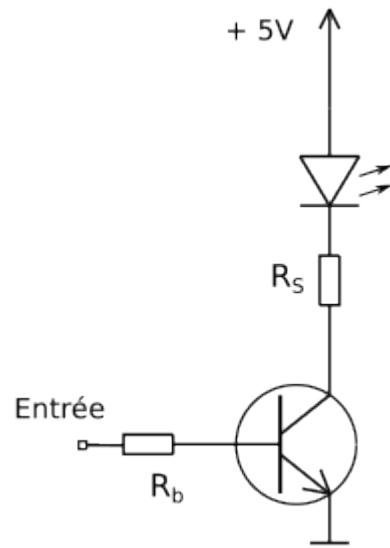
- le mode bloqué
- le mode amplificateur
- le mode saturé

Dans toutes les applications liées aux enseignes et afficheurs à LED, nous allons utiliser les transistors en mode de commutation, c'est-à-dire soit bloqué, soit saturé. Le transistor fonctionne alors comme un interrupteur, pour allumer ou éteindre des LED.

Le transistor est en mode bloqué lorsque la jonction base-émetteur n'est pas polarisée en direct. C'est le cas où la tension base-émetteur est inférieure à la tension de seuil. Aucun courant ne circule alors entre le collecteur et l'émetteur : $I_c = 0$.

Lorsque la tension base-émetteur dépasse la tension de seuil, le transistor va conduire. Étant donné que le facteur d'amplification β du transistor est généralement important, le courant du collecteur va rapidement n'être limité que par la charge se trouvant dans le circuit du collecteur. On dit alors que le transistor est saturé.

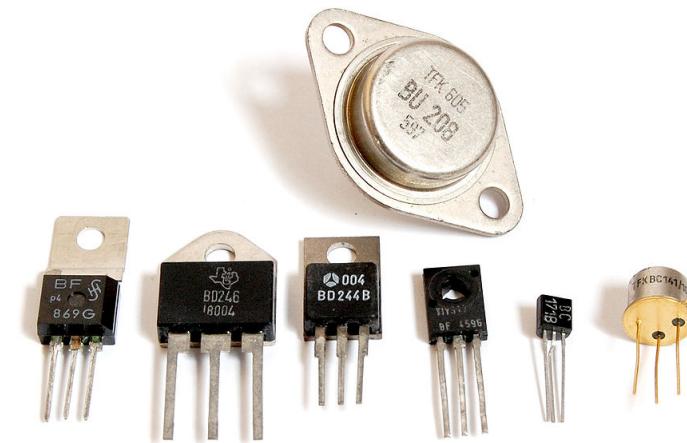
Voici un montage fréquemment utilisé pour les enseignes à LED. Un transistor est utilisé en commutation pour allumer ou éteindre une LED. La résistance connectée à la base limite le courant de base. Elle est calculée de telle manière que le courant produit par un état logique 1 soit suffisant pour saturer le transistor.



Commande d'une LED par un transistor

5.5 CARACTÉRISTIQUES DES TRANSISTORS

Il existe des milliers de modèles de transistors sur le marché ! Ils se présentent dans des boîtiers de tailles et de formes très différentes, dont voici quelques exemples :



Quelques boîtiers de transistors

Aujourd’hui, les boîtiers sont souvent prévus pour le montage en surface sur les circuits imprimés (SMD : *Surface Mounted Device* = Composants Montés en Surface) :



Transistor à monter en surface

Comment choisir un transistor adapté à une application particulière ? Les fabricants de semi-conducteurs indiquent un grand nombre de paramètres dans les fiches techniques (*data sheets* en anglais) décrivant leurs composants.

Voici les paramètres généralement les plus importants à prendre en compte :

5.5.0.1 Le courant maximum dans le collecteur I_{cmax}

La taille du transistor et la dimension de ses broches déterminent ce courant maximal, pouvant aller de quelques milliampères jusqu'à plusieurs dizaines d'ampères.

5.5.0.2 La tension maximale entre la base et le collecteur V_{bcmax}

Au-dessus d'une certaine valeur de la tension entre la base et le collecteur, la jonction base-collecteur risque de se détériorer. Cette valeur est souvent supérieure à 10 V et peut aller jusqu'à plusieurs centaines de volts.

5.5.0.3 La puissance maximale admissible par le transistor

Même lorsqu'il est saturé, la résistance entre le collecteur et l'émetteur n'est pas nulle et le courant qui traverse le transistor produit donc de la chaleur par effet Joule. Les transistors *de puissance* ont des boîtiers conçus spécialement pour dissiper cette chaleur. La puissance dissipée varie de quelques centaines de milliwatts jusqu'à des centaines de watts.

5.5.0.4 La fréquence maximale de fonctionnement

Le fabricant indique également la fréquence maximale de fonctionnement du transistor. Elle est généralement de l'ordre du mégahertz, voire davantage.

CHAPITRE 6

PROGRAMMATION EN C-ARDUINO

Pierre-Yves Rochat, EPFL
rév 2016/01/04

6.1 DIFFÉRENTES SIGNIFICATIONS DU MOT ARDUINO

L'**Arduino** a participé à rendre populaires les microcontrôleurs de manière extraordinaire. Qu'est-ce qui se cache derrière ce nom ?

Il faut différencier trois significations différentes du mot Arduino :

- une carte à microcontrôleurs
- un environnement de développement
- une librairie pour microcontrôleurs.

1. L'Arduino est une **carte à microcontrôleurs**, plus exactement une famille de cartes. L'**Arduino UNO** est la plus connue. Elle contient un microcontrôleur AVR du fabricant Atmel, le modèle ATmega328. Un câble USB permet de la brancher sur un PC, principalement pour déposer un programme dans le microcontrôleur. Les cartes Arduino sont *open hardware* : leurs plans sont publiques. Comme elles sont produites par de nombreux fabricants, leur prix est très favorable.



Exemple de carte Arduino

2. Le programme Arduino est aussi environnement de développement (IDE = Integrated Developpement Environnement). C'est donc un logiciel qui s'exécute sur un PC. Il fonctionne sur les principaux systèmes d'exploitation courant : Windows, Linux et MacOS. Il associe principalement un éditeur et un compilateur C. Il permet d'écrire un programme, de le compiler et de l'envoyer sur une carte Arduino. C'est un logiciel libre, écrit en Java, inspiré de l'environnement *Processing*.

On voit sur cette copie d'écran que l'interface est très simple :

```

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage level)
  delay(1000);                // wait for a second
  digitalWrite(led, LOW);       // turn the LED off by making the voltage LOW
  delay(1000);                // wait for a second
}

Compilation terminée.
Change-warnings --change-section-lma ..\eeprom\ /tmp/build8241316694376817726.eep
/usr/share/arduino/hardware/tools/avr/bin/avr-objcopy -O ihex -R .eeprom /tmp/
/tmp/build8241316694376817726.tmp/Blink.cpp.hex
Taille binaire du croquis : 1 056 octets (d'un max de 32 256 octets)

```

Logiciel Arduino

Plusieurs programmes similaires au programme Arduino existent pour supporter d'autres cartes à microcontrôleurs. C'est le cas du programme **Energia**, qui supporte les cartes Launchpad MSP430 de Texas Instrument. Nous utiliserons souvent cet environnement dans ce cours.

3. Finalement, on utilise souvent le mot Arduino pour désigner un langage de programmation. Il ne s'agit pas à proprement parlé d'un langage, mais plutôt d'un ensemble de procédures. Rappelons qu'une procédure est un ensemble d'instructions, écrites dans un langage de programmation. Ces procédures sont groupées dans une *librairie* (traduction abusive du mot anglais *library*). Ces procédures permettent de mettre en œuvre un microcontrôleur de manière très simple. Elles sont écrites en C, plus exactement en C++.

Ces procédures sont similaires au langage *Wiring*, qui a précédé l'Arduino. Ce terme serait plus correct, mais il est moins connu. Nous utiliserons dans ce cours l'expression **programmation en C-Arduino** pour désigner le fait de développer des programmes pour microcontrôleurs avec le langage C et les procédures Arduino. C'est le sujet de cette leçon. Plus exactement, nous allons décrire un

minimum de procédures qui vont nous permettre de programmer nos premières enseignes à LED.

6.2 CACHER LA COMPLEXITÉ DU MICROCONTRÔLEUR

Le but du C-Arduino est de cacher en partie la complexité du microcontrôleur. Accessoirement, c'est un moyen d'écrire des programmes qui peuvent, dans une certaine mesure, s'exécuter sur plusieurs modèles de microcontrôleurs.

Nous allons présenter ici :

- la structure générale d'un programme
- les entrées-sorties
- la gestion du temps.

6.3 LA STRUCTURE GÉNÉRALE D'UN PROGRAMME

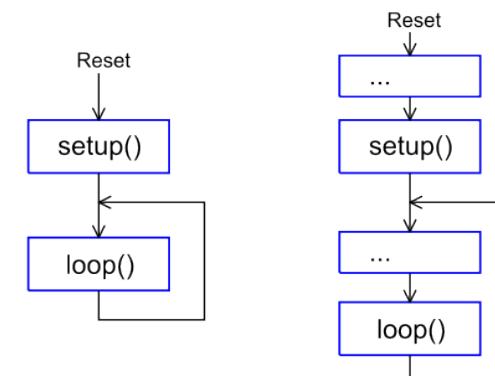
Plutôt que d'écrire un programme complet, avec sa procédure `main()`, Arduino nous propose d'écrire deux procédures : `setup()` et `loop()`.

La procédure `void setup()` s'exécute une seule fois au début de l'exécution du programme, qui correspond au *Reset* du microcontrôleur. Notons qu'un *Reset* se produit automatiquement lorsqu'on applique la tension d'alimentation sur un microcontrôleur. La procédure `setup()` s'exécute donc lorsqu'on allume notre dispositif. Elle contient généralement toutes les initialisations matérielles et logicielles nécessaires pour la suite du programme.

La procédure `void loop()` est appelée à la suite de `setup()`. Mais, contrairement au `setup()` qui n'est appelé qu'une fois, elle est appelée en permanence. Elle correspond donc à la boucle principale du programme, d'où son nom. Rapelons que les programmes pour microcontrôleur n'ont généralement pas de fin : l'exécution se poursuit jusqu'à ce que le microcontrôleur ne soit plus alimenté ou jusqu'à un *Reset*.

6.4 ORGANIGRAMME

Une des manière de représenter le déroulement d'un programme est l'organigramme. Voici donc l'organigramme général d'un programme Arduino :



Organigramme d'un programme Arduino

L'organigramme à gauche de la figure correspond à ce que voit l'utilisateur. Mais en fait, derrière l'usage de `setup()` et `loop()` proposé par Arduino, des instructions cachées s'exécutent avant chacune de ces procédures, comme le montre l'organigramme de droite. C'est dans la procédure `main()` produite par l'environnement Arduino que ces instructions sont ajoutées.

Il faut noter que la procédure `loop()` a une durée d'exécution qui peut varier considérablement d'un programme à un autre. Pour une enseigne ou un afficheur à LED, elle pourrait par exemple durer le temps d'un cycle complet de l'animation. Mais elle pourrait tout aussi bien durer un temps très court, par exemple un temps fixe de 10 µs. Tout dépendra de la manière de programmer.

6.4.1 Exemple

Le programme suivant est un programme Arduino correct :

```

1 void setup() {
2 }
3
4 void loop() {
5 }
```

Il est possible de le compiler et de l'exécuter... mais il ne fait rien !

6.5 LES ENTRÉES-SORTIES

L'usage des broches du microcontrôleur comme entrée ou comme sortie se fait par l'intermédiaire de registres spécialisés. Leurs noms et leurs rôle exacts varient d'un microcontrôleur à l'autre. Pour faciliter l'utilisation des broches comme entrées ou comme sorties, trois procédures sont proposées :

- `void pinMode(pin, mode)`
- `void digitalWrite(pin, value)`
- `value digitalRead(pin)`

6.5.1 pinMode()

La procédure `void pinMode(pin, mode)` est une procédure d'initialisation. Elle permet de placer une broche du microcontrôleur en entrée ou en sortie. Elle reçoit deux paramètres :

- `pin` : c'est le numéro logique de la broche. Attention, c'est un numéro qui a été arbitrairement choisi. Sur les cartes Arduino, c'est le numéro qui est noté sur la carte. Sur Energia, c'est le numéro de la broche sur le boîtier du microcontrôleur. Il s'agit d'un boîtier à 20 broches (DIL20).
- `mode` : la valeur INPUT place la broche en entrée, la valeur OUTPUT place la broche en sortie.

La procédure `pinMode()` ne rend rien à la fin de son exécution, d'où le mot `void` qui précède sa définition.

6.5.2 digitalWrite()

La procédure `void digitalWrite(pin, value)` permet d'agir sur une broche qui a été programmée en sortie. C'est une écriture. Elle permet de placer un 0 ou un 1 sur la sortie. Elle reçoit deux paramètres :

- `pin` : c'est le numéro logique de la broche.
- `value` : la valeur à donner à la sortie, 0 ou 1. Les symboles LOW (bas, 0) et HIGH (haut, 1) peuvent aussi être utilisés.

La procédure `digitalWrite()` ne rend rien à la fin de son exécution.

6.5.3 digitalRead()

La procédure `value digitalRead(pin)` permet de lire le niveau logique sur une broche qui a été programmée en entrée. La valeur rendue sera 0 ou 1 (LOW ou HIGH). Elle reçoit un seul paramètre :

- `pin` : le numéro logique de la broche.

La procédure `digitalWrite()` rend à la fin de son exécution la valeur lue. Ce sera un 0 ou un 1.

6.5.4 Exemple

Voici un programme qui utilise les instructions que nous venons de voir. Il semble correct :

```

1 void setup() {
2   pinMode(P1_0, OUTPUT);
3   pinMode(P1_3, INPUT);
4 }
5
6 void loop() {
7   digitalWrite(P1_0, (digitalRead(P1_3)));
8 }
```

En permanence, il écrit sur la broche P1_0, qui est la LED rouge du Launchpad, la valeur lue sur P1_3, qui est le bouton-poussoir. On devrait donc voir la LED rouge s'allumer lorsque le bouton-poussoir est pressé et s'éteindre lorsqu'il est relâché. Malheureusement... il ne fonctionne pas ! Il faut modifier la ligne d'initialisation de la manière suivante pour qu'il fonctionne un peu mieux :

```
3   pinMode(P1_3, INPUT_PULLUP);
```

C'est une raison électrique qui oblige l'utilisation du mode INPUT_PULLUP. Elle sera expliquée en détail dans une prochaine leçon. On apprendra aussi pourquoi ce programme fait l'inverse de ce qu'on avait imaginé : la LED sera allumée tant qu'on ne presse pas sur le bouton-poussoir et s'éteindra lorsqu'on le presse.

6.6 LA GESTION DU TEMPS

L'utilisation d'un microcontrôleur et d'un programme pour commander une enseigne à LED est motivée par l'envie de produire des animations visuelles. Ces animations sont des variations au cours du temps de l'état des LED. Il nous faut donc la possibilité de maîtriser le temps qui passe. Nous allons le faire avec la procédure `delay()`.

La procédure `void delay(ms)` permet d'attendre un temps donné. Ce temps est exprimé en ms (milliseconde). Elle ne rend rien à la fin de son exécution.

6.7 PROGRAMME BLINK

Prenons quelques-unes des procédures que nous venons de présenter pour écrire un programme tout simple, qui fait clignoter une LED (*blink* en anglais).

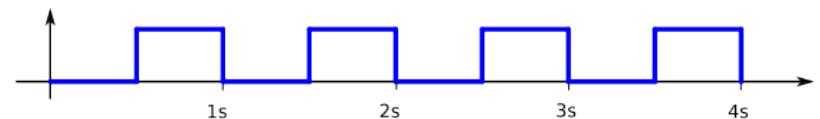
```

1 void setup() {
2   pinMode(P1_0, OUTPUT);
3 }
4
5 void loop() {
6   digitalWrite(P1_0, 1);
7   delay(500);
8   digitalWrite(P1_0, 0);
9   delay(500);
10 }
```

On y trouve la procédure `setup()` qui met en sortie la broche sur laquelle la LED rouge du LaunchPad est branchée.

La procédure `loop()` allume la LED, puis attend une demi-seconde, puis éteint la LED et attend à nouveau une demi-seconde. Tout le cycle dure une seconde. On a donc produit un *signal carré* de 1 Hz.

Voici un chronogramme qui montre l'évolution de la sortie en fonction du temps :



Chronogramme du programme Blink

Ce programme est un *classique* dans le monde des microcontrôleurs. Presque tous les projets commencent par lui ! En effet, on essaie presque toujours d'avoir au moins une LED dans un montage à microcontrôleurs. Au moment des premiers tests, ce programme va permettre de s'assurer que le microcontrôleur est bien fonctionnel et que l'environnement utilisé permet de le programmer.

6.8 UNE RICHE LIBRAIRIE

La librairie Arduino contient de nombreuses autres procédures. Elles sont bien documentées.

Elle est complétée par d'innombrables autres librairies, souvent développées pour l'usage d'un matériel particulier, comme les cartes-filles (*shields*) qu'on peut placer sur les cartes de base (Arduino, Launchpad, etc).

Le domaine des enseignes et afficheurs à LED n'est pas de reste à cet égard. De nombreux fournisseurs proposent du matériel et les librairies associées. Il semble facile de jouer à l'apprenti-sorcier en assemblant ce matériel et en mettant en œuvre les librairies proposées.

Mais il faut bien admettre qu'on se trouve souvent devant des problèmes insolubles lorsqu'on ne maîtrise pas ce qu'on fait... Le but de ce cours est de comprendre ce qui se passe, afin d'être capable de concevoir des enseignes et afficheurs à LED.

CHAPITRE 7

ENSEIGNES À MOTIFS FIXES

*Pierre-Yves Rochat, EPFL
rév 2015/09/15*

7.1 PLACER DES LED FIXES POUR FORMER UNE ENSEIGNE

Une des applications les plus simples des LED est la réalisation d'enseignes à motifs fixes. La disponibilité de LED rouges, vertes, bleues, oranges et blanches, pour ne citer que les plus courantes, permet de réaliser des enseignes multicolores.

Voici deux exemples très simples :

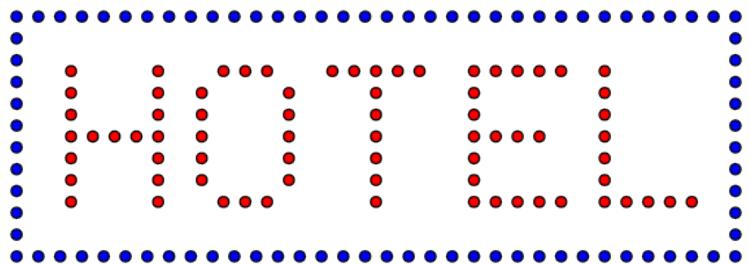


Figure : Exemple d'enseigne à motifs fixes

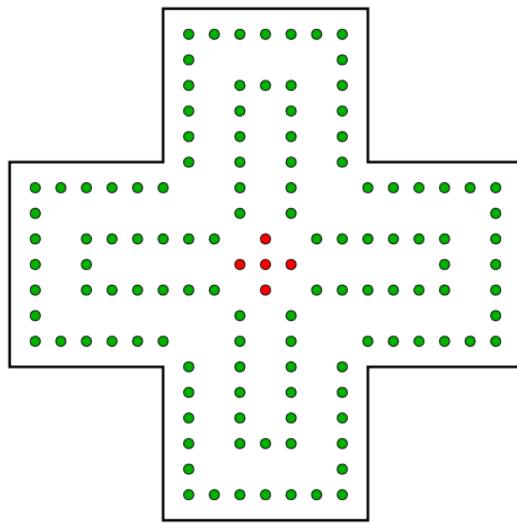


Figure : Autre exemple

La première enseigne affiche un mot lisible : HOTEL. Ce mot ne peut être changé, car les LED sont disposées de façon à former le mot HOTEL. La seconde enseigne affiche un motif géométrique qui fait penser à une croix de pharmacie.

On peut construire de telles enseignes en fixant des LED sur des panneaux par exemple en plexiglas. En allumant les LED, on produit un effet lumineux qui attire le regard et qui de plus est visible de nuit. En fait, plus l'intensité de la lumière ambiante est faible, mieux l'enseigne se verra.

7.2 DÉCOUPER LE MOTIF EN SEGMENTS

Mais on peut aller plus loin en découplant les motifs en différentes parties. Le terme “segment” sera souvent utilisé, bien que chaque partie n'a pas forcément la forme d'un segment de droite. Il est alors possible de commander chaque segment individuellement. On programadera alors des animations qui attirent l'œil, telles que des clignotements, des chenillards et même des variations continues d'intensité par PWM. Ce sujet sera abordé plus tard dans ce cours.

Voici comment l'enseigne de pharmacie peut être découpée en segments :

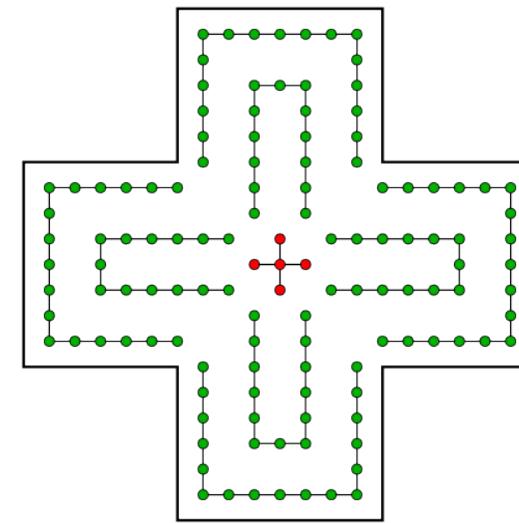


Figure : Découpage du motif en segments

Il y a neuf segments dans cet exemple, huit verts et un rouge.

7.3 SCHÉMA AVEC UN TRANSISTOR

Comment connecter ensemble un grand nombre de LED pour que les motifs formés puissent être commandés facilement, par exemple par un microcontrôleur ?

Le courant consommé par une LED standard est d'environ 10 mA, bien qu'il existe aussi des LED beaucoup plus puissantes, plutôt utilisées pour l'éclairage. Les sorties des circuits intégrés logiques et des microcontrôleurs permettent en général de fournir quelques dizaines de milliampères, soit un courant suffisant pour une ou deux LED. Pour davantage de LED, un transistor bipolaire ou MOS sera utilisé.

Voici le schéma utilisé, comportant un transistor NPN :

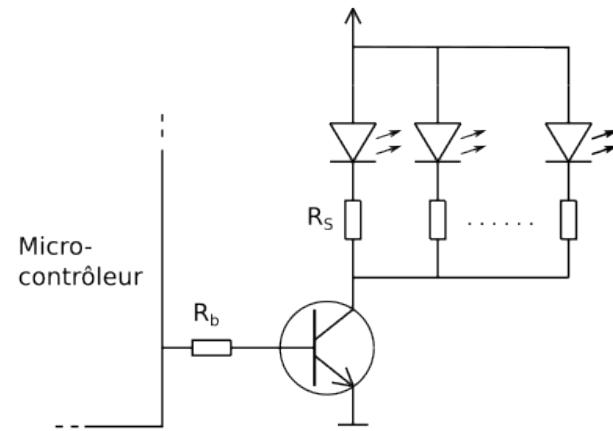


Figure : Transistor commandant plusieurs LED

Une sortie du microcontrôleur va commander la base du transistor à travers une résistance. Lorsque la sortie est à l'état 0, aucun courant ne peut circuler dans la base du transistor, qui est alors bloqué : aucun courant ne peut circuler entre le collecteur et l'émetteur, toutes les LED seront donc éteintes.

La valeur de la résistance de base sera déterminée de telle manière à être certain que le transistor soit saturé lorsque la sortie du microcontrôleur est à l'état logique 1. Il faut tenir compte de la tension de la sortie à l'état 1, souvent 3.3 V ou 5 V, selon la tension d'alimentation du microcontrôleur. Il faut lui soustraire la tension base-émetteur du transistor, qui est de 0.7 V. On s'assurera ensuite que le courant de base soit supérieur au courant de collecteur divisé par le facteur d'amplification du transistor.

Même si elles sont regroupées en un seul motif, commandé par une seule sortie d'un microcontrôleur, il est toujours nécessaire de placer une résistance de limitation de courant pour chaque LED. Il est fortement déconseillé de n'utiliser qu'une seule résistance en série avec plusieurs LED reliées en parallèle, car cette façon de faire ne permet pas d'ajuster le courant consommé par chacune des LED, donc la luminosité de l'afficheur ne sera pas constante. Ceci est dû au fait qu'il y a des dispersions importantes dans les caractéristiques des LED, même si elles sont du même fabricant et de la même série de production.

7.4 PLACER PLUSIEURS LED EN SÉRIE

Il est possible d'augmenter la tension d'alimentation et de placer plusieurs LED en série pour une seule résistance de limitation. Le courant est exactement le même dans chaque LED d'une branche !

Ce schéma montre comment commander plusieurs LED en série avec un transistor, avec une ou plusieurs branches :

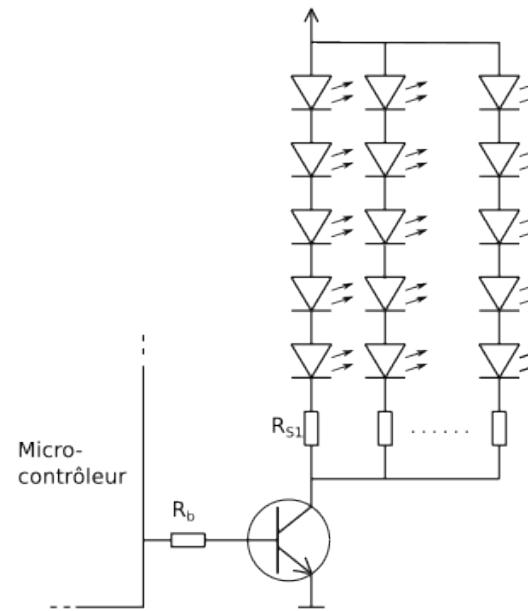


Figure : Utilisation de plusieurs LED en série

Il faut connaître la tension de chaque LED pour choisir le nombre optimal de LED dans chaque branche. Les LED rouges ont en général une tension d'environ 2 V, alors que les vertes ou les bleues ont plutôt 3 V. Mais il existe plusieurs technologies pour produire des LED et il faut bien se renseigner concernant les LED choisies... ou simplement les mesurer !

Avec une tension d'alimentation de 12 V, on peut alimenter par exemple 5 LED rouges ou 3 LED vertes. Notez qu'avec un seul transistor faible signal, il est possible de commander un grand nombre de LED. Par exemple, un BC337, dont le courant maximal est d'environ 500 mA (cette valeur varie selon les fabricants)

permet de commander jusqu'à environ 250 LED rouges avec cette technique : chaque groupe de 5 LED en série reçoit 10 mA et il est possible de placer 50 branches.

Prenons un autre exemple. Avec une alimentation de PC portable, fournissant 16 V et un maximum de 4.5 A, combien de LED vertes est-il possible de commander ? La tension aux bornes de chaque LED doit être d'environ 3 V, pour un courant de 10 mA.

Voici la réponse : on peut mettre 5 LED en série (15 V) avec une résistance de limitation. Il est possible de placer 450 groupes de LED ($450 \times 10 \text{ mA} = 4.5 \text{ A}$). C'est donc un total de 2'250 LED qu'il est possible de commander !

Dans ce cas, voici comment calculer la valeur de la résistance série : les 5 LED présentent une différence de potentiel de 3 V à leurs bornes. Il reste $16 - (5 \times 3) = 1 \text{ V}$ aux bornes de la résistance. Pour un courant nominal, il faut $R = U / I = 1 \text{ V} / 10 \text{ mA} = 100 \Omega$.

En pratique, on réalisera un montage de test avec les 5 LED et une résistance, sans oublier le transistor, qui a aussi une chute de tension. On mesurera la tension aux bornes de la résistance, on calculera le courant qui la traverse ($I = U / R$). On corrigera ensuite la valeur de la résistance pour se rapprocher du courant souhaité (par exemple 10 mA) et on reprendra le test.

Il est même possible de commander un segment contenant des LED de plusieurs couleurs. Voici un schéma correspondant :

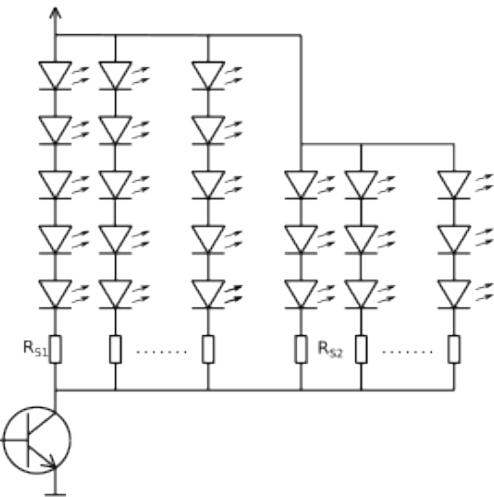


Figure : Utilisation simultanée de plusieurs types de LED

7.5 COURANTS IMPORTANTS

Ce type de montage peut nécessiter des courants d'alimentation importants. Lorsque des milliers de LED sont utilisées pour un seul segment, il faudra utiliser un transistor correctement dimensionné.

Il faudra aussi réaliser un câblage électrique adapté : la section de cuivre du fil ou du câble utilisé devra être choisie correctement. Si le fil est long, la chute de tension peut devenir non négligeable, et donc influencer le courant dans les LED et par conséquent leur luminosité.

Comme il y a en général un nombre important de segments, ce sont surtout les courants dans les conducteurs d'alimentation qui peuvent devenir importants. C'est le cas du câblage qui apporte la tension positive sur les anodes des transistors. Mais c'est tout autant le cas dans les câblages de la masse.

La figure suivante montre l'importance des courants :

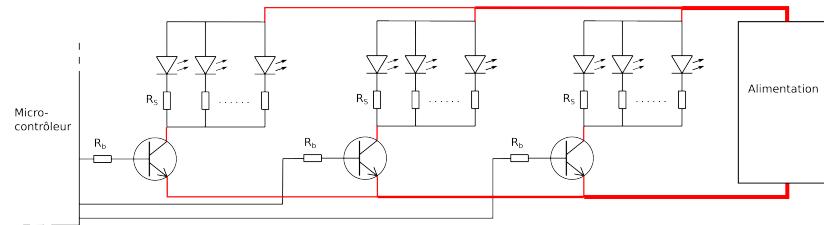


Figure : Courants dans les fils des segments

7.6 PROGRAMMATION D'ANIMATIONS

Le premier programme souvent proposé pour apprendre à mettre en œuvre un microcontrôleur, le fameux *blink* est déjà un programme d'animation d'une enseigne, avec une seule sortie ! Fondamentalement, il est facile d'écrire une animation. Il faut écrire les instructions pour mettre les sorties dans un certain état, puis attendre un certain temps. L'animation va comporter un certain nombre de ces paires d'instructions et tournera en boucle. Prenons un exemple très simple, qui produit une animation du type *chenillard* :

```

1 void setup() {
2   P2DIR |= 0xFF; // P2.0 à P2.7 en sortie
3 }
4
5 void loop() { // Boucle infinie, correspond à toute l'animation
6   P2OUT = 0;           // éteint toutes les LED
7   P2OUT |= (1<<0); delay(200); // allume la première LED
8   P2OUT |= (1<<1); delay(200); // allume successivement chaque LED...
9   P2OUT |= (1<<2); delay(200);
10  P2OUT |= (1<<3); delay(200);
11  P2OUT |= (1<<4); delay(200);
12  P2OUT |= (1<<5); delay(200);
13  P2OUT |= (1<<6); delay(200);
14  P2OUT |= (1<<7); delay(1000); // attend un peu après la dernière LED
15  P2OUT = 0;           delay(500); // éteint toutes les LED pendant 1/2 seconde
16  P2OUT = 0xFF;        delay(500); // allume toutes les LED
17  P2OUT = 0;           delay(500); // répète...
18  P2OUT = 0xFF;        delay(1000);
19  P2OUT = 0;           delay(500); // pause avant la reprise de l'animation
20 }
```

On aurait pu utiliser uniquement des procédures `pinMode()` pour l'initialisation et `digitalWrite()` pour les animations. Dans un cas comme dans l'autre, le style de ce programme n'est pas un modèle du genre et a donc nécessité de nombreux commentaires. Or *un programme bien écrit ne nécessite pas beaucoup de commentaires*, alors qu'on voudrait souvent vous faire croire que plus un programme comporte de commentaires, mieux il est écrit !

L'utilisation de définitions comme :

```
#define Led1ON P2OUT |= (1<<0)
#define Led1OFF P2OUT &=~(1<<0)
```

rendrait le code plus lisible.

Mais le fait d'utiliser l'accès aux sorties par le registre `P2OUT` permet de programmer plus simplement la fonction du chenillard :

```

1 void ChenillardAjoute(uint16_t attente) {
2   uint16_t i;
3   for (i=0; i<8; i++) {
4     P2OUT |=(1<<i);
5     delay(attente);
6   }
7 }
8 ...
9 void loop() { // Boucle infinie, correspond à toute l'animation
10  ChenillardAjoute(200);
11  delay(800); // attend un peu après la dernière LED
12  P2OUT = 0; delay(500); // éteint toutes les LED pendant 1/2 seconde
13 ...
14 }
```

On pourrait aussi écrire une fonction chenillard encore plus générale, avec en paramètres non seulement le temps d'attente, mais le sens du mouvement et le fait de garder ou non les LED précédentes allumées.

Que vois-je ? Ai-je écrit le mot *mouvement* ? Est-ce que les LED bougent ? C'est toute la magie des LED... Elles ne bougent pas, mais peuvent donner l'impression de mouvement.

PWM : MODULATION DE LARGEUR D'IMPULSION

*Pierre-Yves Rochat, EPFL
rév 2015/07/19*

8.1 VARIER L'INTENSITÉ D'UNE LED

Beaucoup d'enseignes à LED se contentent d'allumer et d'éteindre des LED ou des groupes de LED. On pourrait obtenir des effets beaucoup plus intéressants et variés en ayant la possibilité de changer leurs intensités lumineuses.

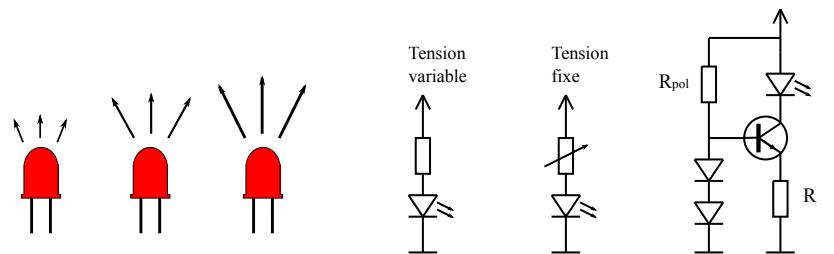


Figure : Varier l'intensité d'une LED

Comment faire varier l'intensité d'une LED ? On sait que c'est le courant qui la traverse qui produit de la lumière. Faire varier ce courant n'est généralement pas facile. On peut faire varier la tension qui alimente la LED, ou encore la résistance qui limite son courant. Il est aussi possible de piloter le courant par un transistor, avec un montage adéquat.

Mais il existe une solution très différente et généralement beaucoup plus simple à implémenter. Souvenons-nous du clignotement d'une LED. Que se passe-t-il lorsqu'on augmente la fréquence du clignotement ? À partir d'une cer-

taine fréquence, notre œil commence à ne plus voir qu'un scintillement. En augmentant encore la fréquence, il voit simplement la LED allumée, mais avec une intensité plus faible que si elle était allumée en permanence.

8.2 LE PWM

À l'idée de faire clignoter rapidement la LED, ajoutons l'idée de faire varier le temps pendant lequel elle est allumée. On obtient alors la Modulation de Largeur d'Impulsion (MLI) ou Pulse Width Modulation (PWM) en anglais. Regardons la figure suivante :

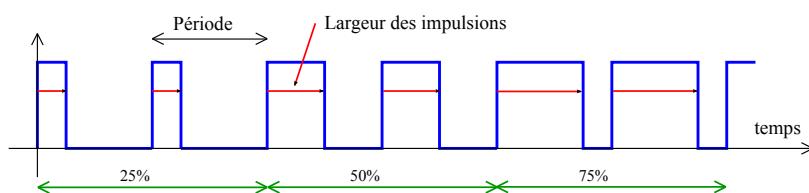


Figure : signal PWM

Le signal a une période constante (donc une fréquence constante). La durée de la partie active du signal varie. Dans la première partie du tracé, 25% de la puissance disponible peut être envoyée à la charge, vu que le signal est à 1 durant 25% du temps alors qu'il est à 0 le reste du temps. De même, la puissance passe à 50% au milieu du tracé et à 75% dans la dernière partie. Il est aussi possible de n'envoyer aucune puissance (0%) en laissant la sortie à 0, ou toute la puissance (100%) en laissant la sortie à 1.

Le rapport entre la durée de la partie active du signal et la durée de la période s'appelle le **rappor^t cyclique** (en anglais *duty cycle*).

8.3 FRÉQUENCE DU PWM

L'usage du PWM est très répandu, avec des applications dans beaucoup de domaines. La commande de moteurs s'effectue très souvent par PWM. Dans ce cas, c'est la nature inductive du moteur qui effectue l'intégration du signal, ainsi que l'inertie mécanique du système.

Dans le cas de la commande de diodes lumineuses, le temps d'allumage et d'extinction est très court. La LED continue à "clignoter" jusqu'à plusieurs dizaines de mégahertz. C'est l'œil humain qui ne voit pas le clignotement. L'œil a

une fréquence de perception maximale située vers 75 Hz. C'est bien lui qui effectue l'intégration du signal pour en percevoir une valeur moyenne.

Dans divers domaines de l'électronique, les fréquences des signaux PWM peuvent aller couramment jusqu'à des dizaines de mégahertz. Mais plus la fréquence est élevée, plus les pertes électriques à l'instant des changements de valeurs sont importantes et peuvent conduire à une dissipation importante d'énergie dans les éléments de commutation.

L'œil a une fréquence limite de perception du clignotement. Par exemple, on sait qu'un tube fluorescent (Néon) clignote à 100 Hz, vu qu'il est commandé par du courant alternatif à 50 Hz, et que chaque période a une alternance positive et une alternance négative. Les cellules sensibles de notre œil (les cônes pour la vision en couleur et les batonnets pour la vision périphérique et à faible intensité) n'ont pas la même limite de perception du clignotement.

Pour les enseignes et afficheurs à LED, on vise généralement des fréquences de l'ordre de 100 à 200 Hz. Il est déjà souvent extrêmement difficile d'envoyer à ces fréquences toutes les informations à l'ensemble des LED, qui peuvent être très nombreuses. Par exemple, les murs de LED capables d'afficher de la vidéo nécessitent des circuits électroniques complexes, qui seront étudiés plus tard dans ce cours.

8.4 PROGRAMMATION D'UN PWM

Voici un programme très simple qui génère un signal PWM sur une sortie d'un microcontrôleur :

```

1 #define LedOn digitalWrite(P1_0, 1)
2 #define LedOff digitalWrite(P1_0, 0)
3
4 uint16_t pwmLed; // valeur du PWM, 0 à 100
5
6 void setup() { // Initialisations
7   pinMode(P1_0, OUTPUT); // LED en sortie
8   pwmLed = 25; // valeur du PWM.
9 }
10
11 void loop() { // Boucle infinie, durée 10ms => un cycle du PWM à 100 Hz
12   LedOn;
13   delayMicrosecond(100*pwmLed); // durée de l'impulsion
14   LedOff;

```

```

15 delayMicrosecond(100*(100-pwmLed)); // solde de la période
16 }

```

Le programme a été écrit de telle manière que les valeurs du PWM doivent être choisies entre 0 et 100, elles représentent donc dans ce cas des *pour cents*. Les informaticiens choisissent plus souvent des valeurs dans une plage binaire, comme par exemple de 0 à 255 (8 bits, voir l'exemple suivant).

La fréquence a été choisie ici à 100 Hz. En effet les deux délais de la boucle principale totalisent $100 \times 100 \mu\text{s}$, donc 10 ms.

Après les initialisations de la sortie et de la variable qui contient en permanence la valeur du PWM, la boucle infinie alterne la partie active et la partie inactive du signal. Les attentes sont obtenues par des délais exprimés en us.

Ce programme donne l'impression que la diode lumineuse est à demi intensité, malgré une commande à 25 %. C'est lié au fait que la réponse de l'œil n'est pas linéaire, mais logarithmique. On remarque aussi que le PWM est visible en déplaçant rapidement la diode devant l'œil.

8.5 GÉNÉRATION DE PLUSIEURS SIGNAUX PWM

Le principe du programme que nous venons de voir ne convient pas à la programmation de PWM sur plusieurs sorties. Voici une manière de programmer un PWM qui se prête à gérer plusieurs sorties :

```

1 uint8_t pwmLed; // valeur du PWM, 0 à 255 (8 bits)
2 uint8_t cptPwm; // compteur du PWM
3
4 void setup() { // Initialisations
5   pinMode(P1_0, OUTPUT); // LED en sortie
6   pwmLed = 64; // valeur du PWM. Elle est ici fixe, mais pourrait changer
7   // à tout moment en complétant le programme.
8   cptPwm = 0; // compteur du PWM
9 }
10
11 void loop() { // Boucle infinie, durée 39us (256 * 39us = ~10ms)
12   if ((cptPwm==0) && (pwmLed>0)) LedOn; // pour une valeur positive
13   if (cptPwm==pwmLed) LedOff;
14
15   cptPwm++; // passe automatiquement de 255 à 0 (overflow)

```

```

16   delayMicroseconds(39);
17 }

```

Dans ce cas, la boucle principale dure seulement le temps qui correspond à une fraction de la période du PWM, ici un 256^e de la période. L'usage de cette valeur, associée à un compteur de type `uint8_t` (8 bits non signés), simplifie le programme, en évitant de gérer le retour à zéro, qui s'effectue au moment du dépassement de capacité (overflow).

Voici comment modifier ce programme pour qu'il commande 8 LED, avec 8 valeurs différentes de PWM :

```

1 uint8_t pwmLed[8]; // valeurs des PWM, 0 à 255 (8 bits), pour 8 LED
2 uint8_t cptPwm; // compteur du PWM
3
4 void setup() ...
5
6 void LedOn (uint8_t n) ... procédure qui allume une des 8 LED
7 void LedOff (uint8_t n) ... procédure qui éteint une des 8 LED
8
9 void loop() { // Boucle infinie
10   for (uint8_t i=0; i<8; i++) {
11     if ((cptPwm==0) && (pwmLed[i]>0)) LedOn(i); // allume la LED concernée
12     if (cptPwm==pwmLed[i]) LedOff(i); // éteint la LED correspondante
13   }
14
15   cptPwm++; // passe automatiquement de 255 à 0 (overflow)
16   delayMicroseconds(39); // on pourrait diminuer cette valeur
17   // pour tenir compte du temps d'exécution de la boucle for.
18 }

```

8.6 PROGRAMMATION D'UNE SÉQUENCE EN PWM

Dans des applications comme la gestion des moteurs d'un robot mobile, les valeurs du PWM sont calculées à chaque cycle, en fonction des valeurs des capteurs (capteurs de distance, caméras, etc.)

Dans le cas des enseignes et afficheurs à LED, les valeurs sont calculées en fonction du temps qui passe, pour former des séquences. Comment connaître le

temps qui s'écoule ? Une des manières consiste simplement à compter les cycles du PWM, qui ont une durée constante.

Prenons un exemple. Pour donner l'impression qu'un appareil est en mode "repos", une LED va avoir son intensité qui varie de la manière suivante :

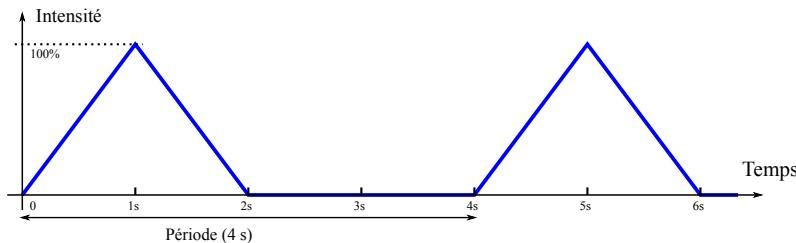


Figure : diagramme des temps de la LED suggérant le repos

La séquence est cyclique et dure 4 s. Il imite la respiration d'une personne qui dort. Durant la première seconde, l'intensité augmente jusqu'au maximum. Durant la deuxième seconde, elle diminue jusqu'à zéro, valeur qui reste jusqu'à la fin du cycle.

La courbe se divise en 3 parties. Il est facile d'exprimer l'intensité en fonction du temps dans chacune des parties. Voici comment compléter le programme :

```

1 uint16_t pwmLed; // valeur du PWM, 0 à 255, 16 bits pour les calculs
2 uint16_t cpt10ms = 0; // compteur des cycles de 0 à 400, par 10ms, total 4s
3 ...
4 void loop() { // Boucle infinie, durée 39us (256 * 39us = ~10ms)
5   if (cptPwm==0) {
6     cpt10ms++;
7     if (cpt10ms<100) { //première seconde
8       pwmLed = cpt10ms * 256 / 100; // droite montante
9     } else if (cpt10ms<200) { // deuxième seconde
10      pwmLed = 256 - ((cpt10ms-100) * 256 / 100); // droite descendante
11    } else {
12      pwmLed = 0;
13    }
14    if ( cpt10ms==400) cpt10ms = 0; // fin des 4 s
15  }
16
17  if ((cptPwm==0) && (pwmLed>0)) LedOn; // seulement si valeur non nulle
18  if (cptPwm==pwmLed) LedOff;

```

```

19
20   cptPwm++; // passe automatiquement de 255 à 0 (overflow)
21   delayMicroseconds(39);
22 }

```

Il est important de contrôler que les variables utilisées ne produisent pas de dépassement de capacité pour les calculs effectués. La variable `pwmLed` a été choisie ici sur 16 bits. Pour la première droite, `cpt10ms` peut aller jusqu'à 99. Multiplié par 256, on obtient un nombre inférieur à 65'535 (qui est le nombre maximum que peut représenter un nombre entier de 16 bits). Le contrôle doit être fait pour tous les cas.

8.7 CONVERTISSEUR DAC SIMPLE

Bien que le signal PWM soit un signal binaire (il est soit à 0 soit à 1 à un instant donné), il est porteur d'une valeur analogique. Il suffit en effet de faire l'intégrale du signal pour s'en rendre compte. Il faut pondérer le 1 logique comme la valeur +1 et le 0 logique comme la valeur -1, comme le montre la figure suivante :

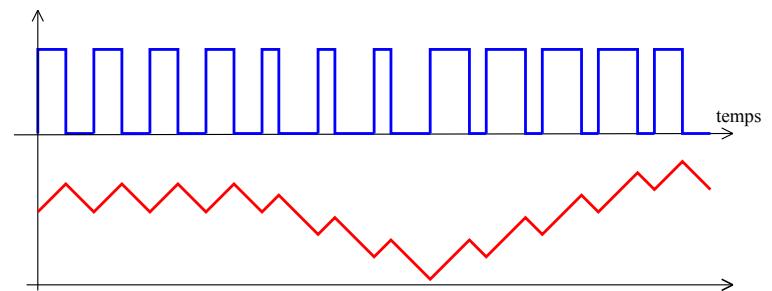


Figure : signal PWM intégré

Il est donc possible de réaliser un convertisseur numérique-analogique (Digital to Analog Converter, DAC) basé sur un signal PWM. Il doit être suivi par un filtre passe-bas adapté à la fréquence du PWM. Ce filtre peut être un simple filtre R-C.

8.8 CIRCUITS SPÉCIALISÉS

Comment générer un signal en PWM avec des circuits logiques ? Il faut utiliser un compteur, piloté par une horloge de fréquence fixe. Un détecteur de 0 permet de signaler le début du cycle. Un comparateur, connecté à un registre qui contient la valeur du PWM, permet de détecter la fin de l'impulsion. Une bascule *Set-Reset* génère le signal.

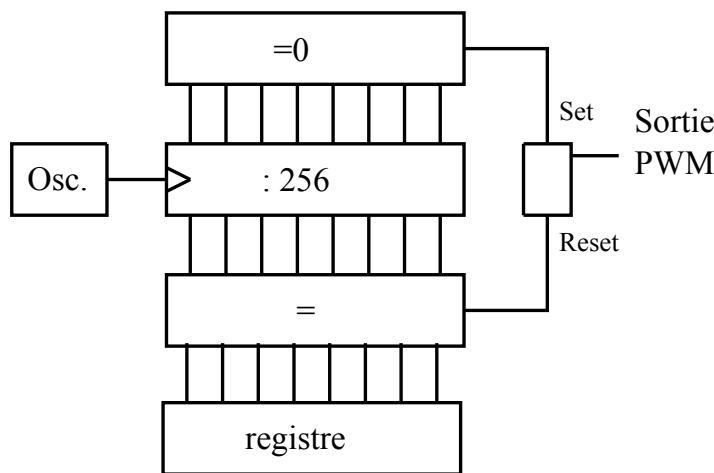


Figure : génération d'un PWM avec un système logique

Les microcontrôleurs contiennent des circuits logiques supplémentaires ressemblant beaucoup à ce schéma. On les appelle des *Timers*. Complétés de circuits logiques dédiés, ils peuvent générer du PWM sans autre programmation que l'initialisation des registres de contrôle qui leur sont associés. Nous en parlerons plus tard dans ce cours.

CHAPITRE 9

LE MULTIPLEXAGE TEMPOREL

Pierre-Yves Rochat, EPFL
rév 2015/07/16

9.1 COMMANDER DEUX LED INDÉPENDANTES AVEC UNE BROCHE

Nous allons nous poser une question un peu curieuse. Est-ce possible de commander deux LED avec une seule broche d'entrée-sortie d'un microcontrôleur ? Bien entendu, il ne s'agit pas ici que les deux LED s'allument et s'éteignent en même temps. On a déjà vu comment commander même des milliers de LED avec une seule broche. On cherche bien ici à les commander indépendamment l'une de l'autre.

En considérant qu'une sortie peut prendre l'état 0 et l'état 1, on voit mal comment le faire... Mais souvenons-nous qu'une broche d'un microcontrôleur peut aussi être une entrée. Or une entrée, c'est une broche qui n'impose ni un 0, ni un 1. On parle d'une broche à *haute impédance*. On utilise même parfois l'expression *logique à 3 états* (tri-state): l'état 0, l'état 1 et l'état H (haute impédance).

Regardons ce schéma, où une LED est reliée entre la broche et la *Masse* (avec sa résistance de protection en série) et une autre LED est reliée entre la broche et le *Plus* (Vcc 3.3V).

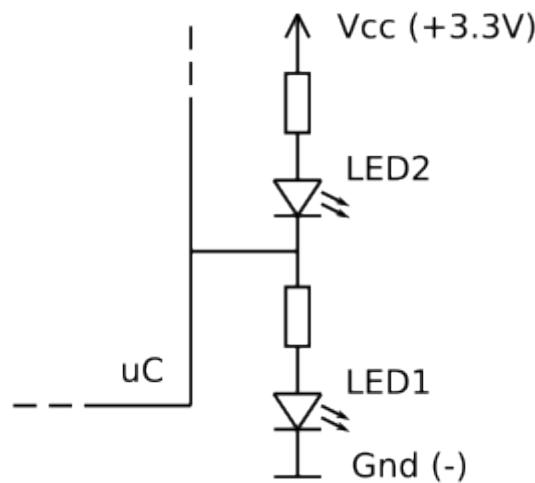


Figure : deux LED sur une broche

Il est possible avec ce montage :

- d'allumer la LED1 (sortie à 1)
- d'allumer la LED2 (sortie à 0)
- de n'allumer aucune des deux LED (sortie à haute impédance).

Mais comment allumer les deux LED en même temps ? C'est évidemment impossible !

Notez qu'une contrainte électronique existe avec ce montage. La tension Vcc doit être inférieure à deux fois la tension de seuil des LED utilisées. Il ne fonctionne pas à 5V, les LED s'allumerait alors en permanence, la tension d'alimentation étant suffisante pour dépasser les seuils des deux LED. Ce montage fonctionne bien à 3.3V, avec des LED vertes ou bleues. La tension de seuil des LED rouges est généralement trop faible.

9.2 MULTIPLEXAGE TEMPOREL

En allumant successivement la LED1 puis la LED2 à une fréquence supérieure à la fréquence maximale de sensibilité de l'œil, on va donner l'impression que les deux LED sont allumées en même temps.

Examinons ce programme :

```

1 #define SORTIE_0 pinMode(P1_4, OUTPUT); digitalWrite(P1_4, 0)
2 #define SORTIE_1 pinMode(P1_4, OUTPUT); digitalWrite(P1_4, 1)
3 #define SORTIE_HI_Z pinMode(P1_4, INPUT)
4
5 void loop() {
6     uint8_t i;
7     SORTIE_HI_Z; delay (1000); // LED1 et LED2 éteintes pendant 1 s
8     SORTIE_1; delay (1000); // LED1 allumée pendant 1 s
9     SORTIE_0; delay (1000); // LED2 allumée pendant 1 s
10
11 for (i=0; i<250; i++) { // 250 x 4 ms = 1 s
12     SORTIE_1; delay (2); // LED1 allumée
13     SORTIE_0; delay (2); // LED2 allumée
14 }
15 }

```

On voit que tous les états possibles pour 2 LED sont affichés (00, 01, 10, 11). Il a toutefois un défaut : lorsqu'elles sont allumées ensemble, chaque LED donne l'impression d'être allumée plus faiblement. En effet, on a généré un PWM de 50%. Ce défaut peut être corrigé en ne dépassant jamais les 50% pour chaque LED :

```

6 ...
7 SORTIE_HI_Z; delay (1000); // LED1 et LED2 éteintes pendant 1 s
8 for (i=0; i<250; i++) { // 250 x 4ms = 1 s
9     SORTIE_1; delay (2); // LED1 allumée
10    SORTIE_HI_Z; delay (2); // LED1 et LED2 éteintes
11 }
12 ...

```

L'affaiblissement de l'intensité peut être partiellement compensé dans ce cas. En effet, le courant nominal d'une LED (donné par le fabricant) ne doit pas être dépassé en continu. Mais lorsque l'allumage n'est jamais continu, c'est le courant maximal de la LED (également donné par le fabricant), qui ne doit pas être dépassé. Or le courant maximal (typiquement 20mA) est supérieur au courant nominal (typiquement 10mA). On va donc pouvoir utiliser une résistance de limitation plus petite pour atteindre ce courant maximal, pour autant que la contrainte de ne jamais allumer la LED plus de 50% du temps soit respectée par le programme.

Cette technique a comme principal désavantage la complexité de sa programmation. L'utilisation d'un Timer et d'une interruption va la simplifier. Le sujet sera abordé plus tard dans ce cours.

9.3 CHARLIEPLEXING

Une technique similaire peut être utilisée avec un nombre quelconque de LED en plaçant deux LED (une dans chaque sens) entre chaque broche utilisée. On parle alors de *Charlieplexing*. Voici le schéma pour 3 broches, avec 6 LED :

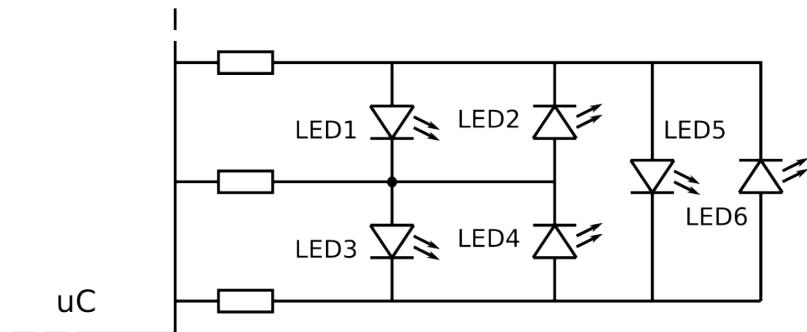


Figure : Charlieplexing avec 3 broches

On voit que 3 broches permettent de commander 6 LED. Une résistance de limitation est placée en série avec chaque broche. Sa valeur doit tenir compte du fait que le courant passant dans une LED passe par deux résistances. Pour allumer une LED, il faut :

- mettre à 1 la broche qui commande son anode
- mettre à 0 la broche qui commande sa cathode
- mettre en entrée (sortie à haute impédance) les broches restantes (une seule dans ce cas).

Ce montage est parfait lorsqu'il y a besoin d'allumer une seule LED parmi plusieurs. Si toutes les LED doivent pouvoir être allumées *en même temps*, il faut les allumer successivement à une fréquence suffisante, comme le montre la figure suivante :

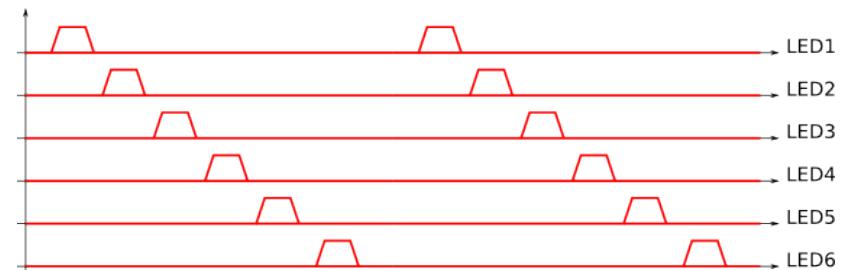


Figure : Chronogramme du Charlieplexing

C'est le principe même du **multiplexage temporel** !

Remarquez la manière de dessiner. A chaque cycle, chaque LED peut être individuellement allumée ou non. La luminosité va diminuer par rapport à l'intensité maximale d'une LED. Dans ce cas, ce sera d'un facteur 6, partiellement compensable par l'usage du courant maximal des LED.

9.4 USAGE DU MULTIPLEXAGE TEMPOREL

Si cette technique permettant de commander plusieurs LED avec un petit nombre de broches vous permet de *sauver* un projet où une broche manque sur votre microcontrôleur, tant mieux. Les amateurs de prouesses réalisées avec de petits microcontrôleurs à 8 ou même à 6 broches vont se régaler ! Ils vont, par exemple, commander tous les feux tricolores d'un carrefour sur une maquette miniature avec un microcontrôleur à 8 broches. Les familles PIC (Microchip) et les AVR (Atmel) offrent de nombreux microcontrôleurs dans cette gamme. Par exemple, l'ATtiny85 a son fan-club sur Internet.

Beaucoup plus sérieusement, la commande des afficheurs matriciels peut être simplifiée de manière très significative par le multiplexage temporel. Ce sujet sera étudié en détail, tant au niveau de l'architecture matérielle qu'au niveau de la commande logicielle.

9.5 AFFICHEURS 7 SEGMENTS MULTIPLEXÉS

Voici encore un usage très pratique du multiplexage temporel : la commande d'afficheurs 7 segments à plusieurs digits. Même sous forme de modules comportant un seul digit, les fabricants choisissent généralement, pour limiter le nombre de broches, de regrouper les anodes (montage à *anode commune*) ou les cathodes (montage à *cathode commune*) :

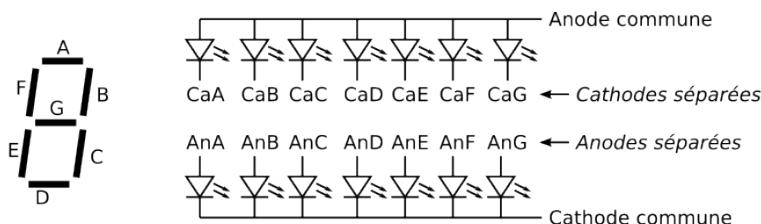


Figure : Afficheurs 7 segments

Sur les modules qui contiennent plusieurs digits, en plus des anodes (ou cathodes) regroupées par digit, les cathodes (ou les anodes) sont regroupées au niveau de chaque segment pour tous les digits.

Le multiplexage temporel convient très bien pour commander plusieurs digits, sans multiplier le nombre de broches utilisées sur le microcontrôleur. Chaque digit a son anode (commune à tous les segments), chaque segment a sa cathode (commune à tous les digits).

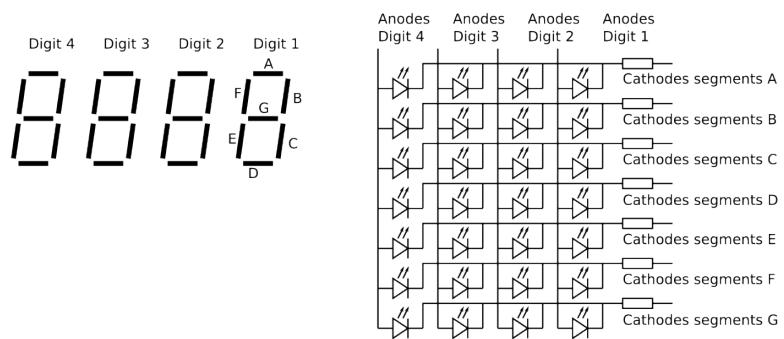


Figure : Afficheurs 7 segments à 4 digits

Les résistances de limitation doivent impérativement être placées du côté des segments. En effet, le multiplexage temporel va consister à allumer les segments concernés du premier digit, puis du deuxième, etc. À chaque instant, le courant concernant le digit sélectionné peut correspondre à un nombre compris entre 0 et 7. Par contre, du côté des segments, un seul peut être allumé à un instant donné.

En plaçant une résistance du côté du digit, son courant varierait en fonction du nombre de segments concernés. La chute de tension aux bornes de la résistance serait donc variable, entraînant une luminosité changeante selon le motif affiché (le chiffre 1 serait plus lumineux que le chiffre 8). Par contre, en plaçant une résistance du côté des segments, un seul segment peut être affiché à la fois. Le courant sera donc constant (ou nul si le segment est éteint).

Le programme de commande doit produire des signaux comme le montre la figure suivante :

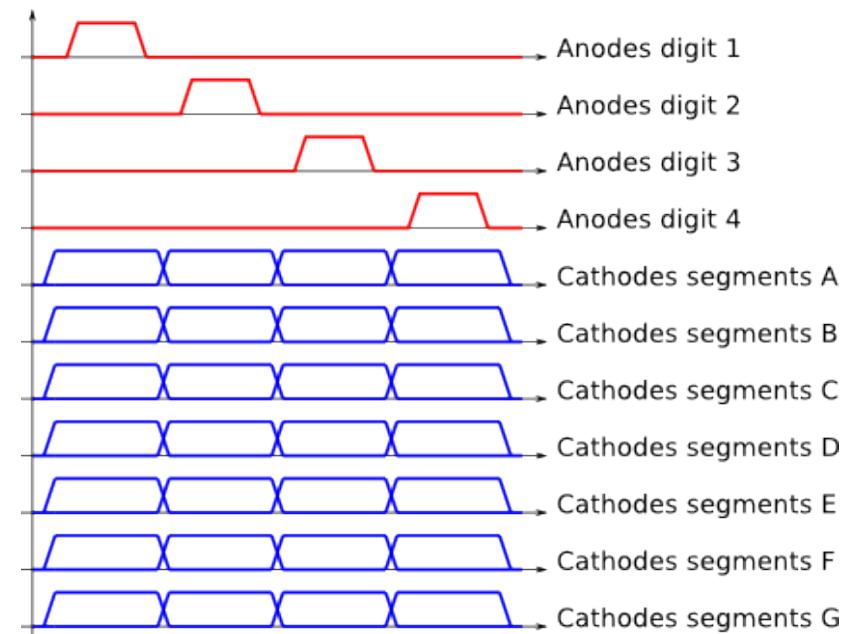


Figure : Diagramme des temps de la commande d'un afficheur 7 segments à 4 digits

Remarques :

- Les anodes sont actives à l'état 1.
- Les cathodes sont actives à l'état 0.
- Le courant dans les cathodes est limité au courant d'une LED. Une broche d'un microcontrôleur peut les commander directement.
- Le courant dans les anodes peut atteindre le courant de sept LED à un instant donné. Il faut donc un élément d'amplification. On utilisera généralement un transistor PNP.

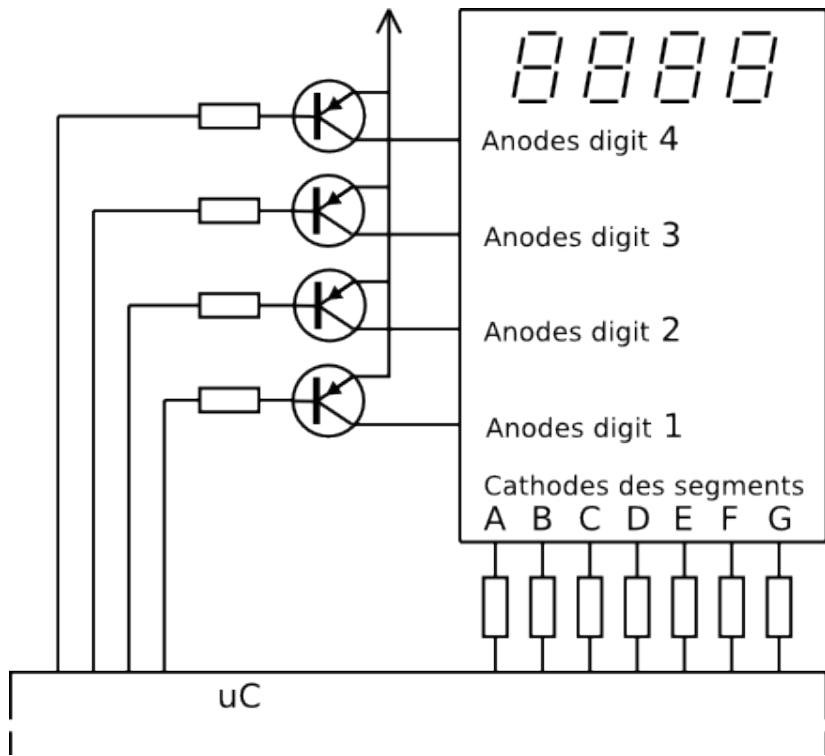


Figure : Schéma de commande d'un afficheur à 7 segments de 4 digits

Le transistor PNP va conduire et allumer le digit concerné lorsqu'une tension négative est appliquée entre sa base et son collecteur. La commande est donc active à l'état 0.

CHAPITRE 10

REGISTRE D'EXTENSION SÉRIE-PARALLÈLE

Pierre-Yves Rochat, EPFL
rév 2015/09/18

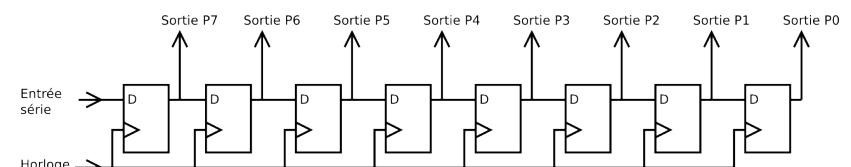
10.1 BESOIN DE BROCHES

Les enseignes et surtout les afficheurs à LED nécessitent beaucoup de broches dont on peut commander l'état. Les microcontrôleurs n'ont généralement pas assez de broches d'entrées-sorties pour faire face à ce besoin. Plusieurs circuits logiques classiques offrent la fonctionnalité de fournir des sorties supplémentaires. Notons par exemple les *latch adressables*, qui sont très pratiques. Le circuit 74HC259 est un latch adressable à 8 sorties.

Mais le composant le plus souvent utilisé dans le domaine des afficheurs à LED est le *registre série-parallèle*.

10.2 REGISTRE SÉRIE

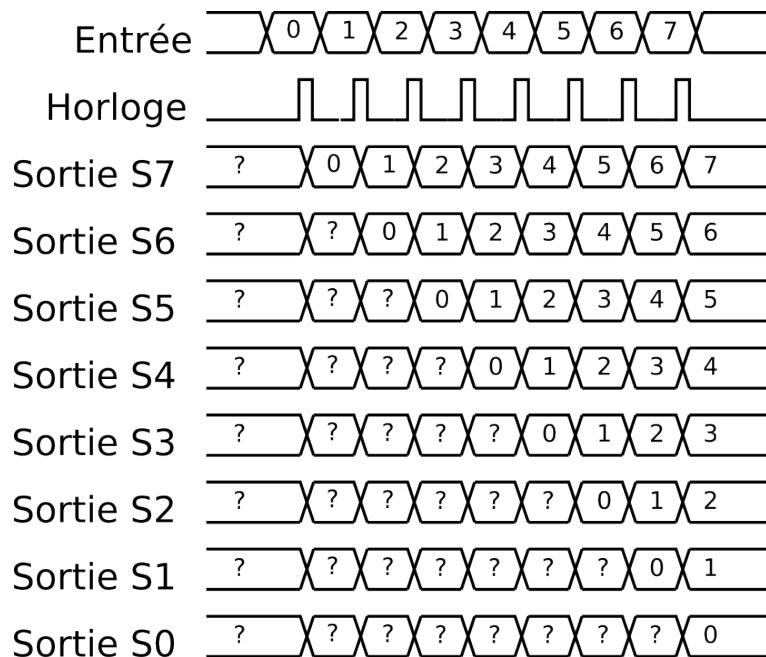
Les registres à décalage sont réalisés avec des bascules. Voici le schéma d'un registre série 8 bits :



Registre série

On y trouve 8 bascules D. Les horloges des 8 bascules sont reliées ensemble. L'entrée D de la première bascule est l'entrée de notre registre. Sa sortie est reliée à l'entrée de la seconde bascule et ainsi de suite. Le système a 8 sorties.

Voici un diagramme des temps qui permet de comprendre comment fonctionne ce registre. Pour chaque bascule D, la valeur de l'entrée est copiée sur la sortie au moment du flanc montant de l'horloge.



Exemple de diagramme des temps d'un registre série

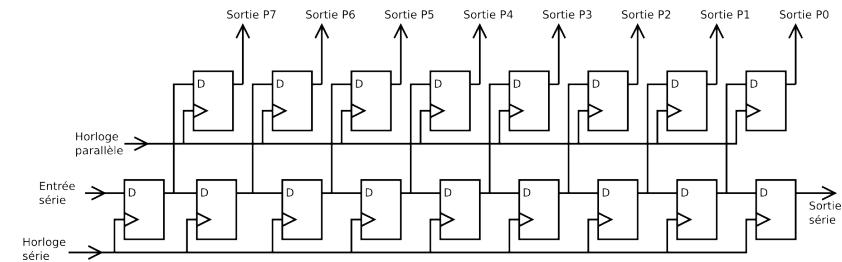
Les valeurs notées 0,1,2..7 correspondent à des valeurs binaires 0 ou 1 placées successivement sur l'entrée. Ces valeurs vont se décaler d'une bascule vers la suivante, à chaque coup d'horloge. Après 8 coups d'horloge, les 8 valeurs envoyées en série vont se retrouver sur les sorties. Il est donc possible par ce moyen de placer n'importe quelle valeur sur les 8 sorties.

Ce dispositif permet donc de disposer 8 sorties, tout en ne monopolisant que 2 broches du microcontrôleur. Mais il a un problème majeur : durant le transfert des données, des valeurs non désirées vont apparaître sur les sorties. Dans certains cas particuliers, ce n'est pas grave. Mais c'est souvent inacceptable. Dans le

cas de la commande de LED, l'œil est très sensible et des artefacts sur les LED deviennent vite gênants.

10.3 REGISTRE SÉRIE-PARALLÈLE

Ajoutons 8 bascules D supplémentaires à notre montage. Ces 8 bascules forment cette fois un registre parallèle, avec 8 entrées et 8 sorties. Comme pour le registre série, les horloges des 8 bascules sont reliées ensemble.



Registre série-parallèle

Sur ce diagramme des temps, on voit que les données transmises en série sont ensuite copiées sur les sorties du registre parallèle.

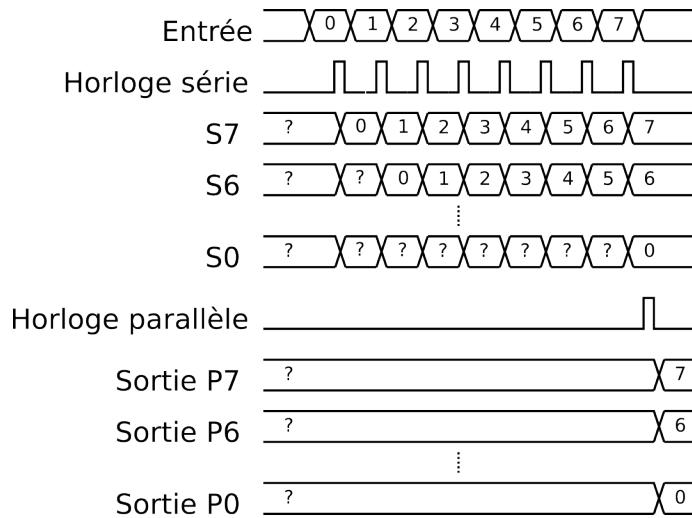
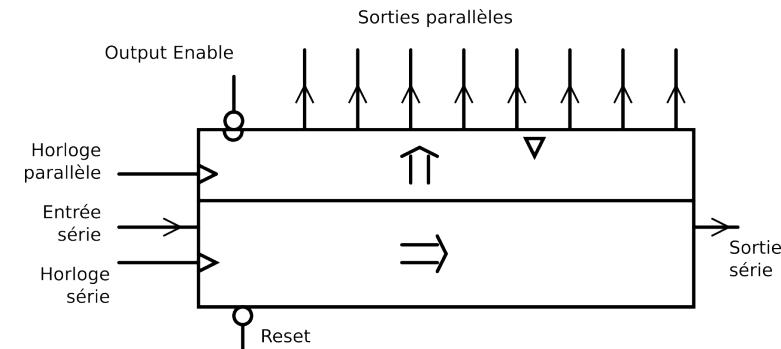


Diagramme des temps d'un registre série-parallèle

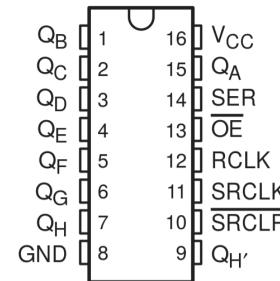
Les 8 valeurs arrivent en même temps sur les sorties du registre parallèle. Les anciennes valeurs sont présentes sur toutes les sorties durant le transfert série et sont mises à jour en même temps. Il n'y a donc pas de risque d'artefacts.

10.4 LE CIRCUIT 74HC595

Le circuit 74HC595 est très souvent présent dans des afficheurs à LED. C'est un circuit C-MOS, de la série 74HCxx. Il comporte un registre série-parallèle de 8 bits. Ses sorties sont à *trois états*, commandées par le signal Output Enable. Une entrée Reset permet de forcer les valeurs du registre de sortie à zéro.

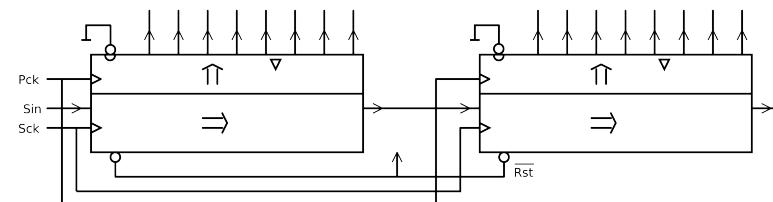


Registre 74HC595



Brochage du 74HC595

Sa sortie série permet de cascader les circuits, c'est-à-dire les placer les uns à la suite des autres, comme le montre la figure suivante :



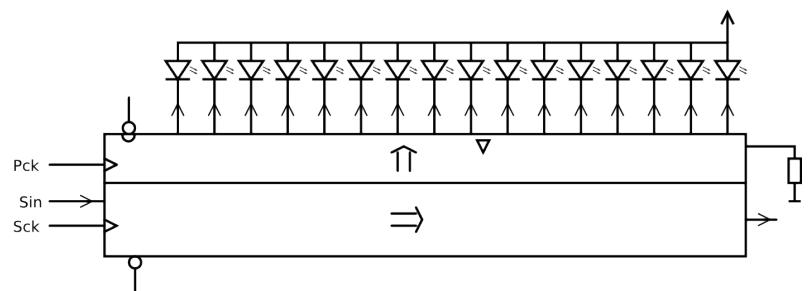
Registre série 16 bits utilisant 2 registres 74HC595 en cascade

Pour ce faire, les horloges doivent être communes à tous les registres : l'horloge série, notée **Sck** et l'horloge parallèle, notée **Pck**. La sortie du premier registre est reliée à l'entrée du deuxième et ainsi de suite. Il est possible de créer de très longs registres. Quelle que soit la longueur du registre, seules trois sorties du microcontrôleur sont nécessaires pour le commander. Si le nombre de registre est vraiment important, il sera judicieux de placer un amplificateur sur les horloges avant de les transmettre au registre suivant. Dans beaucoup d'afficheurs à LED, on trouve des 74HC245 qui jouent ce rôle. Il s'agit de passeurs bi-directionnels, qui ne sont utilisés dans ce cas que dans un sens.

10.5 REGISTRES À SORTIES À COURANT CONSTANT

Bien entendu, les sorties du 74HC595 sont des sorties C-MOS normales. Pour les utiliser pour commander des LED, il faut ajouter en série avec chaque LED la traditionnelle résistance pour limiter le courant.

Il existe plusieurs circuits registres série-parallèles dont les sorties incorporent des sources de courant. Il est ainsi possible de brancher des LED sans la résistance série, ce qui simplifie le schéma :



Registre 16 bits avec sources de courant

La valeur du courant dans toutes les sorties est fixé par une seule résistance, notée R sur le schéma.

Plusieurs fabricants proposent différents modèles de registres série-parallèle avec sources de courant. Parmi les circuits 16 bits, on en trouve un bon nombre dont les noms sont différents, mais qui ont en commun leur brochage. Toshiba propose le TB62701, Texas Instrument le TLC5905, Allegro le A6276, etc. Un fabricant chinois propose le SUM2016, dont je n'ai trouvé la documentation... qu'en chinois !

DRAFT
2016-03-20
01:16:01

10.6 PROGRAMMATION

La procédure pour envoyer 8 bits dans notre registre série parallèle est constituée d'une boucle **for**, répétée autant de fois qu'il y a de bits dans le registre. La valeur binaire à transmettre est d'abord placée sur une sortie, puis un flanc montant est produit sur l'horloge. L'horloge repasse ensuite à 0 pour être prête pour l'envoi de la valeur suivante. Tout à la fin, une impulsion est envoyée sur l'horloge parallèle.

```

1 #define SortieSerieOn P1OUT |= (1<<0)
2 #define SortieSerieOff P1OUT &=~(1<<0)
3
4 #define ClockSerHaut P1OUT |= (1<<1)
5 #define ClockSerBas P1OUT &=~(1<<1)
6
7 #define ClockParHaut P1OUT |= (1<<2)
8 #define ClockParBas P1OUT &=~(1<<2)
9
10 void Envoie8bitsSerie (uint8_t valeur) {
11     uint16_t i;
12     for (i=0; i<8; i++) {
13         if (valeur & (1<<i)) {
14             SortieSerieOn;
15         } else {
16             SortieSerieOff;
17         }
18         ClockSerHaut; ClockSerBas;
19     }
20     ClockParHaut; ClockParBas;
21 }
```

Cette procédure devient souvent la procédure centrale dans un programme qui gère un afficheur à LED, celle qui consomme le plus de temps . Il faut l'écrire avec soin, en allant même jusqu'à regarder comment le compilateur l'a traduite en instructions assembleur, pour chercher à optimiser le code pour une exécution aussi rapide que possible.

Voici par exemple une version plus optimisée de l'envoi des bits :

DRAFT
2016-03-20
01:16:01

```

11 ...
12   for (i=0; i<8; i++) {
13     if (valeur & (1<<0)) {
14       SortieSerieOn;
15     } else {
16       SortieSerieOff;
17     }
18     ClockSerHaut; ClockSerBas;
19     valeur = valeur >> 1;
20 ...

```

Cette version supprime l'opération `valeur & (1<<i)`, qui prend davantage de cycles d'horloge du microprocesseur que l'instruction `valeur & (1<<0)`.

Ce n'est qu'un petit exemple d'optimisation. Une autre technique sera décrite plus loin dans ce cours : il s'agit de l'Accès Direct en Mémoire (Direct Memory Access = DMA). Dans ce cas, ce n'est plus le microcontrôleur qui va effectuer le travail d'envoi des bits, mais un contrôleur spécialisé, disponible dans certains microcontrôleurs comme les ARM.

CHAPITRE 11

AFFICHEURS MATRICIELS

Pierre-Yves Rochat, EPFL
rév 2015/09/16

11.1 AFFICHEURS ET ÉCRANS

Voici une définition du mot afficheur : dispositif électronique permettant de présenter visuellement des données. Cette définition correspond aussi très bien à ce qu'on appelle un écran. Ce terme "écran" vient de la technique des tubes cathodiques, qui comportaient un écran de phosphore, capable de transformer le faisceau d'électrons en une tache lumineuse.

Depuis plusieurs décennies, les LCD (*Liquid Cristal Display*) dominent ce domaine, tant pour de petits afficheurs que pour des écrans de taille respectable. On parle parfois, par abus de langage, d'écrans à LED pour désigner des écrans LCD rétroéclairés par des LED. Il ne faut pas les confondre avec les écrans à O-LED (LED organiques), qui prennent des parts de marché de plus en plus importantes.

Ces domaines ne sont pas le sujet de notre cours. Nous allons nous concentrer sur les dispositifs réalisés avec des LED indépendantes.

11.2 NOTION DE PIXEL

Chaque point d'un afficheur ou d'un écran est appelé un *pixel*. Il peut être d'une seule couleur (monochrome) ou capable de prendre plusieurs couleurs (polychrome ou multicolore). Dans le cadre de ce cours, les mots *points* et *pixels* sont synonymes et utilisés indifféremment. *Pixel* est un mot-valise formé par la fusion des mots de la locution anglaise *picture element* ou *élément d'image* en français.

Un afficheur est caractérisé par plusieurs paramètres, dont un des plus importants est le nombre de pixels qu'on indique souvent sous la forme de deux paramètres : le nombre de lignes et le nombre de points par lignes.

Dans le domaine des écrans, les modèles VGA des années 1980 affichaient 480 lignes de 640 points. Aujourd’hui, l’écran d’un ordinateur portable à faible coût peut afficher 800 lignes de 1’280 pixels. Une image vidéo *Full HD* affiche 1’080 lignes de 1’920 pixels.

La taille de l’afficheur est évidemment aussi un paramètre important, sa *hauteur* (on part de l’idée que l’écran est vertical) et sa *largeur*.

À partir de la taille et du nombre de pixels, on peut calculer deux autres caractéristiques d’un afficheur :

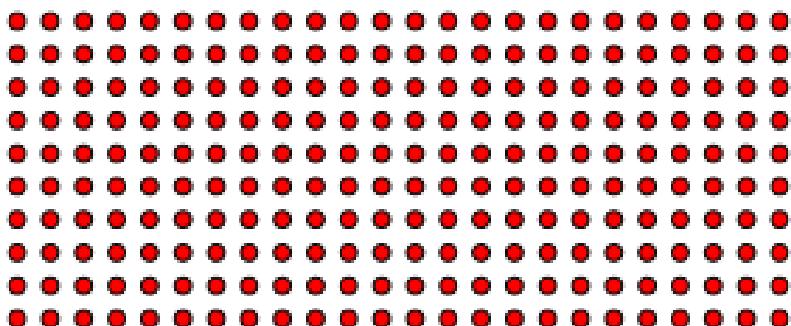
- Sa *résolution* : C’est la distance entre un pixel et son plus proche voisin. On l’exprime généralement en millimètre. Les fabricants donnent souvent une expression comme P6 ou P10. La lettre P vient du mot *Pitch* (le pas). Il s’agit de la distance entre chaque pixel exprimée en millimètre (mm).
- Sa *densité* : C’est le nombre de pixels par unité de surface. L’unité est généralement exprimée en pixels par mètre carrés (px/m^2).

Prenons l’exemple d’un afficheur P6. La distance entre chaque pixel est de 6 mm. On peut donc aligner environ 167 LED sur un mètre. Il faut donc plus de 27’800 LED pour remplir un mètre carré ! En l’absence d’indication contraire, on considère que les résolutions horizontales et verticales sont identiques.

11.3 AFFICHEURS À LED

Un afficheur à LED est donc un ensemble de LED dont il est possible de choisir l’état de chacune d’elles indépendamment des autres.

Les LED sont généralement disposées en lignes et en colonnes : on obtient un afficheur orthogonal.



Afficheur de 10x24 LED

Si la distance est la même horizontalement et verticalement (en x et y), l’afficheur est orthonormé. La géométrie des LED permet de réaliser toutes sortes d’afficheurs, sans se limiter à une grille orthonormée. Il existe des afficheurs cylindriques, sphériques ou en forme de pyramide ! Plus couramment, on trouve des afficheurs qui prennent une forme dont la signification est connue, comme les afficheurs en forme de croix de pharmacie, très répandus depuis quelques années.

La taille des afficheurs à LED varie de manière considérable : on trouve de petits journaux lumineux intégrés à des médaillons de ceinture, mais il existe aussi des afficheurs vidéos d’une surface de plusieurs dizaines de mètres carrés.

Chaque pixel peut être composé d’une LE. Il existe aussi des afficheurs comportant deux LED par pixel, généralement verte et rouge. Il s’agit d’un héritage historique, de l’époque où les LED bleues n’étaient pas disponibles ou hors de prix. Il faut noter que la composition du rouge et du vert donne une couleur ressemblant à l’orange. Finalement, beaucoup d’afficheurs à LED comportent trois LED, rouge, verte et bleue. Il est alors possible d’obtenir toutes les autres couleurs par composition.

11.4 COMMANDE DES LED PAR DES REGISTRES

Le nombre important de LED d’un afficheur matriciel, même de petite taille, ne permet généralement pas une commande de chaque LED par une broche d’un microcontrôleur. C’est seulement le cas pour de petits afficheurs commandés par *multiplexage temporel*, sujet qui sera abordé plus tard dans ce cours. Dans tous les autres cas, des registres sont utilisés pour commander les LED.

Prenons comme exemple l’afficheur de 8 lignes de 16 LED dont le schéma est indiqué sur la figure ci-dessous. Chacune de ses lignes utilise un registre série-parallèle de 16 bits, il y a donc 8 registres. Les registres séries sont indiqués avec une flèche pointant vers la droite. Les registres parallèles sont indiqués avec une flèche pointant vers le haut. Les entrées des horloges des registres sont indiquées par des triangles.

Les horloges des registres séries sont toutes connectées à la broche P1.4, ce qui implique que les données sont chargées en même temps sur tous ces registres. À chaque coup d’horloge, la valeur présentée à l’entrée est décalée dans le registre. Sur la figure, les valeurs d’entrée sont données par les broches P2.0, P2.1...P2.7.

Une fois les 16 valeurs introduites dans les registres séries, elles sont transférées aux registres parallèles dont les horloges sont toutes connectées à la broche P1.5 du microcontrôleur. Les valeurs ainsi chargées dans les registres parallèles sont immédiatement affichées sur les LED.

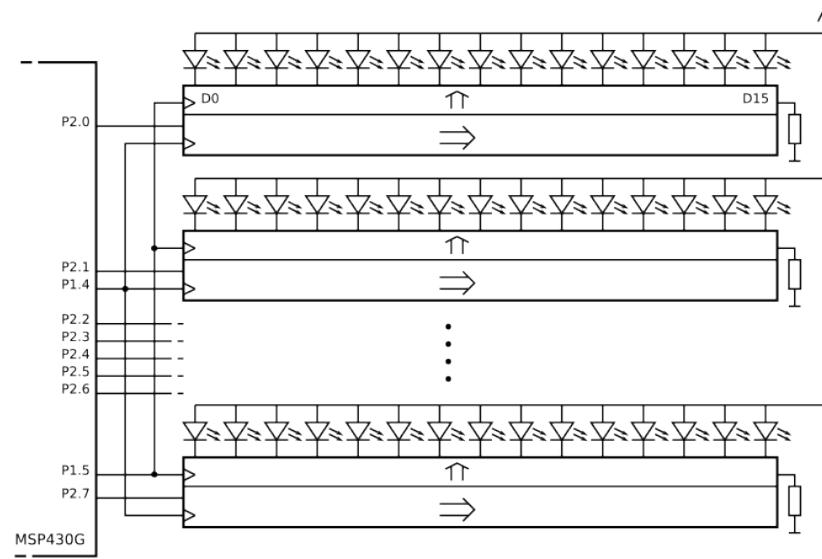


Schéma d'un afficheur comportant 8 lignes de 16 LED

Remarquez que les LED n'ont pas de résistances série. Les registres utilisés contiennent des sources de courant ajustables au moyen d'une seule résistance pour tout le registre.

11.5 PROGRAMMATION

Comment écrire le programme qui contrôle ce montage ? On souhaite par exemple faire défiler un texte, pour afficher un journal lumineux. La première idée qui vient à l'esprit est d'utiliser les propriétés du registre pour introduire successivement les colonnes de pixels qui forment chaque caractère. Voici un programme qui pourrait fonctionner. Il se limite pour le moment à envoyer un motif en *dents de scie* :

```

1 int main() {
2     init(); // initialisations...
3     uint8_t i;
4     while (1) {
5         for (i=0; i<16; i++) { // envoie une colonne avec un seul pixel allumé
6             P1OUT = (1<<(i&7)); // 1 col de 8 px, 1 seul allumé -> dents de scie

```

DRAFT
2016-03-20
01:16:01

```

7     SerClockOn; SerClockClear; // envoie un coup d'horloge série
8     ParClockOn; ParClockClear; // envoie un coup d'horloge
9 }
10 }
11 }

```

Pour générer des caractères, il faut disposer d'une table décrivant les positions des pixels des différents caractères. Voici une manière de les représenter :

```

1 const uint8_t GenCar [] { // tableau des pixels des caractères
2     0b01111110, // caractère 'A'
3     0b00001001, // Il faut pencher la tête à droite
4     0b00001001, // pour voir sa forme !
5     0b00001001,
6     0b01111110,
7     0b01111111, // caractère 'B'
8     0b01001001, // Les caractères forment
9     0b01001001, // une matrice de 5x7
10    0b01001001,
11    0b00110110,
12    0b00111110, // caractère 'C'
13    0b01000001, // Les caractères ont ici
14    0b01000001, // une chasse fixe, c'est-à-dire
15    0b01000001, // que tous les caractères ont
16    0b01000001 // la même largeur en pixels
17 };
18
19 }

```

Voici un programme qui affiche un texte :

```

1 const char *Texte = "ABC\0"; // texte, terminé par le caractère nul
2 char *ptTexte; // pointeur vers le texte à afficher
3
4 int main(void) {
5     init(); // initialisations...
6     while(1) { // le texte défile sans fin
7         ptTexte = Texte;
8         while (*ptTexte!='\0') { // boucle des caractères du texte
9             caractère = *ptTexte; // le caractère à afficher

```

DRAFT
2016-03-20
01:16:01

```

10 idxGenCar = (caractere-'A') * 5; // conversion ASCII à index GenCar[]
11 for (i=0; i<5; i++) { // envoie les 5 colonnes du caractère
12     P2OUT = ~GenCar[idxGenCar++]; // 1 colonne du caractère (actif à 0)
13     SerClockSet; SerClockClear; // coup d'horloge série
14     ParClockSet; ParClockClear; // coup d'horloge parallèle
15     AttenteMs (delai);
16 }
17 ptTexte++; // passe au caractère suivant
18 P2OUT = ~0; // colonne vide, séparant les caractères
19 SerClockSet; SerClockClear; // coup d'horloge série
20 ParClockSet; ParClockClear; // coup d'horloge parallèle
21 AttenteMs (delai);
22 }
23 }
24 }
```

Dans l'exemple ci-dessus, le texte à afficher est enregistré dans un tableau. L'instruction `const` indique au compilateur que ce tableau peut être stocké en mémoire de programme. Sans cette instruction, il aurait été enregistré en mémoire RAM qui est souvent nettement plus petite que la mémoire de programme. Pour accéder aux caractères de ce texte, un pointeur est utilisé. La déclaration du pointeur s'écrit : `const char *ptTexte;`. Le symbole `*` indique qu'il s'agit d'un pointeur.

Cette manière d'envoyer les caractères fonctionne, mais présente tellement de limitations qu'elle ne sera jamais utilisée. Par exemple, elle ne peut pas fonctionner si l'ordre des LED est inversé : le texte ne pourra pas être décalé correctement de droite à gauche. Elle est aussi incompatible avec les afficheurs multiplexés.

11.6 GÉNÉRATION ET RAFRAÎCHISSEMENT SÉPARÉS

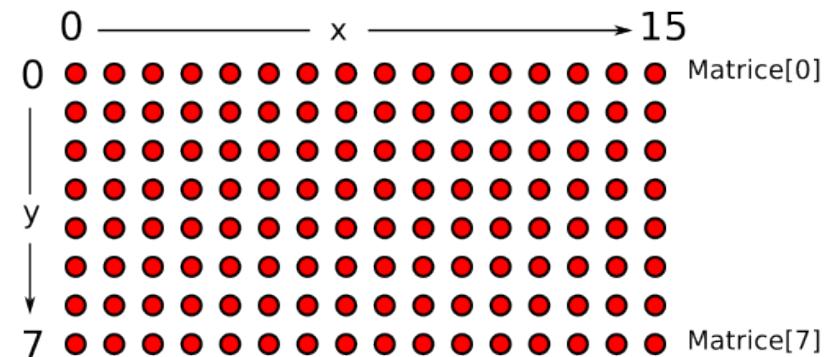
La bonne manière de programmer un afficheur est de **séparer** complètement la génération de l'image à afficher et l'envoi de cette image sur l'afficheur. La valeur courante de chaque pixel est placée dans une **mémoire**. La partie du logiciel qui prépare les images **écrit** dans cette mémoire. Les procédures qui envoient les informations à l'afficheur **lisent** dans cette mémoire.

Dans notre exemple, l'afficheur a 8 lignes de 16 pixels. Un mot de 16 bits pourra donc stocker une ligne. Voici comment réservier la zone mémoire pour les pixels :

```

#define NbLignes 8
uint16_t Matrice[NbLignes]; // mots de 16 bits, correspondant à une ligne
```

Nous choisissons de placer les axes x et y de la manière suivante :



Organisation de l'afficheur 8x16 pixels

Les procédures permettant d'allumer et d'éteindre un pixel, désigné par ses coordonnées, sont particulièrement simples dans ce cas :

```

1 void AllumePoint(int16_t x, int16_t y) {
2     Matrice[y] |= (1<<x); // set bit
3 }
4
5 void EteintPoint(int16_t x, int16_t y) {
6     Matrice[y] &=~(1<<x); // clear bit
7 }
```

Voici une procédure pour afficher une diagonale en travers de l'afficheur :

```

1 #define MaxX 16
2 #define MaxY NbLignes
3
4 void Diagonale() {
5     int16_t i;
6     for (i=0; i<MaxY; i++) {
```

```

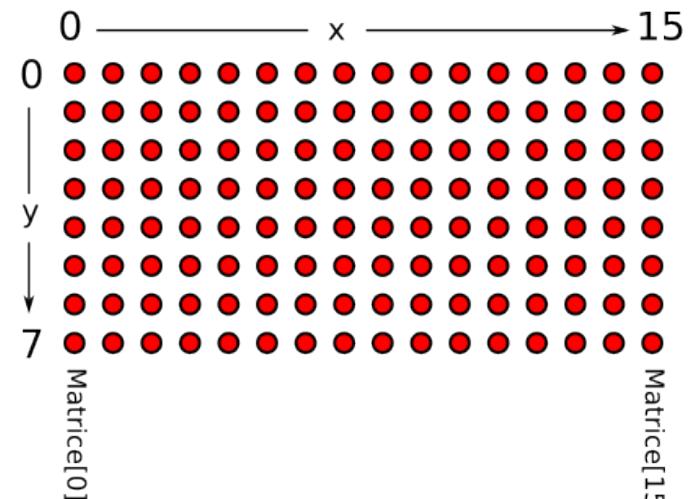
7     AllumePoint(i*MaxX/MaxY, i);
8 }
9 }
```

Mais toutes ces procédures ne vont rien afficher sur les LED ! Il faut encore une procédure qui va placer chaque pixel sur la LED correspondante. Pour l'écrire, il faut garder en mémoire l'organisation matérielle de notre matrice, avec les 8 registres série-parallèles de 16 bits.

```

1 void AfficheMatrice() {
2     for (uint16_t x=0; x<MaxX; x++) {
3         // Préparation des valeurs qui doivent être envoyées aux 8 registres:
4         for (uint16_t y=0; y<MaxY; y++) {
5             if (Matrice[y] & (1<<x)) P2OUT &= ~(1<<y); else P2OUT |= (1<<y);
6         }
7         SerClockSet; SerClockClear; // envoie un coup d'horloge série
8     }
9     ParClockSet; ParClockClear; // envoie les valeur sur les LED
10 }
```

Cette procédure semble compliquée. Une organisation optimisée des données en mémoire pourrait la simplifier :



Organisation plus optimale des pixels en mémoire

Voici la définition et la procédure correspondante :

```

1 #define NbColonnes 16
2 uint8_t Matrice[NbColonnes]; // mots de 8 bits, correspondant à une colonne
3
4 void AfficheMatrice() {
5     for (uint16_t x=0; x<MaxX; x++) {
6         P2OUT = ~Matrice[x];
7         SerClockSet; SerClockClear; // envoie un coup d'horloge série
8     }
9     ParClockSet; ParClockClear; // envoie les valeur sur les LED
10 }
```

Non seulement la procédure `AfficheMatrice()` est beaucoup plus simple, mais en plus elle va prendre moins de temps à être exécutée. Dans notre cas, la vitesse ne pose pas de problème. Mais dès que les afficheurs deviennent plus grands, cette question devient cruciale.

De manière générale, on va donc chercher à optimiser l'organisation des pixels en mémoire en vue de simplifier et de rendre plus rapide l'envoi des pixels sur les LED, quitte à compliquer un peu les procédures qui créent les images.

11.7 PROGRAMMER DES ANIMATIONS

Pour générer des animations sur l'afficheur, il faut :

- préparer une image en mémoire,
- envoyer son contenu sur l'afficheur,
- attendre le temps nécessaire,
- préparer une autre image
- et ainsi de suite.

Voici un programme complet qui génère une animation graphique sur notre afficheur :

```

1 // Afficheur didactique 16x8
2 // Les 8 bits sont sur P2
3 // Usage d'une matrice en bytes
4 // Ping !
5
6 #include <msp430g2553.h>
7
8 #define DELAI 100
9
10 #define SerClockSet P1OUT|=(1<<5)
11 #define SerClockClear P1OUT&=~(1<<5)
12
13 #define ParClockSet P1OUT|=(1<<4)
14 #define ParClockClear P1OUT&=~(1<<4)
15
16 void AttenteMs (uint16_t duree) {
17     for (uint16_t i=0; i<duree; i++) {
18         for (volatile uint16_t j=0; j<500; j++) {
19             }
20     }
21 }
22
23 #define NbColonnes 16
24 uint8_t Matrice[NbColonnes]; // mots de 8 bits, correspondant à une colonne
25
26 #define MaxX NbColonnes
27 #define MaxY 8
28
29 void AllumePoint(int16_t x, int16_t y) {
30     Matrice[x] |= (1<<y); // set bit

```

```

31 }
32
33 void EteintPoint(int16_t x, int16_t y) {
34     Matrice[x] &=~(1<<y); // clear bit
35 }
36
37 void AfficheMatrice() {
38     for (uint16_t x=0; x<MaxX; x++) {
39         P2OUT = ~Matrice[x];
40         SerClockSet; SerClockClear; // envoie un coup d'horloge série
41     }
42     ParClockSet; ParClockClear; // envoie les valeurs sur les LED
43 }
44
45 void Ping() {
46     int16_t x=0;
47     int16_t y=0;
48     int8_t sensX=1;
49     int8_t sensY=1;
50     do {
51         AllumePoint(x,y);
52         AfficheMatrice();
53         AttenteMs(DELAI);
54         EteintPoint(x,y);
55         x+=sensX;
56         if(x==(MaxX-1)) sensX=(-1);
57         if(x==0) sensX=1;
58         y+=sensY;
59         if(y==(MaxY-1)) sensY=(-1);
60         if(y==0) sensY=1;
61     } while (!((x==0)&(y==0)));
62 }
63
64 int main(void) {
65     WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
66     BCSCTL1 = CALBC1_16MHZ; DCOCTL = CALDCO_16MHZ; // Horloge à 16 MHz
67     P1DIR = (1<<4)|(1<<5); P2DIR = 0xFF; P2SEL = 0;
68
69     for (uint16_t i=0; i<NbColonnes; i++) { // initialise la matrice
70         Matrice[i]=0x0;
71     }
72
73     while(1) {
74         Ping();

```

CHAPITRE 12

MÉMOIRES PERMANENTES

*Pierre-Yves Rochat, EPFL
rév 2015/09/18*

12.1 TYPES DE MÉMOIRES

Tous les microcontrôleurs disposent de deux types de mémoire :

- la mémoire Flash, prévue principalement pour recevoir le programme, qui est permanente
- la mémoire vive (RAM = Random Access Memory), qui perd son contenu lorsque le circuit n'est plus alimenté.

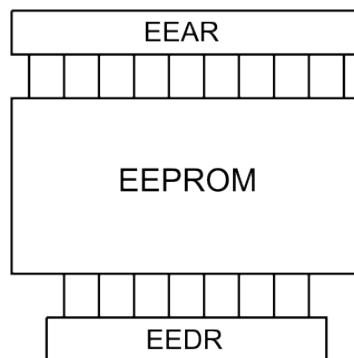
Pour commander des enseignes et afficheurs à LED, on placera souvent en mémoire non volatile non seulement le programme proprement dit, mais aussi les informations sur le contenu qui sera visualisé. En effet, les séquences d'une enseigne ou les textes d'un afficheur doivent être mémorisés de manière non-volatile, pour leur assurer un fonctionnement correct après une coupure de courant.

Une particularité de la mémoire Flash est son organisation en blocs. S'il est possible d'écrire une valeur dans une position mémoire précise, il n'est pas possible d'effacer une position mémoire seule. Les effacements se font toujours par bloc. La taille d'un bloc varie considérablement d'un microcontrôleur à un autre, parfois seulement 64 octets, jusqu'à plusieurs dizaines de kB.

12.2 LES MÉMOIRES EEPROM

Certains microcontrôleurs disposent d'un troisième type de mémoire, en plus de la Flash et de la RAM. Il s'agit de mémoires EEPROM. Ce sont aussi des mémoires non-volatiles, mais chaque position mémoire peut être écrite et effacée indépendamment des autres. On trouve ce type de mémoire dans les microcontrôleurs AVR, dont l'ATmega328, bien connu pour être utilisé dans les Arduino.

L'accès à l'EEPROM interne d'un microcontrôleur est généralement très simple. Des registres sont prévus pour l'adresse (EEAR) et pour la donnée (EEDR), en lecture ou en écriture. Sur les AVR, deux registres donnent l'accès aux adresses et aux données :



Accès à l'EEPROM sur les AVR

Voici les instructions pour lire et écrire dans la mémoire :

```

// Lecture en EEPROM :
EEAR = adresse; // l'adresse est donnée
EECR = (1<<EERE); // le fanion de lecture est activé
valeur = EEDR; // lecture de la valeur

// Ecriture en EEPROM :
while (EECR & (1<<EEPE)) {} // attend la fin d'une écriture précédente
EEAR = adresse; // l'adresse est donnée
EEDR = valeur; // la valeur est donnée
EECR = (1<<EEMPE); // autorise une écriture (Master Write Enable)
EECR = (1<<EEPE); // lance le cycle d'écriture (Write Enable)

```

Le fanion EEMPE signifiait EEprom Master Program Enable. Mais les lettres PE sont traduites dans la documentation actuelle par Write Enable. EEMPE doit être activé juste avant l'activation du fanion EEPE (EEprom Write Enable) qui lance le cycle d'écriture. Son activation ne dure que quelques cycles d'horloge : il est automatiquement remis à zéro. Aucune écriture n'est possible s'il n'est pas actif. Son but est de rendre improbable une écriture accidentelle en EEPROM. L'activation des EEMPE et de EEPE doivent en effet se suivre de près pour qu'une écriture soit possible.

DRAFT
2016-03-20
01:16:01

Le fanion EEPE est remis à zéro automatiquement par le microcontrôleur lorsque le cycle d'écriture est terminé. Alors que la lecture est immédiate, il faut se souvenir que l'écriture prend un temps relativement long, de l'ordre de quelques millisecondes.

12.3 ACCÈS À LA MÉMOIRE FLASH PAR PROGRAMMATION

Pour y placer le programme, la mémoire Flash du microcontrôleur est accédée de l'extérieur, au moyen d'un programmeur. S'il reste de la place dans la mémoire Flash, il est possible de l'utiliser depuis le programme pour y placer des données. La manière de le faire dépend de l'architecture du microcontrôleur. Il faut se référer à la documentation du microcontrôleur pour avoir les détails.

Sur les MSP430, l'architecture de Von Neumann rend facile l'accès à la mémoire Flash. La lecture se fait simplement en indiquant l'adresse de la position mémoire :

```

// Lecture en Flash :
uint8_t *pointeur; // pointeur dans la Flash
pointeur = (uint8_t *) 0x1040; // place l'adresse dans le pointeur
uint8_t valeur = *pointeur;

```

On remarque le trans-typage de l'adresse, indiquée ici en hexadécimal, vers le type du pointeur, qui est dans ce cas un pointeur vers des valeurs 8 bits non-signées (`uint8_t *`).

L'écriture se fait de la même manière. Il faut toutefois précéder l'écriture par la désactivation du fanion qui bloque l'écriture en mémoire Flash. Son rôle est de rendre improbable des effacements accidentels de la mémoire.

```

// Ecriture en Flash :
FCTL3 = FWKEY; // Clear Lock bit
*pointeur = valeur; // écrit la valeur dans la Flash
FCTL3 = FWKEY + LOCK; // Set LOCK bit

```

L'effacement s'effectue par bloc. Il s'effectue lorsqu'une écriture est faite dans la zone du bloc alors que le fanion d'effacement est activé. Mais il est nécessaire de désactiver au préalable le fanion qui bloque toute écriture en mémoire Flash.

```

// Effacement d'un bloc de la mémoire Flash
FCTL1 = FWKEY + ERASE; // Set Erase bit

```

DRAFT
2016-03-20
01:16:01

```

FCTL3 = FWKEY; // Clear Lock bit
*pointr = 0; // lance l'effacement du bloc, valeur sans importance
FCTL3 = FWKEY + LOCK; // Set LOCK bit
FCTL1 = FWKEY; // Clear WRT bit

```

Souvent, des librairies sont disponibles pour faciliter ce travail. Pour les AVR, construits sur une architecture de Harvard, la librairie `Pgmspace.h` est utilisée.

12.4 LIMITÉ DU NOMBRE DE CYCLES D'ÉCRITURE

Toutes les mémoires non-volatiles ont une limite dans le nombre d'écritures ou d'effacements. Pour les Flash, la limite est souvent de 10'000 cycles. S'il ne s'agit que de programmer le microcontrôleur, ce nombre semble très grand : qui va programmer 10'000 fois un même microcontrôleur ?

Toutefois... à l'occasion du MOOC Comprendre les microcontrôleurs, proposé par l'EPFL sur Coursera dès 2013, nous avons mis au point un système de correction de devoirs. Il a corrigé à ce jour plus de 70'000 devoirs. Les microcontrôleurs ont donc été changés de temps en temps, pour éviter des pannes.

Les EEPROM sont souvent capables d'un nombre de cycles d'écriture plus grand que les Flash. L'ordre de grandeur est de 100'000 cycles ou davantage. Mais attention, ce nombre est très vite atteint si certaines précautions ne sont pas prises. Remarquez par exemple qu'en écrivant une valeur chaque seconde, on atteint la limite peu après un jour de fonctionnement.

12.5 MÉMOIRES EXTERNES

Il est possible d'ajouter de la mémoire non-volatile à un microcontrôleur en utilisant des circuits mémoire externes. On choisira un circuit en fonction de la taille des données à mémoriser. Les solutions les plus courantes sont :

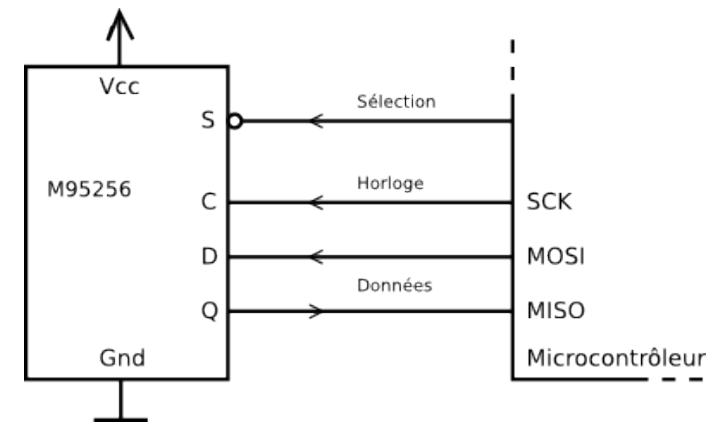
- les mémoires EEPROM, souvent disponibles dans des boîtier à 8 broches (DIL 8 ou modèles correspondant en SMD)
- les cartes SD (ou microSD).

Au moyen d'une autre technologie, les circuits horloges temps réel offrent souvent quelques positions mémoires RAM. Or celles-ci sont rendues non-volatiles par la pile qui maintient le circuit en fonctionnement permanent.

12.6 EEPROM I2C OU SPI

Plusieurs fabricants proposent des familles de mémoires EEPROM, tels que Microchip, Atmel, STmicro, Winbond, etc. Le dialogue entre ces mémoires et le microcontrôleur se fait en série, le plus souvent par des signaux aux normes I2C ou SPI.

Par exemple, la mémoire M95256 de STmicro a une capacité de 256 kb. Attention, la lettre b est en minuscule, ce qui signifie bit. En d'autre termes, c'est une mémoire de 32 kB (kilo Bytes). Elle utilise les signaux SPI. Voici un schéma de mise en œuvre :



Mise en œuvre d'une EEPROM

Pour dialoguer avec ces circuits pour lire ou écrire des valeurs en mémoire, on trouve facilement sur Internet des librairies prêtées à l'emploi. La lecture des sources permet d'en comprendre le fonctionnement.

Il faut être attentif à un point important. Presque tous les microcontrôleurs sont équipés de contrôleurs spécialisés pour I2C ou pour SPI. Pour pouvoir utiliser ces contrôleurs, il faut impérativement utiliser les broches choisies à cet effet par le fabricant. On n'est donc pas libre de choisir les broches.

Bien entendu, il est possible de programmer les procédures I2C ou SPI en *bit banging*, c'est-à-dire en agissant directement sur les broches d'entrée-sortie. Dans ce cas, on a toute liberté dans le choix des broches. L'usage des contrôleurs intégrés a deux avantages. D'une part, le travail de programmation est simplifié.

Mais surtout, ces contrôleurs permettent d'atteindre des vitesses de transfert plus élevées. Pour les afficheurs à LED de grande taille, c'est un aspect très important.

On trouve des mémoires EEPROM dont les capacités vont de quelques dizaines d'octets jusqu'à des dizaines de mégaoctets.

12.7 CARTES SD

Pour des capacités plus grandes, on choisira des cartes SD. Elles utilisent en interne des mémoires Flash, mais un contrôleur s'occupe de gérer l'accès aux blocs. Il gère également les mauvais blocs, qui peuvent souvent apparaître avec ce type de mémoires.

Les cartes SD seront aussi utilisées lorsqu'il est nécessaire de déplacer la mémoire pour la connecter à un PC. Dans le cas d'un afficheur à LED, on peut envisager de mettre le contenu à afficher dans une carte SD à partir d'un PC, puis de placer la carte dans l'afficheur.

La capacité des cartes SD est très importante, aujourd'hui jusqu'au To (1 tera octet = 1'000'000'000'000 octets).

12.8 SYSTÈME DE FICHIER

Pour placer ou lire des données sur une carte SD, on choisit généralement d'utiliser le format spécifique d'un système de fichiers (*file system*). On bénéficie ainsi d'un accès aux données par fichier. On peut aussi utiliser les dossiers, appelés aussi répertoires. Le standard FAT32 est le plus souvent utilisé, vu qu'il est compatible avec tous les systèmes d'exploitation.

Pour faciliter l'accès aux données des fichiers en lecture ou en écriture, de nombreuses bibliothèques sont disponibles : *FatFS*, *PetitFat*, *SdFat*, etc.

À titre d'exemple, la bibliothèque ***PetitFat*** fournit les primitives suivantes :

Procédure	Rôle
<code>pf_mount:</code>	Monter un volume
<code>pf_open:</code>	Ouvrir un fichier
<code>pf_read:</code>	Lire des données dans un fichier
<code>pf_write:</code>	Écrire des données dans un fichier
<code>pf_lseek:</code>	Déplacer le pointeur de lecture ou d'écriture
<code>pf_opendir:</code>	Ouvrir un dossier
<code>pf_readdir:</code>	Lire le contenu d'un dossier.

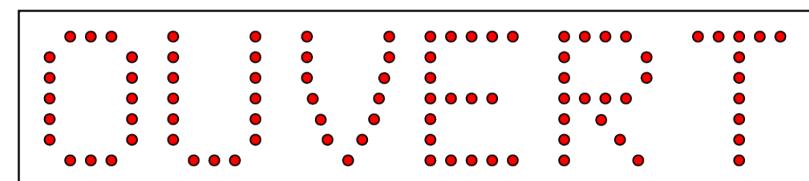
CHAPITRE 13

SÉQUENCEURS À COMPTEURS

*Yves Tiecoura, INP-HB Yamoussoukro
et Pierre-Yves Rochat, EPFL
rév 2015/11/11*

13.1 ANIMATION D'UNE ENSEIGNE

Une enseigne peut être jolie simplement par sa forme et les couleurs des LED. Mais des animations la rendent encore plus attrayante. Une animation, c'est une séquence, ce sont des états qui se succèdent, avec des valeurs pour chaque groupe de LED.



Enseigne à animer

Prenons un exemple : pour attirer l'œil, une enseigne avec le mot OUVERT va être animée avec la séquence dont voici le diagramme des temps :

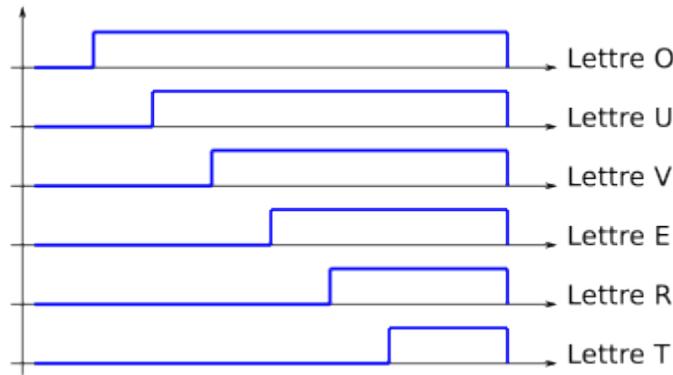
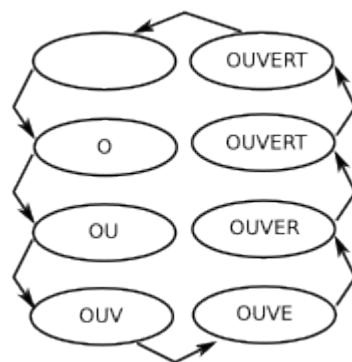


Diagramme des temps d'une séquence pour une enseigne

Une autre manière de représenter cette séquence est un graphe d'état :



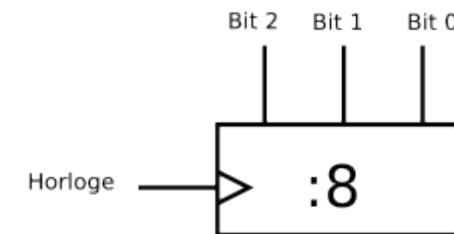
Graphe d'état d'une séquence pour une enseigne

Remarquons qu'il n'y a aucune condition pour passer d'un état à l'autre : le système change d'état à chaque coup d'horloge.

Nous avons volontairement utilisé des durées constantes pour chaque état, pour simplifier la réalisation. Nous remarquons que la séquence choisie a 8 valeurs. Or un compteur binaire de 3 bits passe aussi par 8 valeurs.

13.2 COMPTEUR BINAIRE

Un compteur binaire est un système séquentiel qui va fournir sur ses sorties les valeurs binaires successives. Voici le symbole d'un compteur par 8, ou compteur 3 bits.



Symbole d'un compteur binaire

Les sorties sont notées *Bit 0* pour le bit de poids faible (lsb), *Bit 1* et *Bit 2* pour le bit de poids fort (msb).

Voici son diagramme des temps :

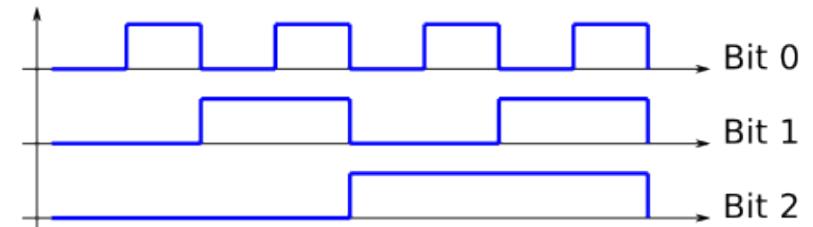
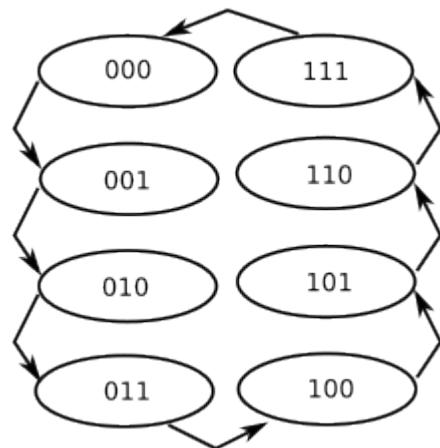


Diagramme des temps d'un compteur binaire

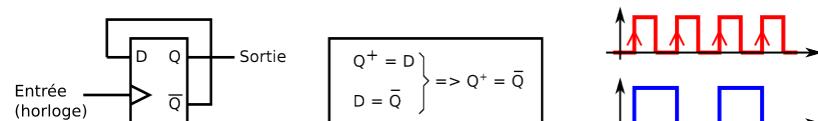
Voici son graphe d'état :



Graphe d'un compteur binaire

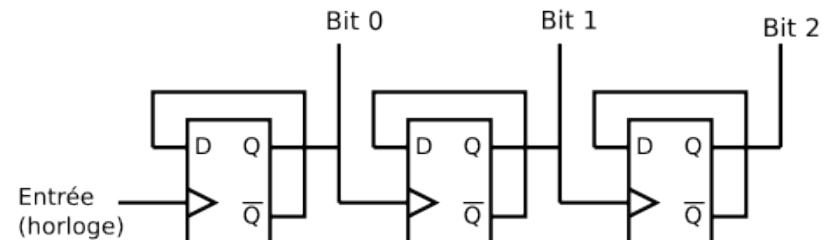
De manière générale, un compteur n bits donne une séquence de 2^n valeurs sur ses sorties.

La base d'un compteur binaire est un compteur par deux, facile à réaliser avec une bascule D, dont on relie la sortie inversée à l'entrée.

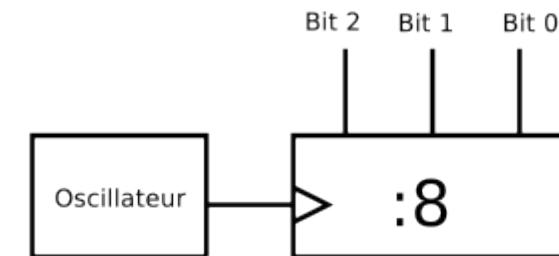


Compteur par deux réalisé avec une bascule D

On peut mettre en cascade plusieurs compteurs par 2 pour réaliser un compteur binaire par 2^n . On appelle aussi parfois ce type de compteurs un compteur modulo 2^n , ce qui met en évidence qu'il reprend la valeur zéro après 2^n-1 .

Compteur binaire par 2^3

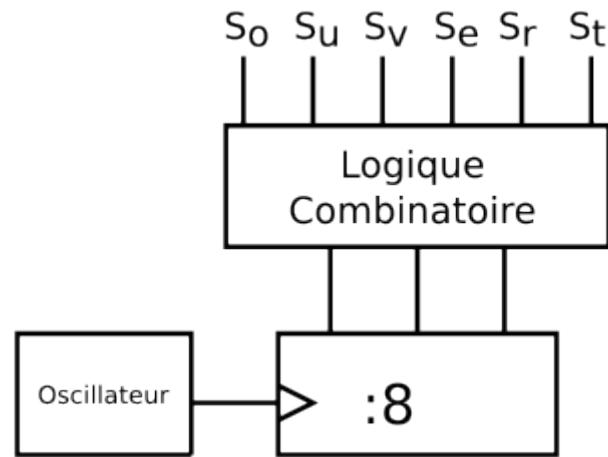
En reliant l'entrée Horloge du compteur à un oscillateur, on obtient 2^n temps de même durée. On a donc réalisé la séquence souhaitée, sauf que les valeurs pour chacun des état ne correspondent pas aux valeurs souhaitées pour notre enseigne.



Générateur de séquence binaire

13.3 LOGIQUE DE DÉCODAGE POUR LES SÉQUENCES

Un simple système combinatoire va nous permettre d'obtenir les signaux de commande des groupes de LED :



Générateur de séquence pour l'enseigne

Voici sa table de vérité :

A	B	C	S_0	S_u	S_v	S_e	S_r	S_t
0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0
0	1	0	1	1	0	0	0	0
0	1	1	1	1	1	0	0	0
1	0	0	1	1	1	1	0	0
1	0	1	1	1	1	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

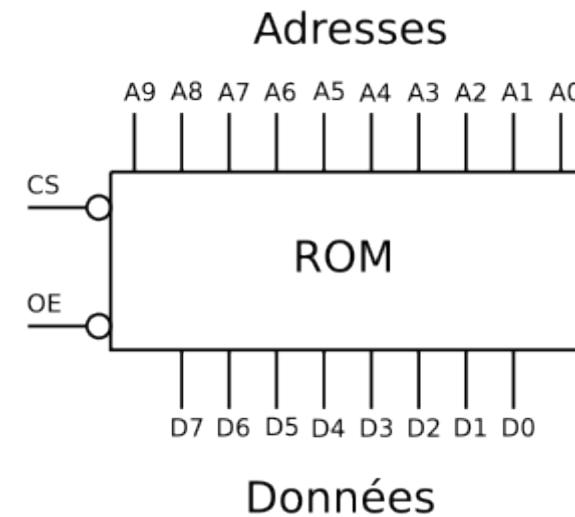
Table de vérité du système combinatoire

Il est possible d'obtenir le schéma le plus simple pour cette logique combinatoire par simplification avec des tables de Karnaugh. Mais on voit déjà que le montage résultant sera compliqué. D'autre part, cette solution n'est pas flexible.

Tout changement de la séquence va aboutir à une refonte complète du schéma logique. Ce n'est pas la bonne piste...

13.4 MÉMOIRE MORTE

On appelle *mémoire morte* ou ROM (Read Only Memory), un circuit mémoire à contenu fixe. Voici comment elle se présente :



Mémoire morte

On remarque des entrées, qui sont les adresses ("Address" en anglais), ainsi que des sorties, qui sont les données (*Data* en anglais). Des entrées de sélections sont généralement disponibles, comme le signal OE (*Output Enable* = sélection des sorties). Pour chaque combinaison des adresses, une valeur particulière est présente sur les sorties.

Dans le cas simple de notre enseigne, il faudrait une ROM munie de seulement 3 bits d'adresse et de 6 bits de donnée. Bien entendu, il suffira de laisser à zéro les adresses non utilisées d'une ROM comportant davantage d'adresses.

Voici le schéma complet de notre commande d'enseigne :

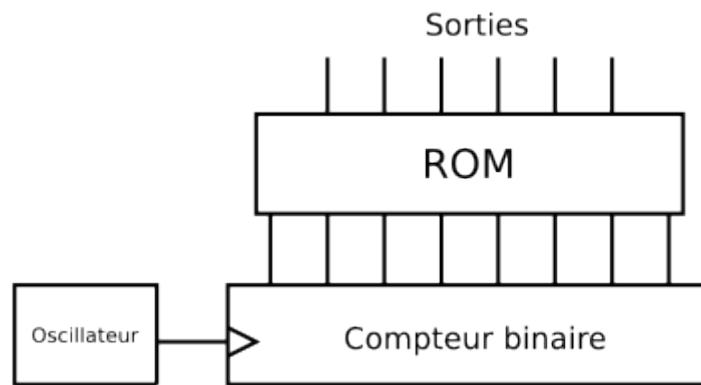


Schéma du séquenceur à ROM

13.5 TYPES DE ROM ET PROGRAMMATION

Faire réaliser une ROM avec un contenu spécifique pour un type d'enseigne est possible, mais n'est intéressant que lorsque la quantité est importante. On utilisera plus souvent des ROM programmables.

Des PROM contenant des fusibles ont été beaucoup utilisées dans les années 1970. Bien qu'existant encore chez certains fabricants, on utilisera plus volontiers des ROM programmables et effaçables : EPROM (Erasable Programmable Read Only Memory).

La technologie des années 1970 et 1980 est encore disponible, avec des mémoires effaçables par ultra violet. Des appareils dédiés permettent l'effacement de la mémoire en une dizaine de minutes. Mais ces mémoires doivent posséder une fenêtre, pour que les UV puissent atteindre la surface du circuit intégré.



EPROM avec sa fenêtre en quartz

Nous avons encore trouvé en 2007 des électroniciens qui utilisaient une EPROM et un compteur binaire C-MOS pour la commande d'enseignes commerciales. C'était à Douala, au Cameroun. Nous ignorons si ces technologies sont encore utilisées actuellement. Le schéma devait être approximativement le suivant :

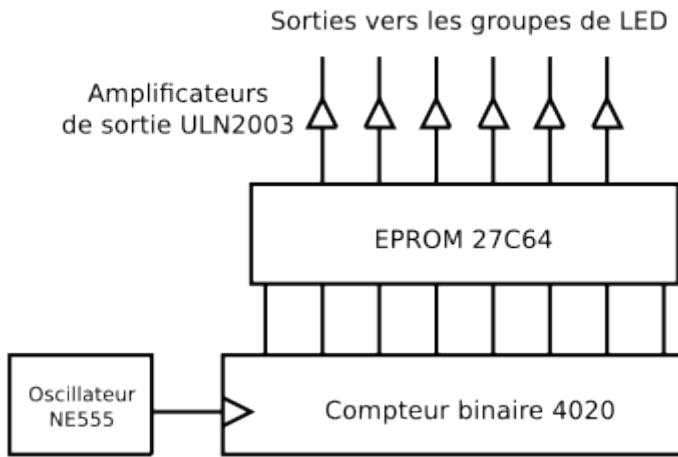


Schéma et circuits du séquenceur à ROM

Pour ne pas devoir utiliser les coûteux boîtiers avec une fenêtre, les fabricants sont parvenus dans les années 1990 à produire des mémoires EEPROM : Electrically Erasable Programmable Read Only Memory. A noter que cette technologie est souvent présente dans les microcontrôleurs.

13.6 SOLUTION PLUS SIMPLE

Aujourd’hui, il est nettement plus simple de réaliser des commandes d’enseignes avec un microcontrôleur. Non seulement, le schéma est beaucoup plus simple, basé sur un seul circuit intégré. Mais le coût des composants est nettement plus faible.

D’autre part, une plus grande flexibilité peut être obtenue :

- le temps peut être divisé en durées variables
- la production de signaux PWM permet une variation de l’intensité des LED
- le changement de la séquence se fait par une simple programmation du microcontrôleur

CHAPITRE 14

LES INTERRUPTIONS

*Yves Tiecoura, INP-HB Yamoussoukro
rév 2015/12/13*

Version provisoire. Nous travaillons sur ce document, mais les remarques sont les bienvenues !

14.1 MOTIVATION

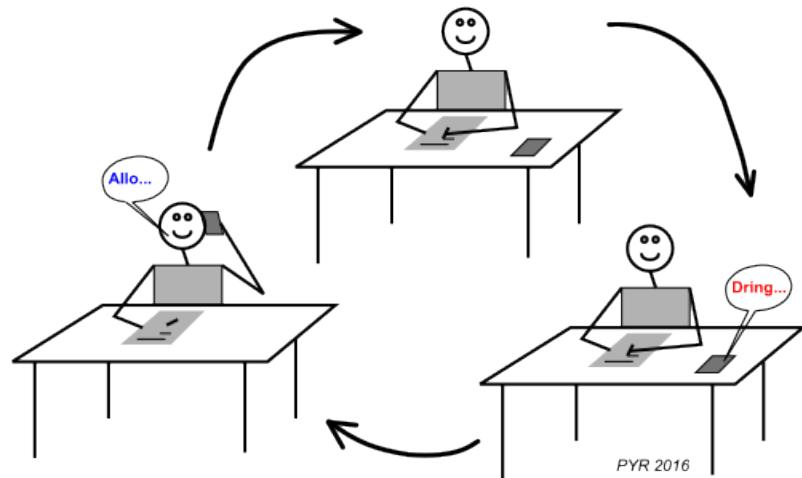
Un système à microcontrôleur est généralement pourvu d’entrées et de sorties. Le but premier du programme est souvent de réagir correctement aux changements d’état des entrées, en agissant en conséquence sur les sorties.

Les enseignes et afficheurs à LED sont plutôt une exception dans ce domaine. Beaucoup d’enseignes ou d’afficheurs n’ont aucune entrée et ne font que faire évoluer les sorties selon un ordre prédéfini.

Il existe toutefois des cas où une enseigne ou un afficheur doit réagir à des entrées. Par exemple, une télécommande peut être utilisée pour allumer et éteindre un afficheur, changer sa luminosité ou le texte qu’il doit afficher. Un autre cas où le système doit réagir à un événement est la gestion du temps dans un afficheur multiplexé : à des instants précis, il faut envoyer de nouvelles valeurs sur les LED.

14.2 DÉFINITION

On appelle **interruption** dans un système informatique l’arrêt temporaire d’un programme au profit d’un autre programme, jugé à cet instant plus important. L’interruption correspond au sens qu’on donne à ce mot dans nos vies courantes. Prenons un exemple : je suis en train de travailler à mon bureau. Le téléphone sonne. Je vais répondre au téléphone. Après la conversation, je reprends mon travail là où je l’avais laissé.



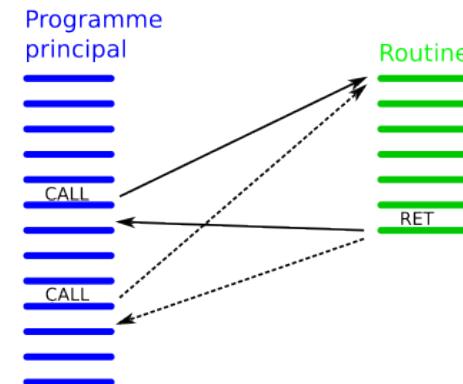
Interruption dans la vie courante

C'est toujours un *événement* qui va produire une interruption. Cet événement a un caractère imprévisible, le programme ne sait pas quand il va se produire.

14.3 IMPLÉMENTATION

Pour utiliser les interruptions, il n'est pas indispensable de comprendre en détail le mécanisme qui les rend possibles. C'est comme les procédures ou les fonctions, que nous avons l'habitude d'utiliser sans forcément connaître les mécanismes matériels qui les rendent possibles.

Toutefois, nous allons ici faire une petite incursion dans le monde de la programmation en assembleur, pour mieux comprendre les interruptions. La figure ci-dessous montre un programme, dont les instructions successives sont notées en bleu. Dans le cas où une fraction du programme doit s'exécuter plusieurs fois, on a l'habitude de grouper ses instructions, notées ici en vert. On appelle ce morceau de programme une *routine* ou *sous-routine*. Ce concept correspond aux procédures et aux fonctions dans les langages évolués comme le C.

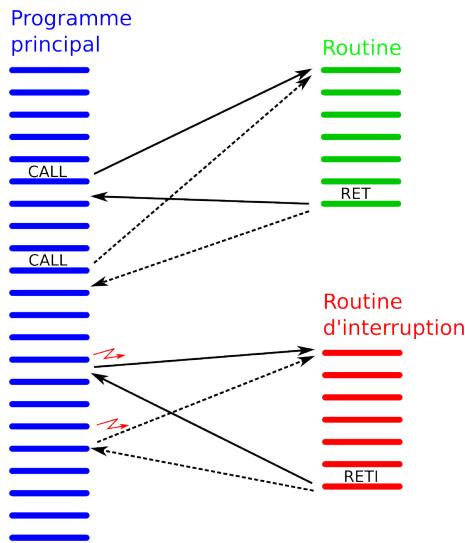


Appel d'une routine

L'appel de la routine se fait par une instruction **Call** dans le programme principal. À la fin de l'exécution de la routine, une instruction **Ret** (return = retour) permet de revenir au programme appelant, juste après l'instruction *Call*. La routine peut être appelée plusieurs fois dans le programme principal.

Notons que l'adresse de retour doit être mémorisée pour que le retour soit possible. C'est une **pile** (*stack*) en mémoire vive qui est utilisée à cette fin. Nous ne détaillerons pas son mécanisme ici.

Regardons maintenant la figure suivante. Un nouvelle routine, appelée **Routine d'interruption** est représentée en rouge. On voit qu'elle va aussi s'exécuter. Mais son exécution n'est pas la conséquence d'une instruction *Call*.



Principe des interruptions

Rien dans le programme principal ne permet de savoir que cette routine va s'exécuter. C'est un **événement** qui est la cause de son exécution. C'est ce qu'on appelle un interruption.

La routine d'interruption se termine aussi par une instruction de retour, appelée **Reti** (*return from interrupt*). En plus d'effectuer le retour au programme interrompu, elle rétablit le mode de sensibilité aux interruptions qui prévalait avant l'interruption.

14.4 NATURE DES ÉVÉNEMENTS

Quels sont ces événements qui vont produire une interruption ? Il en existe principalement deux sortes :

- Les événements **extérieurs** au microcontrôleur. Il s'agit par exemple d'un changement sur une entrée.
- Les événements **intérieurs** au microcontrôleur. Par exemple, beaucoup de microcontrôleurs sont pourvus d'un convertisseur analogique-numérique (*ADC = Analog to Digital Converter*). La conversion est déclenchée par un fanion et dure un certain temps. Plutôt que d'attendre la fin de la conversion,

DRAFT
2016-03-20
01:16:01

le programme principal peut continuer, puis être interrompu au moment de la fin de la conversion.

Dans cette catégorie des interruptions intérieures au microcontrôleur, les plus importantes sont celle liées aux **Timers**. Ce sujet sera abordé dans un chapitre séparé, vu son importance pour la commande des enseignes et afficheurs à LED.

14.5 DISCRIMINATION DES SOURCES D'INTERRUPTION

Il existe généralement plusieurs sources interruptions sur un microcontrôleur. Lorsqu'une interruption se produit, le système doit être capable d'en savoir la source. Si rien n'est prévu au niveau matériel, la routine d'interruption doit consulter les registres pour chaque interruption, pour connaître celle qui a été activée.

Les **vecteurs d'interruption** (*interrupt vectors*) permettent d'être plus efficace : une adresse différente est réservée pour le début de la routine de chaque interruption.

Souvent ces deux mécanismes vont être utilisés successivement, comme nous le verrons plus bas lors d'une interruption produite par une entrée sur un MSP430.

Voici la table résumée des vecteurs d'interruption pour un MSP430G, y compris l'adresse pour le Reset :

- 0xFFFFE : Reset
- 0xFFFFC : NMI
- 0xFFFFA : Timer1 CCR0
- 0xFFFF8 : Timer1 CCR1, CCR2, TAIFG
- 0xFFFF6 : Comparator_A
- 0xFFFF4 : Watchdog Timer
- 0xFFFF2 : Timer0 CCR0
- 0xFFFF0 : Timer0 CCR1, CCR2, TAIFG
- 0xFFEE : USCI status
- 0xFFEC : USCI receive/transmit
- 0xFFEA : ADC10
- 0xFFE8 : -
- 0xFFE6 : Port P2
- 0xFFE4 : Port P1

Les adresses se trouvent en mémoire flash, ce sont les dernières adresses de l'espace d'adressage de 16 bits.

DRAFT
2016-03-20
01:16:01

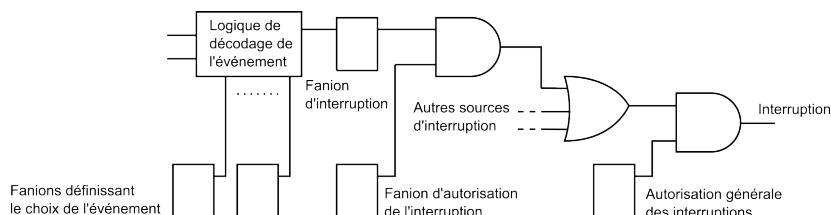
Plusieurs sources d'interruptions nécessitent la scrutation pour déterminer la cause exacte de l'interruption. C'est le cas par exemple des interruptions sur les ports : chaque bit peut produire une interruption. C'est aussi le cas d'une des interruptions des Timers : les registre de comparaison 1 et 2, ainsi que l'interruption générale du Timer sont regroupés sur un vecteur unique.

14.6 MISE EN ŒUVRE D'UNE INTERRUPTION

Plusieurs étapes sont nécessaire pour mettre en œuvre une routine d'interruption :

1. Enclencher l'interruption qui nous intéresse. Par exemple une interruption sur une entrée.
2. Préciser comment cette interruption doit fonctionner. Par exemple dire sur quel flanc l'interruption doit se produire.
3. Enclencher globalement les interruptions. Les microprocesseurs disposent d'un fanion général qui autorise les interruptions. Il est souvent utilisé pour pouvoir éviter les interruptions dans certaines parties critiques au programme, qui ne doivent pas être interrompues.

Le schéma logique ci-dessous montre la logique qui permet de générer les interruptions et les fanions qu'il faut ajuster.



Logique de génération des interruptions

On y trouve :

- la logique qui permet de saisir un événement
- les fanions qui règlent la manière dont l'événement est décodé
- le fanion qui enclenche cette interruption particulière
- la porte ET associée à ce fanion pour produire cette interruption
- la porte OU qui permet à toutes les interruptions d'être prises en compte

- le fanion général d'autorisation des interruptions
- la porte ET qui produit finalement les interruptions.

14.7 SYNTAXE DES ROUTINES D'INTERRUPTIONS

Le langage C ne définit pas la syntaxe des routines d'interruptions. Plusieurs notations sont utilisées, dépendant des compilateurs. Nous présenterons ici une des syntaxe supportée par les compilateurs GCC.

```
#pragma vector=NUMERO_DU_VECTEUR
__interrupt void Nom_de_la_routine (void) {
    ...
}
```

La première ligne indique au compilateur à quel vecteur d'interruption la routine sera associée. La seconde ligne est une déclaration de procédure presque habituelle, avec un nom et aucun paramètres d'entrée (`void`). L'indication `__interrupt` permet au compilateur d'utiliser les instructions correspondant à une routine d'interruption, en particulier le *Reti* final.

14.8 INTERRUPTION PRODUITE PAR UNE ENTRÉE

Sur les microcontrôleurs MSP430, plusieurs registres permettent de définir la manière dont une broche d'entrée-sortie est utilisée. Ils sont associés à un Port, composé de 8 broches. Sur le MSP430G2553 du Launchpad, deux ports sont disponibles : P1 et P2. On connaît déjà les registres suivant :

- **P1DIR** : détermine le rôle de la broche (entrée ou sortie)
 - **P1OUT** : donne la valeur pour les broches de sortie
 - **P1IN** : permet de lire la valeur des entrées
 - **P1REN** : permet d'enclencher une résistance de tirage (pull-up ou pull-down, selon l'état de bit de P1OUT correspondant)
- Pour mettre en œuvre les interruptions sur des broches du port P1, trois registres supplémentaires sont disponibles :
- **P1IE** : (*Interrupt Enable*) permet l'enclenchement de l'interruption pour chaque bit. L'usage habituel est d'écrire dans ce registre pour choisir quels bits vont causer une interruption.

- P1IES** : (*Interrupt Edge Select*) permet de choisir pour chaque bit le flanc qui va produire l'interruption. Lorsque le bit est à 0, l'interruption va se produire lors d'une transition de 0 vers 1 (*low-to-high transition*). L'usage habituel est d'écrire dans ce registre pour choisir quel flanc va causer une interruption, pour chaque bit.
- P1IFG** : (*Interrupt FlaG*) les **fanions d'interruption**. Lorsque qu'un transition telle qu'elle est spécifiée dans un bit de P1E, le bit correspondant s'active dans P1IFG. C'est son activation qui produit l'interruption elle-même.

Voici un programme qui met en œuvre une interruption sur l'entrée P1.3 (le poussoir du Launchpad) et qui change d'état P1.6 (la LED verte) à chaque flanc descendant.

```

1 int main() {
2     WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
3     P1DIR |= (1<<6); // Led verte en sortie
4     P1OUT |= (1<<3); P1REN |= (1<<3); //pull-up sur l'entrée P1.3
5
6     P1IES |= (1<<3); // Mode d'interruption : sur le flanc descendant
7     P1IE |= (1<<3); // Interruption P1 activée sur le bit 3
8     P1IFG &=~(1<<3); // Fanion d'interruption remis à zéro
9     __enable_interrupt(); // General Interrupt Enable
10
11    while(1) { // il n'y a rien à faire dans la boucle principale !
12    }
13}
14
15 // Routine d'interruption associée au Port P1
16 // Syntaxe spécifique pour les interruptions :
17 #pragma vector=PORT1_VECTOR
18 __interrupt void Port1_ISR(void) {
19     P1IFG &= ~(1<<3); // Fanion d'interruption correspondant au bit 3 remis à 0
20     P1OUT ^= (1<<6); // inverse P1.6 (LED verte)
21 }
```

La première remarque, c'est que la boucle principale `while(1)...` ne fait rien ! En plus des initialisations classique des entrées et des sorties, trois instructions ont été ajoutées, correspondant aux étapes de mise en œuvre d'une interruption :

- L'activation d'un bit dans P1IES sélectionne le flanc descendant
- L'activation d'un bit dans P1IE autorise l'interruption sur l'entrée P1.3

- L'appel de la procédure `__enable_interrupt()` autorise globalement les interruptions sélectionnées

La mise à 0 du bit 3 dans le registre des fanions d'interruption P1IFG évite qu'un flanc sur l'entrée P1.3 pouvant s'être produit avant l'autorisation des interruptions ne cause une interruption.

La routine d'interruption a été placée à la suite du programme principal, alors que nous avons l'habitude de placer les procédures avant le programme principal. Cette procédure n'est en effet jamais appelée explicitement.

14.9 SCRUTATION DU BIT CONCERNÉ

Dans notre exemple, seul le bit 3 est concerné par les interruptions. Dans la routine d'interruption, il n'y a donc pas besoin de regarder quel bit a produit l'interruption et c'est systématiquement le fanion 3 qui est remis à 0.

Lorsque l'interruption peut provenir de plusieurs entrées, il est alors nécessaire de scruter le registre pour connaître le bit qui a causé l'interruption, comme le montre cet exemple :

```

1 int main() {
2 ...
3     P1IES &=~((1<<3)|(1<<4)); // interruptions aux flancs montants
4     P1IE |= (1<<3)|(1<<4); // Interruption activée sur 2 entrées
5     P1IFG &=~((1<<3)|(1<<4)); // Fanions d'interruption remis à zéro
6 ...
7
8 #pragma vector=PORT1_VECTOR
9 __interrupt void Port1_ISR(void) {
10     // scrutation des causes possibles de l'interruption :
11     if (P1IFG & (1<<3)) {... ; P1IFG &= ~(1<<3);}
12     if (P1IFG & (1<<4)) {... ; P1IFG &= ~(1<<4);}
13 }
```

14.10 INTERRUPTION DE FIN DE CONVERSION

Voici un autre exemple d'interruption, où l'événement est interne au microcontrôleur. L'interruption doit se produire lorsque le convertisseur Analogique-Numérique (ADC) termine sa conversion.

```

1 int main() {
2     WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
3     P1DIR |= (1<<6); P1OUT &=~(1<<6); // LED verte en sortie
4     // Activation du convertisseur ADC 10 bits (ADC10) :
5     ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE; // ADC10ON, interrupt enabled
6     ADC10CTL1 = INCH_1; // Canal 1 = entrée A1 = P1.1
7     ADC10AE0 |= (1<<1); // Enclanchement de l'entrée A1
8     __enable_interrupt(); // General Interrupt Enable
9     ADC10CTL0 |= ENC + ADC10SC; // lance une première conversion
10
11    while(1) { // il n'y a rien à faire dans la boucle principale !
12    }
13}
14
15 // Routine d'interruption associée à la fin de conversion ADC
16 #pragma vector=ADC10_VECTOR
17 __interrupt void ADC10_ISR(void) {
18     uint16_t val = ADC10MEM; // lit le résultat de la conversion
19     ADC10CTL0 |= ENC + ADC10SC; // lance la conversion suivante
20     if (val > 512) { // Montre sur la LED verte si la valeur dépasse Vcc/2
21         P1OUT |= (1<<6); // LED verte On
22     } else {
23         P1OUT &=~(1<<6); // LED verte Off
24     }
25}

```

CHAPITRE 15

LES TIMERS

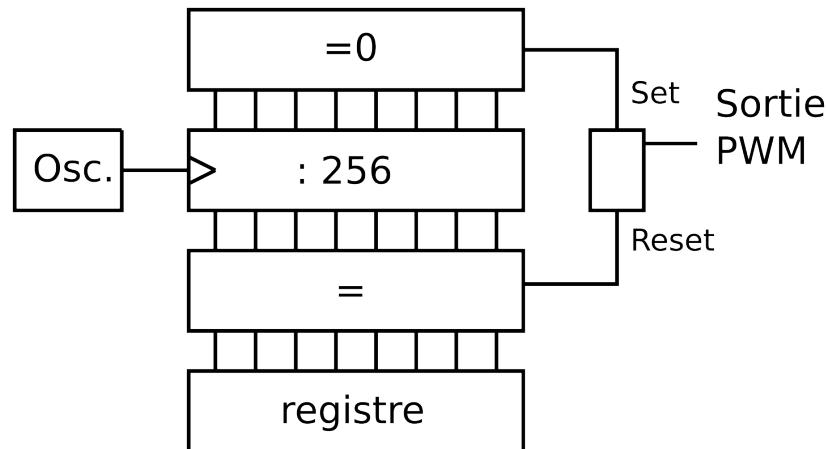
Pierre-Yves Rochat, EPFL
et Yves Tiecoura, INP-HB
rév 2016/02/02

15.1 GESTION EXACTE DU TEMPS

Les enseignes et afficheurs à LED, comme beaucoup d'applications des microcontrôleurs, nécessitent souvent une gestion exacte du temps. Les animations doivent être correctement cadencées et, plus difficile encore, la gestion des afficheurs matriciels multiplexés exige une gestion du temps (*timing*) exacte.

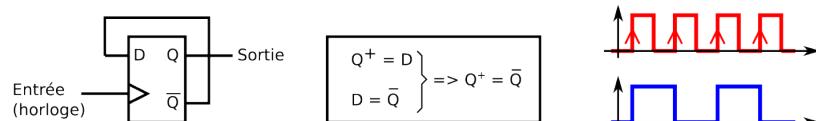
Il est souvent difficile d'assurer correctement cette gestion du temps en utilisant uniquement les instructions du processeur. C'est la raison pour laquelle les microcontrôleurs offrent presque toujours des circuits spécialisés dans le comptage et la gestion du temps, appelés les *timers*.

Dans le chapitre sur la modulation de largeur d'impulsion (PWM), nous avions proposé le montage suivant, pour faciliter la génération de signaux PWM :



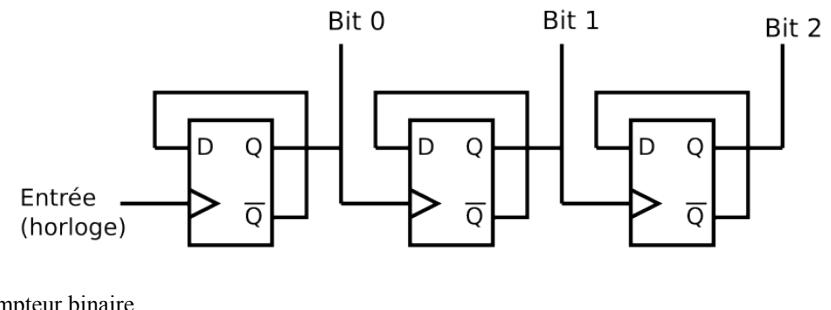
Compteur générant du PWM

Ce montage est basé sur un compteur binaire, qu'on appelle aussi un *diviseur de fréquence*. Rappelons qu'à chaque flanc montant de l'horloge, le compteur passe à la valeur binaire suivante.

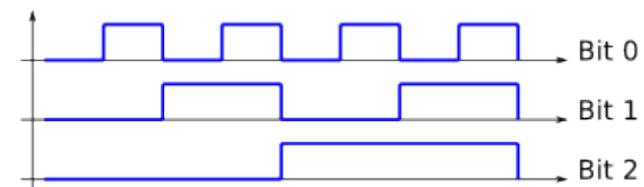


Diviseur par 2

En ajoutant bout à bout plusieurs diviseurs par 2, on obtient un compteur binaire :



On peut observer que lorsqu'un signal de fréquence fixe F_0 est placé sur l'entrée, les sorties successives prennent des fréquences sous-multiples : la fréquence est divisée par 2, par 4, par 8, etc.



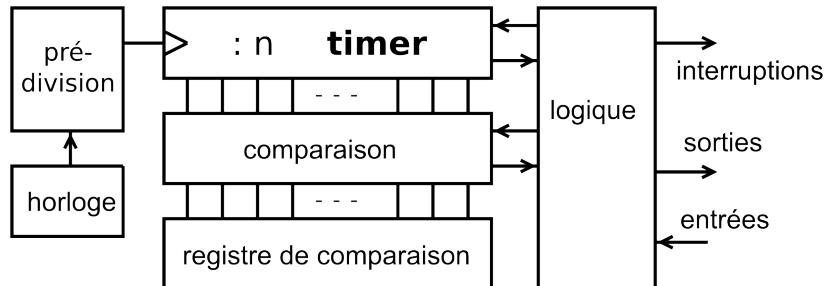
Chronogramme d'un compteur binaire

Le terme anglais *timer* désigne non seulement le compteur binaire, mais aussi souvent l'ensemble du montage. Les traductions françaises, *minuterie* ou *temporisateur*, ne sont que rarement utilisées. C'est la raison pour laquelle nous utiliserons ici plutôt l'anglicisme *timer*, que nous considérerons comme un néologisme.

Le PWM n'est pas la seule application des timers. Beaucoup de tâches — liées le plus souvent à la gestion du temps ou au comptage d'événements — peuvent lui être confiées.

15.2 LES TIMERS

La figure ci-dessous généralise ce concept :



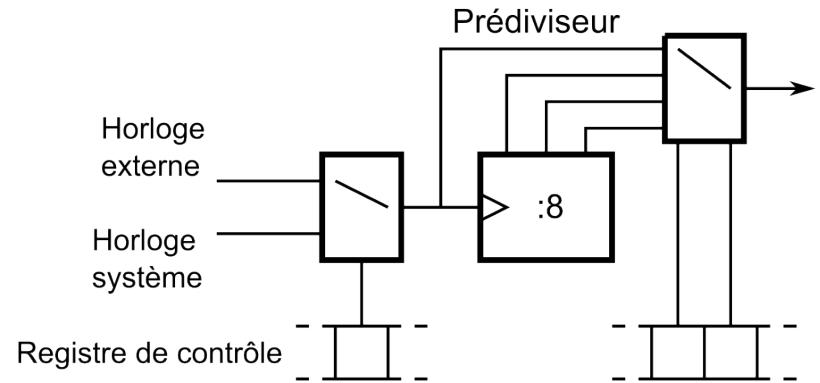
Timer

On y trouve :

- Un **compteur binaire**. Il peut être de 8 bits, 16 bits, parfois même de 32 bits. C'est le timer proprement dit.
- Une **horloge**, c'est-à-dire un oscillateur (OSC). Il s'agit généralement de l'horloge également utilisée pour le processeur.
- Un système de **choix de l'horloge et du prédiviseur**, qui permet de choisir une fréquence d'horloge bien adaptée au problème à résoudre.
- Une logique de **comparaison**, par exemple pour tester l'égalité.
- Un **registre de comparaison**, associé à la logique de comparaison. Plusieurs registres de comparaison sont souvent présents.
- Une logique de gestion, permettant de faire interagir des **entrées** et des **sorties** avec le timer, ainsi qu'à générer des **interruptions** dans certaines conditions.

15.3 PRÉDIVISION

Voici comment peut se présenter le choix de l'horloge et du prédiviseur :



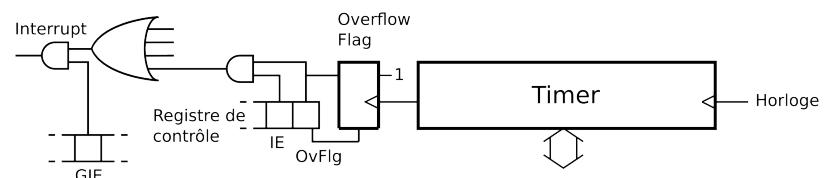
Exemple de système de choix de l'horloge

Un premier multiplexeur permet de choisir entre une horloge interne ou externe. Un compteur binaire, utilisé en diviseur de fréquence, fournit des signaux à des fréquences sous-multiples de celle de l'horloge. Un second multiplexeur permet de choisir la fréquence qui commande le timer.

Les deux multiplexeurs sont commandés par des bits d'un registre de contrôle, dont le rôle est de fixer le mode de fonctionnement du timer.

15.4 LOGIQUE DE GESTION

Une logique permet de mettre en œuvre le timer. Elle diffère beaucoup d'un microcontrôleur à l'autre. En voici un exemple très simple :



Exemple de logique de gestion d'un timer

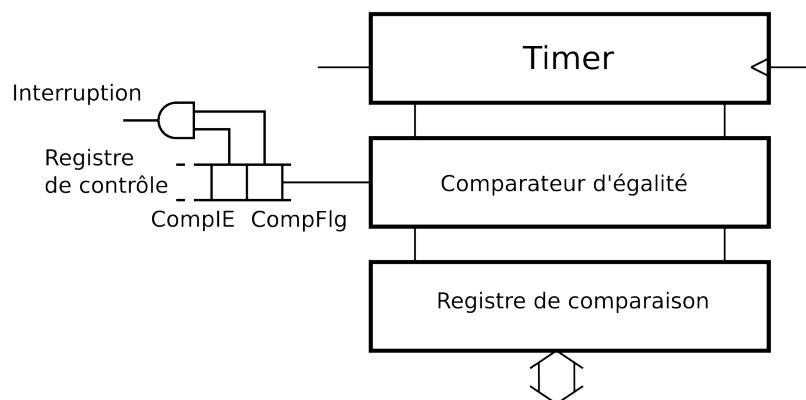
On y trouve une bascule qui détecte le dépassement de capacité du timer. C'est le moment où le compteur binaire repasse à la valeur 0. La bascule est mise à 1 à cet instant. Elle fait généralement partie d'un registre de contrôle et peut donc être lue en tout temps.

Il faut pouvoir remettre ce fanion à zéro lors qu'il a été pris en compte. Parfois, il faut écrire un 0 dans le bit correspondant du registre. Mais sur certains microcontrôleurs, c'est l'écriture de la valeur 1 qui met ce fanion à 0. C'est le cas des timers des AVR.

La génération d'interruptions est très importante dans l'utilisation des timers. Ici, on voit un fanion IE (*Interrupt Enable*) qui permet de générer une interruption. En effet, la porte logique ET nécessite qu'IE soit à 1 pour que l'interruption soit transmise. Elle ne sera effective que si l'autorisation générale des interruptions est activée (GIE ou *General Interrupt Enable*), comme toutes les autres interruptions.

15.5 REGISTRES DE COMPARAISON

La présence d'un ou de plusieurs registres de comparaison associés à un timer le rend beaucoup plus intéressant. En voici un exemple simple :



Exemple de registre de comparaison

Un comparateur d'égalité est placé entre le timer et un registre dont il est possible à tout moment de modifier la valeur. Chaque fois que le timer a la même valeur que le registre de comparaison, le fanion passe à 1. À nouveau, il est possible de générer une interruption, avec un mécanisme similaire à celui du dépassement de capacité.

15.6 LES TIMERS DES MICROCONTRÔLEURS

Quelques années après l'apparition des premiers microprocesseurs, des circuits spécialisés incorporants des timers sont apparus sur le marché. C'est le cas du très célèbre 8253 d'Intel, datant de 1981, dont on trouve encore des descendants dans les PC modernes.

Les microcontrôleurs ont eux aussi très vite été complétés par des timers, comme le célèbre PIC16x84, qui incluait déjà un unique compteur 8 bits très simple, mais très utile.

Les microcontrôleurs ARM ont tous plusieurs timers. L'ATmega328, connu pour équiper les Arduino, a trois timers, le TIMER 0 de 8 bits, le TIMER 1 de 16 bits et le TIMER 2 de 8 bits, mais différent du TIMER 0. Ces timers sont riches en fonctionnalités permettant de nombreuses applications.

Les microcontrôleurs plus modernes ont souvent des timers très complexes. Dans les familles de microcontrôleurs ARM, les timers diffèrent d'un fabricant à l'autre : cette partie du microcontrôleur est propriétaire, elle n'est pas développée par la société ARM.

Nous étudierons ici les timers utilisés dans les microcontrôleurs MSP430G de Texas Instruments, qui se trouvent sur la carte Launchpad.

15.7 TIMER A DU MSP430

Les MPS430 de la série G disposent de timers de 16 bits, en nombre et en configurations variables selon les modèles. Le MSP430G2231 avec un boîtier de 14 pattes en a un seul, disposant de deux registres de comparaison. Le MSP430G2553 en a deux, disposant chacun de trois registres de comparaison.

Le fonctionnement de ces registres est très bien documenté : 20 pages, bien évidemment en anglais. Voici les références du document : *MSP430x2xx Family User's Guide, literature Number: SLAU144H*. On le trouve facilement sur internet.

Afin de nous familiariser avec la lecture de la documentation, nous allons nous baser sur les documents fournis par Texas Instruments pour comprendre le minimum nécessaire à la mise en œuvre d'un de ces timers. Nous allons aussi respecter la syntaxe proposée pour l'accès aux registres.

La figure ci-dessous donne la vue d'ensemble du TIMER A :

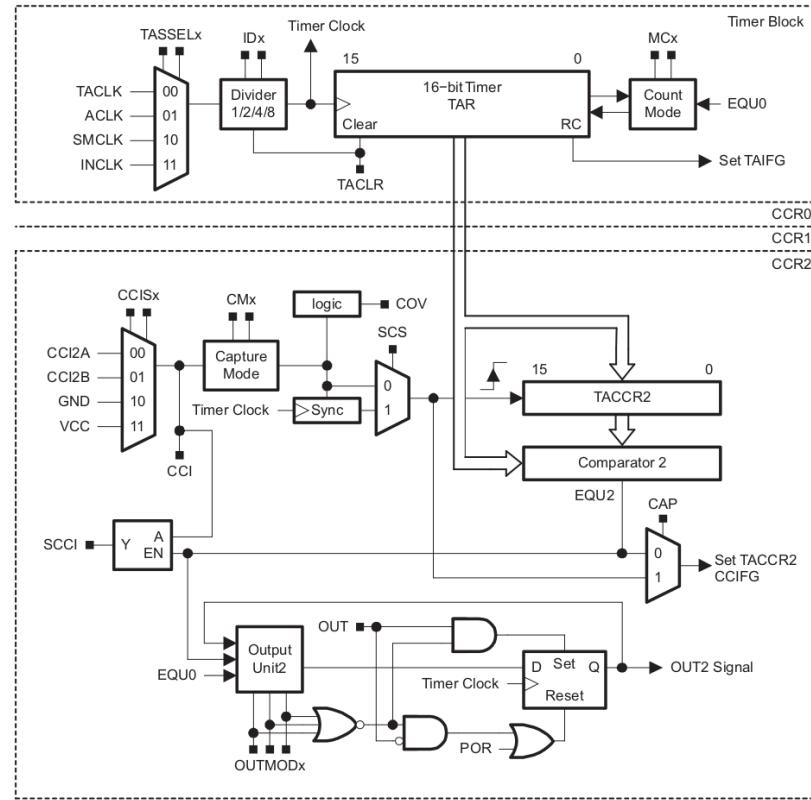


Figure 12-1. Timer_A Block Diagram

TIMER A du MSP430

Ce schéma n'est pas simple, mais il est clair et complet. On y trouve un compteur 16 bits appelé TAR. Il est possible à tout moment de lire sa valeur. Il est aussi possible d'écrire une nouvelle valeur, mais nous n'utiliserons pas cette possibilité ici.

Ce compteur reçoit un signal d'horloge qu'il est possible de sélectionner parmi plusieurs sources. Un prédiviseur peut être utilisé, qui donne le choix entre la fréquence d'origine et des divisions par 2, 4 ou 8. Le compteur peut compter selon plusieurs modes.

Un registre de contrôle de 16 bits appelé TACTL est associé à chaque timer. Il peut aussi apparaître sous le nom TA0CTL, pour les microcontrôleurs qui ont

plusieurs TIMER A (le deuxième s'appelant alors TA1CTL). Il n'apparaît pas explicitement dans le schéma, mais c'est de lui que proviennent plusieurs signaux (TASSELx, IDx, TACLR, etc.) Ce sont les différents bits de ce registre qui vont permettre de choisir l'horloge, les prédiviseurs, le mode de comptage, etc.

Voici comment la documentation le décrit ce registre TACTL :

12.3.1 TACTL, Timer_A Control Register

	15	14	13	12	11	10	9	8
	Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	TAIE	TAIFG
IDx	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
Unused	Bits 15-10	Unused						
TASSELx	Bits 9-8	Timer_A clock source select						
	00	TACLK						
	01	ACLK						
	10	SMCLK						
	11	INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)						
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock.						
	00	/1						
	01	/2						
	10	/4						
	11	/8						
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.						
	00	Stop mode: the timer is halted.						
	01	Up mode: the timer counts up to TACCR0.						
	10	Continuous mode: the timer counts up to 0xFFFFh.						
	11	Up/down mode: the timer counts up to TACCR0 then down to 0000h.						
Unused	Bit 3	Unused						
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.						
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request.						
TAIFG	Bit 0	Timer_A interrupt flag						
	0	No interrupt pending						
	1	Interrupt pending						

Registre TACLT

Parcourons quelques bits de ce registre de contrôle pour choisir les valeurs pour notre premier exemple :

- TASSELx permet de choisir l'horloge. Utilisons l'horloge du processeur : SMCLK. Les deux bits correspondants doivent prendre la valeur binaire 10. Texas Instruments utilise la syntaxe suivante : TASSEL_2 (valeur 2 pour les bits TASSEL).
- IDx permet de choisir la prédivision. Choisissons une division par 8. La valeur est ID_3.
- MCx permet de choisir le mode de comptage. Choisissons le mode continu. La valeur est MC_2.

L'instruction d'initialisation de notre timer sera donc : `TACTL = TASSEL_2 + ID_3 + MC_2;`

15.8 PREMIER PROGRAMME AVEC LE TIMER A

Voilà un premier programme... qui va faire clignoter une LED !

Il commence comme toujours par l'instruction de mise hors service du comparateur *watchdog*, mais aussi par deux instructions permettant de choisir une des fréquences calibrées d'usine, ici 1 MHz :

```
int main() {
    WDTCTL = WDTPW + WDTHOLD; // Watchdog hors service
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ; // Fréquence CPU
    P1DIR |= (1<<0); // P1.0 en sortie pour la LED
    TACTL0 = TASSEL_2 + ID_3 + MC_2;
    while (1) { // Boucle infinie
        if (TACTL0 & TAIFG) {
            TACTL0 &= ~TAIFG;
            P1OUT ^= (1<<0); // Inversion LED
        }
    }
}
```

Comment fonctionne la boucle principale ? Chaque fois que le fanion TAIFG passe à 1, l'alimentation de la LED est inversée. Le fanion TAIF (qui se trouve aussi dans le registre TACTL) signale un dépassement de capacité, c'est-à-dire le retour à zéro du compteur. Il doit être remis à zéro en vue du prochain cycle. Calculons la période de clignotement : l'horloge de 1 MHz est divisée par 8 par le prédiviseur. Le timer est donc commandé à une fréquence de 125 kHz ce qui correspond à une période de 8 µs. Le timer a 16 bits, il va donc faire un cycle complet en 65'536 coups d'horloge, soit environ 524 ms.

15.9 LES REGISTRES DE COMPARAISON

L'intérêt principal des timers réside dans les registres de comparaison qui leur sont associés. Dans le schéma de la page 1, on voit qu'il y a trois registres de comparaison, notés 0, 1 et 2. Le détail est donné pour le groupe 2.

Ces trois registres de comparaison se nomment CCR0, CCR1 et CCR2. Ces registres permettent de mémoriser une valeur qui va être en permanence comparée avec la valeur du timer TAR.

À chacun de ces registres de comparaison est associé un registre de contrôle, appelé respectivement TACCLT0, TACCLT1 et TACCTL2.

La figure suivante donne la description de ce registre. Elle n'est pas simple :

Timer_A Registers								www.ti.com
12.3.4 TACCTLx, Capture/Compare Control Register								
15	14	13	12	11	10	9	8	
CMx		CCISx		SCS	SCCI	Unused	CAP	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)	
7	6	5	4	3	2	1	0	
OUTMODx		CCIE		CCI	OUT	COV	CCIFG	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)	
CMx	Bit 15-14	Capture mode						
		00 No capture						
		01 Capture on rising edge						
		10 Capture on falling edge						
		11 Capture on both rising and falling edges						
CCISx	Bit 13-12	Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections.						
		00 CClxA						
		01 CClxB						
		10 GND						
		11 V _{cc}						
SCS	Bit 11	Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock.						
		0 Asynchronous capture						
		1 Synchronous capture						
SCCI	Bit 10	Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit						
Unused	Bit 9	Unused. Read only. Always read as 0.						
CAP	Bit 8	Capture mode						
		0 Compare mode						
		1 Capture mode						
OUTMODx	Bits 7-5	Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0, because EQUx = EQU0.						
		000 OUT bit value						
		001 Set						
		010 Toggle/reset						
		011 Set/reset						
		100 Toggle						
		101 Reset						
		110 Toggle/set						
		111 Reset/set						
CCIE	Bit 4	Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.						
		0 Interrupt disabled						
		1 Interrupt enabled						
CCI	Bit 3	Capture/compare input. The selected input signal can be read by this bit						
OUT	Bit 2	Output. For output mode 0, this bit directly controls the state of the output.						
		0 Output low						
		1 Output high						
COV	Bit 1	Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software.						
		0 No capture overflow occurred						
		1 Capture overflow occurred						
CCIFG	Bit 0	Capture/compare interrupt flag						
		0 No interrupt pending						
		1 Interrupt pending						

Registre TACCRx

Modifions notre programme de la manière suivante :

```

int main() {
...
TACCR0 = 62500; // 62500 * 8 us = 500 ms
while (1) { // Boucle infinie
    if (TACCTL0 & CCIFG) {
        TACCTL0 &= ~CCIFG;
        TACCR0 += 62500;
        P1OUT ^= (1<<0); // Inversion LED
    }
}
}

```

Au début du programme, le registre de comparaison a été initialisé à 62'500, une valeur qui correspond à une demi-seconde dans notre cas : $62'500 \times 8 \mu\text{s} = 500 \text{ ms}$. Une fois cette valeur atteinte, il faut ajouter 62'500 à la valeur courant du registre de comparaison. On va dépasser la capacité du registre, qui a 16 bits. On obtiendra : $(62'500 + 62'500) \% 65'536 = 59'464$ où le signe % représente l'opération de modulo, c'est-à-dire le reste de la division entière. Mais comme le timer augmente toujours et qu'il a lui aussi 16 bits, cette valeur est effectivement la bonne pour la prochaine comparaison.

Si vous avez des doutes, imaginez qu'il est 9 h 50 et que vous voulez faire sonner votre réveil dans 30 minutes. Vous devez le régler sur 10 h 20. En ne tenant compte que des minutes, on a bien : $(50 + 30) \% 60 = 20$.

15.10 LES INTERRUPTIONS ASSOCIÉES AUX TIMERS

L'intérêt principal des timers est de les associer à des interruptions. Modifions le programme de la manière suivante :

```

int main() {
...
TACTL |= TAIE; // Interruption de l'overflow
_BIS_SR (GIE); // Autorisation générale des interruptions
while (1) { // Boucle infinie vide
}

// Timer_A1 Interrupt Vector (TAIV) handler
#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_A1 (void) {
    switch (TAIV) { // discrimination des sources d'interruption
        case 2: // CCR1 : not used
}
}

```

DRAFT
2016-03-20
01:16:01

```

break;
case 4: // CCR2 : not used
break;
case 10: // Overflow
P1OUT ^= (1<<0); // Inversion LED
break;
}
}

```

Notez le nom de la routine d'interruption. Elle ne concerne pas le TIMER 1 ! Elle est la seconde routine d'interruption du TIMER 0, la première étant présentée dans le prochain exemple.

L'interruption associée au timer lui-même correspond à un *overflow* (dépassement de capacité, c'est le passage de la plus grande valeur à 0). La syntaxe de la routine d'interruption est un peu compliquée. Il faut la copier et non pas chercher à la comprendre ! Notez qu'elle varie selon les compilateurs : il ne s'agit pas d'une norme du C. Dans ce cas, trois sources différentes d'interruption (OVERFLOW, COMPARAISON 1 et COMPARAISON 2) sont regroupées dans une même routine d'interruption. Un registre appelé TAIV permet de connaître dans chaque cas la cause de l'interruption. Les valeurs 2, 4 et 10 sont le choix arbitraire du fabricant : il faut respecter scrupuleusement la syntaxe des instructions `switch TAIV... case...`. Il n'a pas été nécessaire de remettre à zéro le fanion TAIFG, car c'est la gestion matérielle des interruptions qui le fait automatiquement au moment de l'appel de la routine d'interruption.

15.11 INTERRUPTION DE COMPARAISON

De même, une interruption peut être associée à chaque registre de comparaison. Cette fois, c'est dans le registre TACCTLx (x valant 0, 1 ou 2) qu'il faut activer le fanion d'interruption.

```

int main() {
...
TACCTL0 |= CCIE; // Interruption de la comparaison
_BIS_SR (GIE); // Autorisation générale des interruptions
while (1) { // Boucle infinie vide
}

#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0 (void) {
    CCR0 += 62500;
}

```

DRAFT
2016-03-20
01:16:01

```
P1OUT ^= (1<<0); // Inversion LED
}
```

15.12 PWM PAR INTERRUPTION

En combinant les interruptions du dépassement de capacité et de la comparaison, on peut produire un signal PWM sur n'importe quelle broche du microcontrôleur :

```
int main() {
    ...
    TACTL |= TAIE; // Interruption de l'overflow
    TACCTL0 |= CCIE; // Interruption de la comparaison
    _BIS_SR(GIE); // Autorisation générale des interruptions
    while (1) { // Boucle infinie vide
    }

#pragma vector=TIMER0_A1_VECTOR
_interrupt void Timer_A1 (void) {
    switch (TAIV) { // discrimination des sources d'interruption
        case 2: // CCR1 : not used
            break;
        case 4: // CCR2 : not used
            break;
        case 10: // Overflow
            P1OUT |= (1<<0); // Activer le signal au début du cycle
            break;
    }
}

#pragma vector=TIMER0_A0_VECTOR
_interrupt void Timer_A0 (void) {
    P1OUT &= ~(1<<0); // Désactiver le signal au moment donné par le registre de comparaison
}
```

Les timers offrent de très nombreuses possibilités. L'étude détaillée de la documentation peut prendre du temps. De nombreux exemples sont fournis par les fabricants pour en illustrer les divers usages.

CHAPITRE 16

HORLOGES TEMPS RÉEL

Pierre-Yves Rochat, EPFL
rév 2015/09/18

16.1 AFFICHER L'HEURE

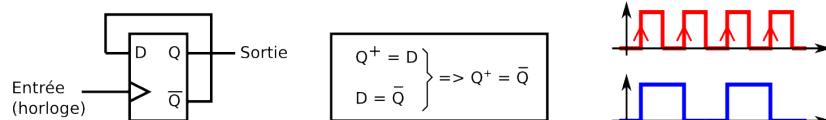
Les LED se prêtent très bien à afficher l'heure. Que ce soit pour de petits réveils ou pour des horloges géantes, l'heure est souvent affichée au moyen de LED. Une zone avec une horloge peut très bien être ajoutée à une enseigne à motifs fixes pour la rendre plus attrayante. Un journal lumineux peut compléter les informations qu'il affiche en donnant de temps en temps l'heure et la date.

Encore faut-il que l'heure indiquée soit juste ! Rien de plus désagréable qu'une horloge affichant une heure fausse... Il faut donc disposer d'un moyen de connaître l'heure de manière fiable.

16.2 BASE DE TEMPS

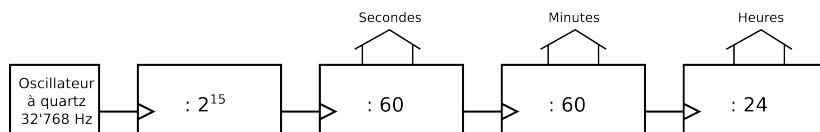
Une montre mécanique est réalisée sur la base d'un mouvement oscillant, dont la fréquence est aussi stable que possible. Une chaîne de diviseurs mécaniques, basés sur des engrenages, permet ensuite d'afficher les secondes, les minutes et les heures au moyen d'aiguilles.

Une montre électronique utilise un principe similaire. La différence est que le système oscillant est un cristal de quartz. La fréquence la plus couramment utilisée est 32'768 Hz. Pourquoi cette valeur ? Parce que c'est une puissance de 2, très exactement 2 à la puissance 15. Il va donc suffire de diviser la fréquence produite par l'oscillateur à quartz par une chaîne de 15 diviseurs par 2 pour obtenir un signal de 1 Hz. Un diviseur par deux, appelé aussi compteur binaire, peut être réalisé avec une bascule, comme le montrent le schéma, les équations et le diagramme des temps de la figure suivante :



Diviseur par 2

Ensuite, des diviseurs successifs vont produire les secondes, les minutes, les heures, etc, comme l'indique la figure ci-dessous :



Principe d'une horloge électronique

Les circuits logiques qui composent une horloge électronique s'appellent souvent Real Time Clock, abrégé RTC (Horloge en Temps Réel).

16.3 PILE DE SECOURS

Il est presque impossible d'assurer qu'un dispositif soit en permanence connecté à un réseau électrique fonctionnel. Dans beaucoup de pays du monde, il est illusoire de vouloir compter sur un approvisionnement électrique sans pannes. De plus, certains dispositifs doivent être parfois déplacés. Il faut donc une source d'énergie de secours, qui assure en permanence le fonctionnement de l'oscillateur à quartz et de la chaîne de division, pour ne pas perdre l'heure. Il est possible d'utiliser pour cela une petite pile.

Anecdote : On trouve une telle pile dans tous les ordinateurs. Elle est souvent désignée par l'expression *pile C-MOS*. Est-ce à dire qu'une pile est fabriquée en technologie C-MOS ? Evidemment non ! Cette expression date du temps où les premiers circuits C-MOS étaient utilisés pour réaliser des horloges, qu'il était alors possible de faire fonctionner en permanence grâce à une pile de taille modeste, vu que la technologie C-MOS consomme un minimum de courant. Aujourd'hui, tous les microprocesseurs et microcontrôleurs sont basés sur la technologie C-MOS. Mais l'expression *pile C-MOS* est encore usitée de nos jours.

Les piles utilisées pour maintenir l'heure sont souvent des piles au lithium. Pourquoi les piles rechargeables ne sont-elles que peu utilisées dans ce domaine ? La raison est la suivante : la capacité d'une pile au lithium est suffisante pour

maintenir une horloge temps réel pendant environ 10 ans. C'est aussi l'ordre de grandeur de la durée de vie de cette pile. Or les piles rechargeables ont une durée de vie généralement plus faible, elle ne sont donc pas particulièrement intéressantes pour cette application.

16.4 SUPERCAP

Il existe un autre système d'accumulation d'énergie, qui peut être utilisé pour des horloges temps réel. Ce sont les *supercap*. Il s'agit de condensateurs électrolytiques basés sur une technologie à double couche électrochimique. Il est courant d'obtenir des capacités de plusieurs Farad, pour une taille similaire à une pile au lithium.

La capacité de la supercap est généralement dimensionnée pour assurer à l'horloge une autonomie de quelques jours ou quelques semaines. Dès que l'appareil est relié au réseau électrique, la supercap est rapidement rechargée. Le nombre de cycles charge-décharge peut être très grand, de l'ordre de 100'000 cycles. Pour comparaison, la durée de vie d'un accumulateur est limité à environ 1'000 cycles. Mais celle d'un condensateur traditionnel, même électrolytique est encore bien supérieure.

16.5 PROGRAMMATION D'UNE HORLOGE AVEC UN MICROCONTRÔLEUR

La programmation d'une horloge temps réel va beaucoup dépendre du microcontrôleur utilisé pour ce qui concerne la production d'un événement toutes les secondes. Voici un exemple applicable à un ATmega328 :

```

1 #include <avr/io.h>
2 #include <avr/interrupt.h>
3 #include <avr/sleep.h>
4
5 volatile uint8_t secondes;
6
7 // Il faut un quartz 32 khz sur les broches TOSC1 et TOSC2
8 ISR (TIMER2_OVF_vect) {
9     secondes++;
10    ...
11 }
12
13 int main () {

```

```

14 Temp=0;
15 ASSR=(1<<AS2); // oscillateur quartz 32 khz
16 TCCR2B=0b101; // prédivision par 128
17 TIMSK2=(1<<TOIE2); // interruption Timer2 Overflow autorisée
18 sei(); // toutes les interruptions autorisées
19 while (1) { //boucle correspondant à tous les réveils dus à l'interruption
20     set_sleep_mode(SLEEP_MODE_PWR_SAVE); // mise en veille
21     sleep_enable();
22     sleep_mode(); // mode normal après un réveil
23     sleep_disable();
24 }
25 }
```

Un quartz va être branché sur les broches prévues à cet effet. L'oscillateur correspondant va être mis en œuvre à l'intérieur du microcontrôleur. Son signal de sortie va actionner directement le Timer2, à travers un pré-diviseur de 128 (7 bits). Le Timer2 est un timer 8 bits, ce qui correspond à un diviseur par 256. Or 128×256 est justement égal à 32'768 (15 bits). En autorisant une interruption sur le dépassement de capacité (Overflow) du Timer2, on obtient bien une interruption à chaque seconde.

Au moment de l'interruption, le microcontrôleur va se remettre à fonctionner. Dans la routine d'interruption, il va prendre en compte dans une variable le passage à la seconde suivante. Cette variable sera conservée en mémoire vive : le bon mode de sommeil du microcontrôleur doit être choisi pour cela. Le microcontrôleur va ensuite repasser en mode sommeil (sleep), pour minimiser sa consommation.

La partie qui compte les secondes, les minutes et les heures est par contre applicable à tous microcontrôleurs :

```

8 ...
9 secondes++;
10 if (secondes == 60) {
11     secondes = 0;
12     minutes++;
13     if (minutes == 60) {
14         minutes = 0;
15         heures++;
16     if (heures == 24) {
17         heures = 0;
18         ...
```

```

19     }
20 }
21 ...
```

16.6 CIRCUITS RTC SPÉCIALISÉS

Pour décharger le microcontrôleur de la tâche de maintenir l'horloge temps réel, mais surtout pour éviter de devoir maintenir en fonctionnement le microcontrôleur au moyen d'une pile, on utilise souvent des circuits intégrés spécialisés. Plusieurs fabricants offrent de tels circuit, comme Maxim Integrated avec le DS1307, NXP (anciennement Philips) avec les PFC8523 ou PFC8536 ou encore Texas Instrument avec le bq32000.

Pour communiquer avec un microcontrôleur, ces circuits utilisent généralement les protocoles I2C ou SPI. Voici un schéma de mise en œuvre d'un bq32000 :

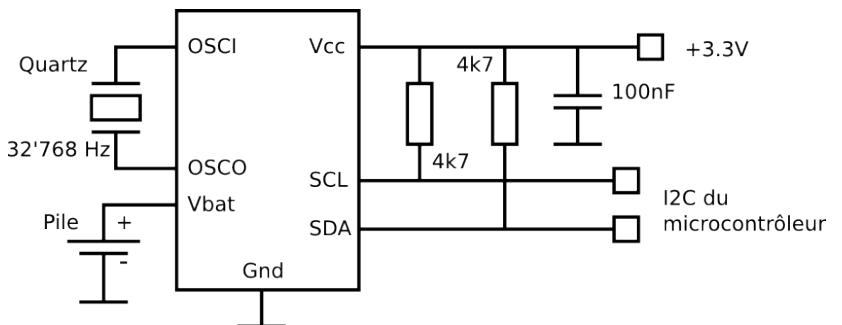
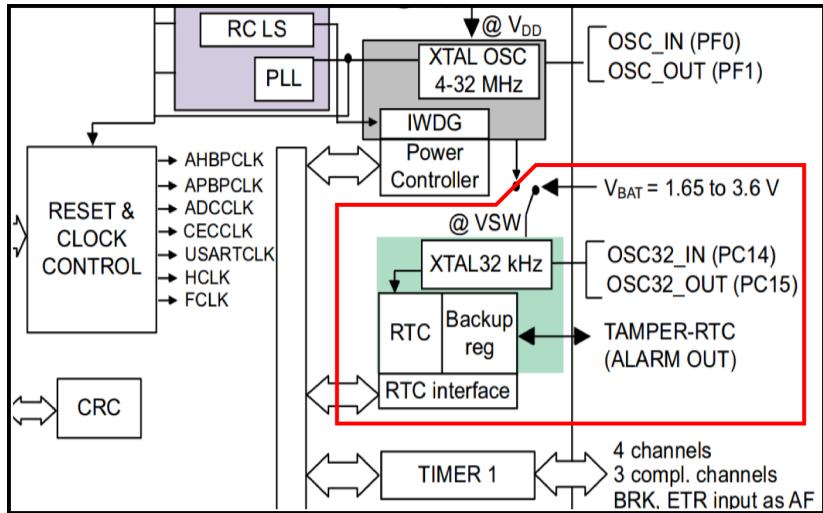


Schéma de la mise en œuvre d'un circuit horloge temps réel

16.7 RTC DANS LE MICROCONTRÔLEUR

Certains microcontrôleurs comportent des circuits logiques qui permettent d'avoir très facilement une horloge en temps réel. L'alimentation de cette partie du circuit intégré est complètement séparée du reste du microcontrôleur. Une entrée permet de brancher directement une pile 3 Volt. C'est le cas par exemple du microcontrôleur STM32F051. La figure montre un extrait de son architecture.



RTC du microcontrôleur STM32f051

Entourée de rouge se trouve une zone isolée, avec son alimentation. Elle comporte une oscillateur à quartz, un diviseur 32 bits, ainsi que quelques positions mémoire RAM, rendues permanentes lorsqu'une pile alimente cette partie du circuit.

16.8 L'HEURE PAR INTERNET

Lorsqu'un afficheur est prévu pour être régulièrement connecté à Internet, il n'est plus indispensable d'avoir une pile pour alimenter un circuit qui conserve l'heure. En effet, il est très facile d'obtenir l'heure par Internet.

C'est le Protocole d'Heure Réseau (Network Time Protocol ou NTP) qui est utilisé, ou sa version simplifiée SNTP.

CHAPITRE 17

LANGAGE INTERPRÉTÉ SPÉCIALISÉ

Pierre-Yves Rochat, EPFL
rév 2016/01/22

17.1 MOTIVATION

Animer une enseigne à LED consiste en une suite d'opérations sur les groupes LED. Animer un afficheur matriciel consiste aussi à envoyer des séquences graphiques. Dans les deux cas, une jolie animation ne se limitera pas à quelques étapes, mais pourra vite devenir longue. Les programmes correspondant vont donc avoir tendance à devenir longs, ce qui va rendre leur lecture fastidieuse et qui risque aussi de remplir rapidement la mémoire du microcontrôleur.

Une technique souvent utilisée consiste à **inventer** un *langage* pour décrire ce qui se passe sur l'enseigne ou l'afficheur et programmer les animations dans ce langage.

17.2 LANGAGE ARDUINO

Prenons l'exemple très simple. Pour décrire une animation sur une enseigne, deux ordres suffisent pour décrire les actions :

- mettre un groupe de LED à une certaine intensité
- attendre un certain temps.

Dans le cas simple de sorties tout-ou-rien, voici les procédures Arduino qui vont suffire :

- digitalWrite() de l'Arduino convient pour donner un état à une sortie
- delay() pour une attente.

En observant la taille d'un petit programme sur Energia et en ajoutant des appels à ces procédures, on constate que :

- digitalWrite() prend 8 octets en mémoire
- delay() prend 10 octets en mémoire.

En prenant par exemple un microcontrôleur MSP430G2213, disposant d'une mémoire flash de 2kB (2048 octets), on sera limité à moins de 80 pas de programme, constitué de paires digitalWrite() - delay(). En constatant qu'un simple chenillard dans les deux sens sur 8 bits en prend déjà 16, c'est réellement limitatif !

```

1 loop() {
2     digitalWrite (P2_0, 1); delay (100);
3     digitalWrite (P2_1, 1); delay (100);
4     digitalWrite (P2_2, 1); delay (100);
5     digitalWrite (P2_3, 1); delay (100);
6     digitalWrite (P2_4, 1); delay (100);
7     digitalWrite (P2_5, 1); delay (100);
8     digitalWrite (P2_6, 1); delay (100);
9     digitalWrite (P2_7, 1); delay (200);
10    digitalWrite (P2_7, 0); delay (100);
11    digitalWrite (P2_6, 0); delay (100);
12    digitalWrite (P2_5, 0); delay (100);
13    digitalWrite (P2_4, 0); delay (100);
14    digitalWrite (P2_3, 0); delay (100);
15    digitalWrite (P2_2, 0); delay (100);
16    digitalWrite (P2_1, 0); delay (100);
17    digitalWrite (P2_0, 0); delay (300);
18 }
```

Bien entendu, les instructions permettant l'accès direct aux registres du microcontrôleur permettent d'économiser la place en mémoire. L'instruction P1OUT |= (1<<0); <---- ---> prend 4 octets. C'est déjà mieux ! Mais cherchons une autre solution.

17.3 INVENTER UN LANGAGE

Une solution élégante est d'inventer un langage. Il aura les deux mêmes instructions :

- **Mettre une intensité sur une sortie.** Paramètres : numéro de la sorte et intensité (0 ou 1)
- **Attendre.** Paramètre : durée de l'attente

Le programme pourrait alors se présenter sous forme d'un tableau. Nous avons utilisé ici un tableau d'octets. Le programme pour notre chenillard se présenterai alors de la manière suivante :

```

1 uint8_t Animation[] = { // définition d'un tableau d'octets
2     Sortie0+On, Attente+10,
3     Sortie1+On, Attente+10,
4     Sortie2+On, Attente+10,
5     Sortie3+On, Attente+10,
6     Sortie4+On, Attente+10,
7     Sortie5+On, Attente+10,
8     Sortie6+On, Attente+10,
9     Sortie7+On, Attente+20,
10    Sortie7+Off, Attente+10,
11    Sortie6+Off, Attente+10,
12    Sortie5+Off, Attente+10,
13    Sortie4+Off, Attente+10,
14    Sortie3+Off, Attente+10,
15    Sortie2+Off, Attente+10,
16    Sortie1+Off, Attente+10,
17    Sortie0+Off, Attente+30,
18    Fin
19 }
```

Sa taille n'est que de 33 octets. Voici les définitions nécessaire pour que ce tableau se compile correctement :

```

1 #define On 0b01000000
2 #define Sortie0 0
3 #define Sortie1 1
4 #define Sortie2 2
5 #define Sortie3 3
6 #define Sortie4 4
7 #define Sortie5 5
8 #define Sortie6 6
9 #define Sortie7 7
10
11 #define Attente 0b10000000
12 #define Fin 0b11111111
```

17.4 LANGAGE BINAIRE

Voici la description binaire de notre langage :

```

1 // Description des instructions :
2 // b7 b6 b5 b4 b3 b2 b1 b0 : instructions sur 8 bits
3 //
4 //   0 i0 s5-s4-s3-s2-s1-s0 : met une intensité sur une sortie
5 //   1 d6-d5-d4-d3-d2-d1-d0 : attente
6 //
7 //
8 // Sorties sur 6 bits (maximum 64 sorties)
9 // Intensité sur 1 bit (On ou OFF)
10 // Durée sur 7 bits, exprimée en dixième de seconde (0 à 12.6 secondes)

```

Ceux qui ont déjà programmé en assembleur trouveront une grande similitude avec la description des instruction en binaire !

On voit que des choix ont été faits pour utiliser au mieux les instructions, qui sont des champs de 8 bits. Le bit de poids fort b7 détermine s'il s'agit d'une instruction pour définir l'intensité ou pour l'attente. Ensuite, les 7 bits restant se répartissent selon l'instruction : une intensité et un numéro de sortie pour l'action sur une sortie, une valeur en dixième de seconde pour l'attente. L'usage de la milliseconde comme unité aurait été trop limitative, étant donné que seuls 7 bits sont à disposition.

17.5 INTERPRÉTEUR

Il reste à écrire une procédure qui va interpréter notre langage et le traduire en instructions pour un microcontrôleur. En voici un exemple :

17.6 EXEMPLE PLUS COMPLEXE