Operations logiques

Ce document s'adresse à des débutants qui ont exécuté et compris quelques programmes ArduinoC et veulent mieux assimiler les opérations arithmétique et logiques.

En C/Arduino, c'est facile de comprendre les opérations arithmétiques, qui nous sont bien familières: `+ - * /` . Les calculs se font en nombres entiers de 8, 16 ou 32 bits. La division (`/`) donne un quotient entier et le modulo (`%`) donne le reste de la division.
Les opérations signées sont parfois délicates, chercher sous Google "Signed operations"

Quelques pas pour bien comprendre les opérations logiques

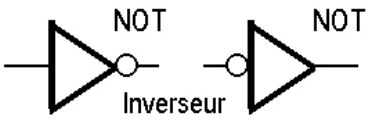
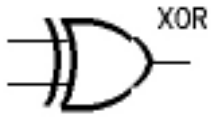
pas1

Une première chose importante en C est la différence entre opérations et comparaisons.
`a=4*b;` est une **opération**, un calcul sur des nombres ou variables
`(a==4)` est une **comparaison**, une question, une opération logique (vrai/faux).
Les signes de comparaison sont

`==`	(est-ce égal ?)	`>`	(est-ce supérieur ?)	`>=`	(est-ce sup ou égal ?)
`!=`	(est-ce différent ?)	`<`	(est-ce inférieur ?)	`<=`	(est-ce sup ou égal ?)

pas2

Une valeur booléenne ou logique a 2 états: Vrai ou Faux. **Faux** est représentée par une valeur numérique 0 et **Vrai** par une valeur différente de zéro (en général 1).
Les fonctions logiques ou booléennes agissent sur un ou deux bits et sont résumées ci-dessous. Des symboles logiques sont utilisés dans les schémas électroniques.

Not !	**And &&**	**Or ||**	**Xor** (^^ pas dispo)
Le résultat a la valeur inverse de l'entrée.	Le résultat vaut 1 si les deux entrées sont à 1	Le résultat vaut 1 si l'une des deux entrées est à 1	Le résultat vaut 1 si les deux entrées sont différentes
Inverseur 	Porte ET 	Porte OU 	Ou exclusif 

pas3

Arduino/C accède à un bit d'une variable ou d'un port avec la notation bitRead (var,bit). Si on veut exécuter un bloc d'instructions si deux bits sont à "1" (condition ET) on écrit
`if (bitRead(var1,bit1) && bitRead(var2,bit2)) { bloc..; }`
Avec Arduino, les pins numérotées sont des variables booléennes et on peut écrire des horreurs comme `if (!2||!3)` pour savoir si l'un des deux poussoirs sur les pins 2 et 3 est activé.
Les opérateurs logiques de comparaison sont `!` (inverse) `&& ||` . A noter que ^^ n'existe pas.

pas4

Dans un microcontrôleur, on effectue les opérations logiques sur des mots de 8,16,32 bits selon le type défini. L'opération et dite "bit-à-bit" (bitwise). Les opérateurs sont
`~` (not) `& | ^` et on met dans cette catégorie les décalages `>> <<` .

Not ~	**And &**	**Or |**	**Xor ^**

`a = 0b10110101`
`~a = 0b01001010`
	`a = 0b10100101`
	`m = 0b00011111`
	`a&m = 0b00000101`
		`a = 0b10100101`
		`m = 0b00011111`
		`a|m = 0b10111111`
			`a = 0b10100101`
			`m = 0b00011111`
			`a^m = 0b10111010`

Attention, -a complément arithmétique de a, n'es pas égal à ~a, complément binaire de a

Dans les opérations "bitwise", l'un des mots est en général vu comme un masque, une passoire, un filtre, qui laisse passer, bloque ou modifie les bits du mot en travail.

pas5

On peut décaler un mot binaire: à droite **Shift right >>** à gauche **Shift left <<**

positifs, non signés "unsigned"		signés (bit de signe sur le poids fort)	
a = 0b10110101 a>>3 = 0b00010110	a = 0b10110101 a<<4 = 0b01010000	a = 0b10110101 a>>3 = 0b11110110	a = 0b10110101 a<<4 = 0b01010000
	dépassement de capacité (bits perdus)	extension du bit de signe	dépassement si le bit de signe n'est pas conservé

pas6

Dans un mot binaire, variable ou port, il faut souvent trier, extraire, forcer dans un état donné des bits. On définit alors un masque qui contient des 1 pour repérer les bits sélectionnés.

Exemple	
mot & masque garde les bits masqués, reste à 0 a 0bxxxxxxxx m 0b01100111 a&m 0b0xx00xxx	mot & ~masque garde les bits non masqués, reste à 0 a 0bxxxxxxxx ~m 0b10011000 a&~m 0bx00xx000
mot | masque garde les bits non masqués, reste à 1 a 0bxxxxxxxx m 0b01100111 a|m 0bx11xx111	mot | ~masque garde les bits masqués, reste à 1 a 0bxxxxxxxx ~m 0b10011000 a|~m 0b1xx11xxx

pas7

L'opérateur de décalage est souvent préféré pour décrire un mot binaire.
Ex: `1<<3` est équivalent à `0b00001000`, `1<<5|1<<0` est équivalent à `0b00100001` .
Pour décrire les masques associés à des périphériques ou des modes il faut donner des noms aux lignes bits et utiliser cette notation. Par exemple, on a 2 sorties sur un port: un moteur sur le bit 3 et un servo sur le bit 5. La valeur à donner au registre de direction (processeur AVR ou MSP) est 0b00101000. C'est du charabia! Il faut déclarer

```
#define bMot = 3
#define bServo = 5
mPort = 1<<bMot | 1<<bServo ;  // mPort sera utilisé pour assigner le registre de direction
```

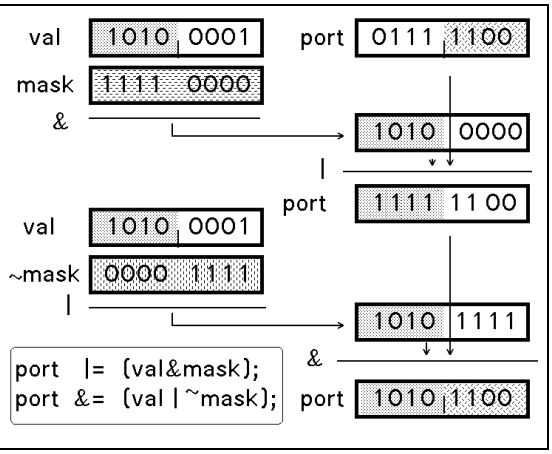
A noter qu'écrire `mPort = 1<<3 | 1<<5 ;` est aussi du charabia!

Application: modifier un port

Si on modifie un port ou un registre de direction, il faut éviter de créer des impulsions parasites (glitch) sur des sorties. Une seule instruction ne suffit pas toujours pour un changement, donc le risque d'un état non souhaité entre les deux instructions.
Prenons un exemple simple. Les 4 bits de poids faible d'un `port` sont intouchables.
On a préparé un registre `val`, dont les 4 bits de poids fort doivent remplacer les 4 bits de poids fort du `port` .

La solution la plus générale est d'agir deux fois de suite sur le port, une premier fois pour forcer à 1 les bits de val à 1, une 2e fois pour forcer à 0 les bits de val à 0.
Avec cette solution, le compilateur ne génère pas de variable supplémentaire et il n'y a pas de glitch, mais un décalage dans le temps des bits qui passent à 1 et à zéro.
Les instructions à écrire sont

```
#define mask = 0b11110000
port |= val & mask ;
port &= val | ~mask ;
```



On peut naturellement modifier un bit à la fois avec des bitSet () et bitClear (), ce qui est plus efficace pour 1-2 bits à changer, et moins rique à l'erreur.

Les ports Kidules sur Arduino/Diduino sont des exemples intéressants, voir
www.didel.com/kidules/PortK.pdf

jdn 130619/0706/1106