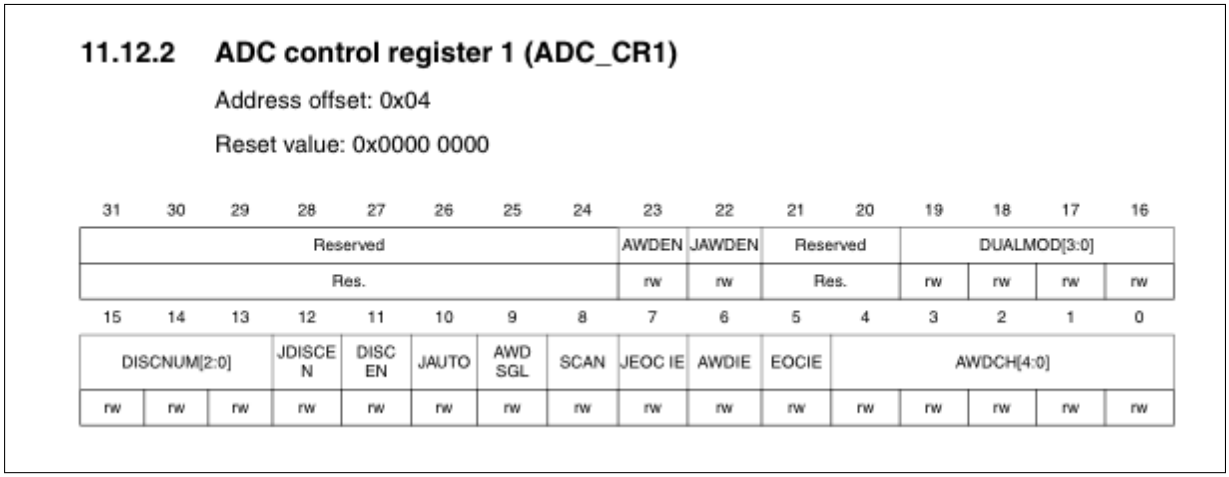
Fiche méthode :
Bah ! Les masques

Periph'Team - INSA de Toulouse

Pour programmer un périphérique il est nécessaire d'aller modifier un ou plusieurs bits dans un registre sans modifier les autres. Par exemple le registre `ADC_CR1` sert à configurer le convertisseur analogique–numérique `ADC1` :



11.12.2 ADC control register 1 (ADC_CR1)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								AWDEN	JAWDEN	Reserved		DUALMOD[3:0]			
Res.								rw	rw	Res.		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 1: *Extrait du* reference manual *du STM32 p. 236, la suite décrit la fonction de chaque bit, comme* `SCAN` *et* `AWDIE` *et la signification des valeurs de chaque tranches de bit tels que* `DISCNUM`

Pour cela on utilise les masques logiques et certaines astuces pour construire un masque clairement sans risquer de se tromper. Dans l'exemple suivant le bit `SCAN` de `ADC_CR1` est mis à 1 et le bit `EOCIE` est mis à 0 sans toucher aux autres bits. Les registres `ADC1_CR2`, `ADC1_SQR1`, `ADC1_SQR3` sont aussi manipulés avec des masques :

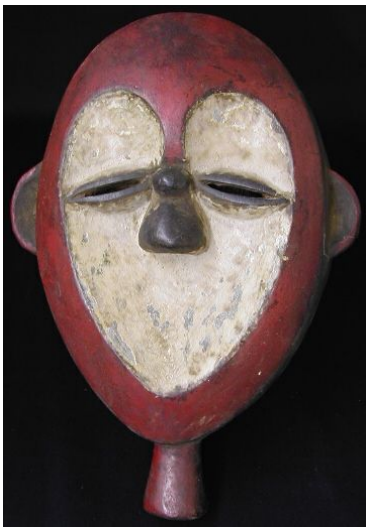
```
ADC1->CR1  |= (ADC_SCAN);     // continuous scan of channels 1,14,15

ADC1->CR1  &= ~(ADC_EOCIE);   // pas d'interruption de fin de conv.

ADC1->CR2  |= (ADC_EXTSEL_ON_SWSTART | ADC_ CONT | ADC_DMA);
    // EXTSEL = SWSTART
    // use data align right,continuous conversion
    // send DMA request
```

```
// convert sequence is channel 1 then 14 then 15
ADC1->SQR3  |= (1 <<SQ1_SHIFT)  |  (14 <<SQ2_SHIFT) | (15 <<SQ3_SHIFT);
```

Listing 1: *Extrait de la configuration de l'ADC de la baguette magique vue en assembleur*



Si vous avez aucune idée de comment ce code fonctionne c'est que vous ne maitrisez pas encore l'art primitif du masque : lisez–donc la suite.
Ci–contre, un masque primitif (on préfère dire d'art premier*) d'origine Gabonaise.*

1 Opérateur logiques et bit à bit

On utilise les opérateurs logiques ET, OU, XOR (OU exclusif) et NOT entre un registre et un masque pour manipuler les bits. Les opérateurs logiques et leurs syntaxe en langage C sont résumés dans le tableau suivant :

Fonction logique	Opérateur logique	Opérateur bit à bit
ET	&&	&
OU	||	|
XOR	aucun	^
NON	!	~

L'opérateur logique considère les opérandes (qu'elle soient 8/16/32 bit) comme une valeur booléenne fausse si tous les bits sont nuls et vrai sinon. Elle fournit un résultat booléen nul si c'est faux et différent de zéro (en général la valeur 1) sinon. Ne confondez donc pas l'opérateur logique avec l'opérateur bit à bit qui effectue 8/16/32 opérations logiques entre chaque bits respectifs des opérandes, par exemple :

```
a = 2; // soit b00000010 en binaire est vrai car different de 0
b = 1; // soit b00000001 en binaire est vrai aussi
y = a && b; // donne 1(vrai) car a ET b est vrai
z = a & b; // donne b00000000 car chaque ET des bits de a et b sont faux
```

2 Mettre un bit à 1

Pour cela on utilise l'opérateur OU avec une valeur binaire, appelée *masque*, ayant des bits à 1 uniquement devant les bits que l'on veut initialiser :

$$\begin{array}{cccccccccc} & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ OU & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ = & b_7 & b_6 & b_5 & 1 & b_3 & b_2 & 1 & 1 \end{array} \quad \text{car} \quad x\ OU\ 1 = 1 \quad \text{et} \quad x\ OU\ 0 = x$$

Ainsi les bits b_4, b_1 et b_0 sont passés à 1 sans modifier la valeur des autres bits.

Pour effectuer cela en langage C on doit calculer la valeur du masque binaire : convertir `b00010011` en hexadécimal (`0x13`) ou en décimal (19) car le langage C n'admet pas de littéraux en binaire.

```
char avoile;
...
//formulations equivalentes
avoile = avoile |  0x13;   // operateur | inline
avoile |=  0x13;            // operateur | prefixe a =
avoile |=  19;              // valeur du masque en decimal
```

Il n'est pas très évident de comprendre que `0x13` ou `19` correspond à un masque visant les bits de rang 0,1 et 4, de plus il est très facile de se tromper lorsque l'on fait la conversion soi–même. Un *geek* utilisera l'opérateur de décalage à gauche `<<x` pour positionner un 1 devant le bit de rang x pour construire sont masque ainsi :

			b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	$(1 << 0)$	$\rightarrow$	0	0	0	0	0	0	0	1
OU	$(1 << 1)$	$\rightarrow$	0	0	0	0	0	0	1	0
OU	$(1 << 4)$	$\rightarrow$	0	0	0	1	0	0	0	0
$=$	$(1 << 0)\|(1 << 1)\|(1 << 4)$	$\rightarrow$	0	0	0	1	0	0	1	1

Ainsi le code suivant est plus lisible et ne risque pas de comporter d'erreur de conversion :

```
//formulations equivalentes
avoile = avoile |  0x13;   // operateur | inline
avoile = avoile | (1<<0)|(1<<1)|(1<<4);
avoile |= (1<<0)|(1<<1)|(1<<4);
```

3 Mettre un bit à 0

Pour cela on utilise l'opérateur ET avec une masque ayant des bits à 0 uniquement devant les bits que l'on veut initialiser :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
ET	1	1	1	0	1	1	0	0
$=$	b_7	b_6	b_5	0	b_3	b_2	0	0

car $x\ ET\ 1 = x$ et $x\ ET\ 0 = 0$

Ainsi les bits b_4, b_1 et b_0 sont passés à 0 sans modifier la valeur des autres bits.

Pour construire le masque, on peut toujours effectuer la conversion soit-même avec un risque d'erreur : $b11101100 = 0xEC = 236$. On peut aussi construire le masque avec des 1 devant les bits à annuler et ensuite inverser chaque bit avec l'opérateur **~** .

				b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
		$(1 << 0)\|(1 << 1)\|(1 << 4)$	$\rightarrow$	0	0	0	1	0	0	1	1
	~	$(1 << 0)\|(1 << 1)\|(1 << 4)$	$\rightarrow$	1	1	1	0	1	1	0	0
ET		avoile	$\rightarrow$	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$=$	avoile &	**~**$((1 << 0)\|(1 << 1)\|(1 << 4))$	$\rightarrow$	b_7	b_6	b_5	0	b_3	b_2	0	0

Ce qui donne en langage C :

```
//formulations equivalentes
avoile = avoile &  0xEB;
avoile = avoile & ~((1<<0)|(1<<1)|(1<<4));
avoile &= ~((1<<0)|(1<<1)|(1<<4));
```

4 Inverser un bit

Pour cela on utilise l'opérateur XOR avec une masque ayant des bits à 1 uniquement devant les bits à inverser :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
XOR	0	0	0	1	0	0	1	1
$=$	b_7	b_6	b_5	$\overline{b_4}$	b_3	b_2	$\overline{b_1}$	$\overline{b_0}$

car $x\ XOR\ 1 = \overline{x}$ et $x\ XOR\ 0 = x$

Ainsi seuls les bits b_4,b_1 et b_0 ont été inversés.
On construit les masques comme les masque OU de mise à 1.

5 Initialiser une tranche de bits

Une tranche de bit est un ensemble de quelques bits contigus dont la valeur a une signification particulière. Par exemple `DISCNUM` est une tranche de 3 bits du registre `ADC_CR1` qui indique le nombre de canaux à convertir.

Un programmeur avertis peut désirer initialiser une tranche de `avoile` à la valeur contenue dans `numero`. Pour cela il faut procéder en trois étapes : limiter la valeur de `numero` pour ne pas dépasser la tranche de 3 bits et la caler au bon endroit ; annuler la tranche de trois bits de `avoile` ; puis recopier les 1 du premier masque dans `avoile` avec un OU :

$$
\begin{array}{lrcccccccc}
 & \text{num} & \rightarrow & n_7 & n_6 & n_5 & n_4 & n_3 & n_2 & n_1 & n_0 \\
 & 0x7 & \rightarrow & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
 & (\text{num} \,\&\, 0x7) & \rightarrow & 0 & 0 & 0 & 0 & 0 & n_2 & n_1 & n_0 \\
 & (\text{num} \,\&\, 0x7) << 4 & \rightarrow & 0 & n_2 & n_1 & n_0 & 0 & 0 & 0 & 0 \\
OU & \text{avoile} \,\&\, \sim(0x7 << 4) & \rightarrow & b_7 & 0 & 0 & 0 & b_3 & b_2 & b_1 & b_0 \\
= & (\text{avoile} \,\&\, \sim(0x7 << 4)) \,|\, (\text{num} \,\&\, 0x7) << 4 & \rightarrow & b_7 & n_2 & n_1 & n_0 & b_3 & b_2 & b_1 & b_0
\end{array}
$$

Le programme suivant permet d'affecter la tranche `DISCNUM` avec la valeur contenue dans la variable `numero` :

```
void set_discnum ( char numero)
{
   // raz de la tranche DISCNUM : 3 bits de rang 13-15 de ADC_CR1
   ADC->CR1 &= ~(0x7<<13); // seuls les bits 13-15 du masque valent 0

   // init de la tranche avec numero
   ADC->CR1 |= numero<<13 // recopie les 1 de numero decale au rang 13

   // Attention si numero > 7 ca deborde sur les bits 16 a 31
   //Un masque doit limiter numero de 0 a 7 : pas plus de trois bits a 1
   // On prefere donc cette ligne
   ADC->CR1 |= (numero & 0x7) <<13 // numero est limite
 }
```