Chapitre

7

Codes représentatifs

Le chapitre 5 a en bonne partie couvert la matière portant sur les codes analytique. Le chapitre qui débute ici complète ces notions en les étendant aux codes dits représentatifs. Les codes représentatifs se distinguent des codes analytiques en cela qu'ils ne sont pas restreints aux objets numéraux. Les codes sont utilisés par les systèmes logiques pour véhiculer de l'information sous encodage binaire. Les codes représentatifs auxquels nous nous intéresserons dans ce chapitre sont principalement utilisés pour représenter des caractères alphanumériques.

7.1 Codes représentatifs

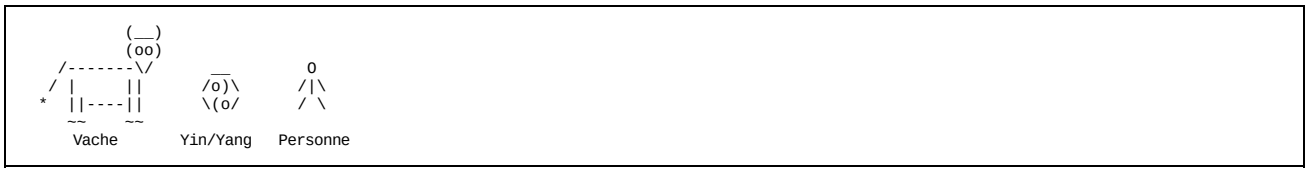
7.1.1 Code Ascii

Le Code ASCII (acronyme de American Standard Code for Information Interchange) est l'un des plus anciens et certainement le plus largement utilisé des codes représentatifs. Le tableau suivant énumère quelques caractères du code ASCII

Caractère	Code	Caractère	Code	Caractère	Code	Caractère	Code
0	0110000	I	1001001	a	1100001	s	1110011
1	0110001	J	1001010	b	1100010	t	1110100
2	0110010	K	1001011	c	1100011	u	1110101
3	0110011	L	1001100	d	1100100	v	1110110
4	0110100	M	1001101	e	1100101	w	1110111
5	0110101	N	1001110	f	1100110	x	1111000
6	0110110	O	1001111	g	1100111	y	1111001
7	0110111	P	1010000	h	1101000	z	1111010
8	0111000	Q	1010001	i	1101001		
9	0111001	R	1010010	j	1101010		
A	1000001	S	1010011	k	1101011		
B	1000010	T	1010100	l	1101100		
C	1000011	U	1010101	m	1101101		
D	1000100	V	1010110	n	1101110		
E	1000101	W	1010111	o	1101111		
F	1000110	X	1011000	p	1110000		
G	1000111	Y	1011001	q	1110001		
H	1001000	Z	1011010	r	1110010		

Le code ASCII présenté dans ce tableau a été initié par la compagnie américaine *Bell* pour représenter les caractères alphanumériques usuels de l'anglais. Cette version du code ASCII ne comportait que 7 bits par mot. Ce qui ne permettait de représenter que $2^7 = 128$ mots possibles. Lorsque le code fut ouvert à d'autres langues, des versions standardisées et rendues internationales suivirent, aboutissant à une représentation sur 8 bits, permettant d'obtenir $2^8 = 256$ mots possibles, permettant de représenter des caractères spéciaux tel que le 'é', 'è' du français. Néanmoins, avec le développement des technologies de l'information, des représentations plus évoluées ont vu le jour, dont le code Unicode, comportant 16 bits, pour un total de 65 536 mots possibles et permettent de représenter différents caractères des nombreux alphabets utilisés un peu partout dans le monde.

Le code ASCII s'est depuis plié au bonheur des artistes, donnant naissance à ce qu'on appelle le Art-ASCII. Ainsi, on dessine des choses simples du type :



7.1.2 Code BCD

Le code BCD est un autre code représentatif très en usage. Il permet de représenter des nombres sous leur écriture décimale. L'acronyme BCD est pour Binary Coded Decimal, en français Décimal, Codé Binaire.

Il s'agit de coder les chiffres de 0 à 9 en binaire. Le code 8421 est le plus répandu. Le tableau suivant en résume la correspondance :

Chiffre	Bits	Chiffre	Bits
0	0000	**5**	0101
1	0001	**6**	0110
2	0010	**7**	0111
3	0011	**8**	1000
4	0100	**9**	1001

Code BCD : 8421

On appelle ce code 8421 car les 4 bits pondèrent les nombres 8, 4, 2 et 1 respectivement. Ainsi, 0111 correspond à 7 puisque 7 = **0** x 8 + **1** x 4 + **1** x 2 + **1** x 1.

Cela permet d'écrire 123 : 0001 0010 0011.

D'autres codes BCD existent, certains fonctionnant selon le même principe de pondération, comme le code 2421 ou le code 5421, dont les tables suivent :

Chiffre	Bits	Chiffre	Bits
0	0000	**5**	1011
1	0001	**6**	1100
2	0010	**7**	1101
3	0011	**8**	1110
4	0100	**9**	1111

Code BCD : 2421

Chiffre	Bits	Chiffre	Bits
0	0000	**5**	1000
1	0001	**6**	1001
2	0010	**7**	1010
3	0011	**8**	1011
4	0100	**9**	1100

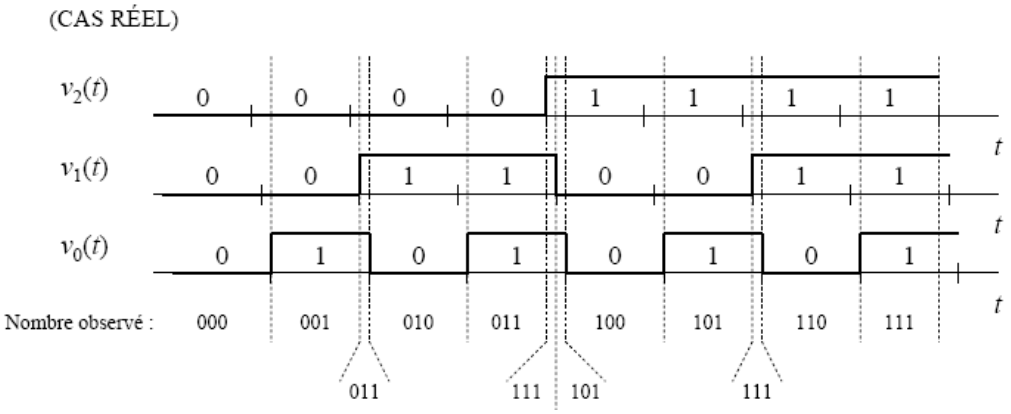
Code BCD : 5421

Il existe également des codes non basés sur la pondération, dont l'exemple le plus connu est certainement le code-barres. Pour l'instant, on retiendra du code BCD son but utilitaire, à savoir la représentation des chiffres 0 à 9 en code binaire. Le grand intérêt du système BCD pour l'électronicien est la possibilité de manipuler les chiffres en base dix pour différentes fins, notamment l'affichage, comme ce sera illustré dans les chapitres subséquents.

7.1.3 Code de Gray

Comme nous aurons le temps de l'apprécier plus tard, un des problèmes importants des systèmes numériques tient dans la synchronisation des différents signaux numériques. Lorsque l'on conçoit un compteur, ce problème devient important car dans la représentation binaire des nombres entiers, il arrive souvent que plusieurs bits changent en même temps. Dans ce cas, il est possible que des nombres non désirés apparaissent dans le compte.

Considérons l'exemple suivant. Nous voulons concevoir un compteur trois bits, en faisant varier trois signaux, v_0, v_1, v_2 :



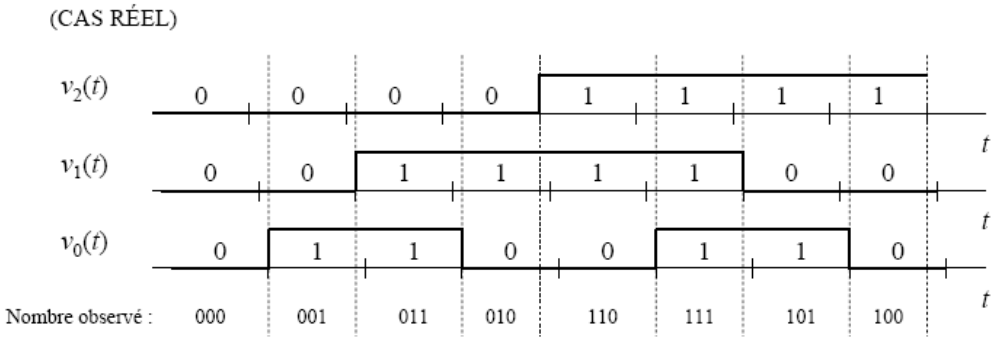
Lorsque le compteur passe de 001 à 010, il doit modifier deux bits. Si il y a un problème de synchronisation entre les bits v_1 et v_0, le nombre 011 s'affiche pendant un court laps de temps, bien que ce ne soit pas le résultat recherché.

Pour remédier à ce problème il est possible d'utiliser le code de Gray, inventé aux *Bell Labs* par Frank Gray et breveté en 1953.

Le code de Gray encode les entiers de telle façon que le passage d'un nombre au suivant ne change qu'un seul bit à la fois, comme l'illustre le tableau suivant où les entiers ont été encodés sur 4 bits :

Décimal	Binaire	Gray
0	00000	0000
1	00001	0001
2	00010	0011
3	00011	0010
4	00100	0110
5	00101	0111
6	00110	0101
7	00111	0100
8	01000	1100
9	01001	1101
10	01010	1111
11	01011	1110
12	01100	0110
13	01101	0111
14	01110	0101
15	01111	0100

Le code de gray règle le problème de compte que nous avons vu précédemment, comme l'illustre la figure suivante :



On construit le code de Gray en recopiant les bits de façon symétrique (effet miroir) et en procédant de façon itérative jusqu'au bit désiré.

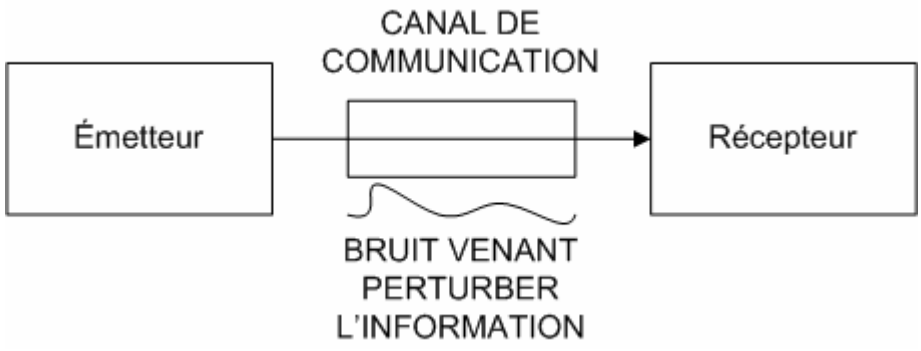
0	0 0	0 00
1	0 1	0 01
	1 1	0 11
	1 0	0 10
		1 10
		1 11
		1 01
		1 00
Itération 1	Itération 2	Itération 3

Le code de Gray est encore très utilisé, notamment dans les ordinateurs où le compteur de programme utilise le code de Gray pour minimiser sa consommation d'énergie. On le retrouve également dans certains systèmes de télécommunication pour la correction d'erreur. Mentionnons aussi que le code de Gray a été utilisé pour trouver une solution au problème des tours de Hanoi.

7.2 Détection d'erreurs

Les systèmes de communication numériques souffrent du bruit ambiant autant que leur pendant analogique. Le bruit est un signal parasitaire non désiré dont on ne peut pas prévoir le comportement. Très tôt, les ingénieurs ont été confrontés à ce problème et ont dû définir des outils leur permettant d'assurer l'intégrité des messages envoyés.

On part de la problématique générale suivante :



On dispose d'une source émettrice (l'émetteur) et d'un dispositif pour la réception (le récepteur). Pour transférer les données de l'émetteur au récepteur, on utilise un canal de communication. Ces trois éléments se retrouve très fréquemment dans les systèmes numériques modernes. Les exemples ne manquent pas. Citons à titre d'exemple (émetteur, canal, récepteur) : Souris d'ordinateur, fil, ordinateur ; télécommande, lien infrarouge, télévision ; modem, cable RJ45, ordinateur ; guichet automatique, ligne téléphonique, central de la banque ; etc….

Le bruit vient altérer l'information transmise. Cela signifie qu'il peut altérer les bits envoyés de telle sorte qu'un 0 devienne 1 et vice versa. Il est cependant possible, en utilisant un protocole approprié permettant de retrouver l'information même si elle a été modifiée par le bruit.

Pour bien comprendre la problématique, supposons que nous disposions d'un code utilisant deux bits d'information, et que ce code soit formé des mots : {00, 01, 10, 11}. Si l'émetteur envoie le mot 00, et que le récepteur obtient 01 (altération du bit le moins significatif), il est clair que le récepteur est incapable d'identifier l'information reçue après intervention du bruit car il serait dans l'incapacité de savoir si aucun bruit n'est venu altérer le message émis qui aurait été 01, ou que le bruit ait altéré un bit, auquel cas le message aurait tout autant pu être 00 que 11.

7.2.1 Distance de Hamming

Dans l'exemple précédent, la difficulté rencontrée dans l'identification des mots reçus réside dans le fait que modifier un bit d'un mot du code revienne à générer un nouveau mot du même code. Supposons maintenant que nous disposions du code formé des mots : {0000, 0011, 1100, 1111}. Il est évident que dans ce cas, si l'émetteur envoie 0000, et que le bruit vient altérer un des bits de sorte que le récepteur reçoive 0100 par exemple, le récepteur est en mesure de constater que le mot reçu n'existe tout simplement pas dans le code, et pourrait demander une retransmission du message.

Afin de mieux manipuler la possibilité de corriger une erreur si elle survient, nous allons nous intéresser à une métrique qui va nous permettre de mesurer la capacité d'un système de communication numérique à s'immuniser au bruit : la distance de Hamming. La distance de Hamming, qui tient son nom d'un informaticien célèbre qui a participé au projet Manhattan, est la mesure entre deux mots dans un code. Pour cela, elle considère le nombre de bits qui ont changé de valeur entre le premier mot et le second. Reprenons l'exemple du paragraphe précédent. Les mots 0000 et 0011 ont une distance de Hamming de 2, car les deux derniers bits ont changé quand les mots passaient de 0000 à 0011. Le code de Gray par exemple offre une distance de Hamming de 1 entre deux mots consécutifs.

Dans un code, on parlera de distance minimale, que nous noterons M. La distance de Hamming minimale d'un code est la plus petite distance entre deux mots différents du code. Pour les deux exemples précédents, on aura :

	00	01	10	11
00	-	1	1	2
01	1	-	2	1
10	1	2	-	1
11	2	1	1	-

Distance minimale : 1

	0000	0011	1100	1111
0000	-	2	2	4
0011	2	-	4	2
1100	2	4	-	2
1111	4	2	2	-

Distance minimale : 2

7.2.2 Théorie de la détection d'erreurs

Grâce aux développements qui vont suivre, nous allons être en mesure d'évaluer la capacité d'un système de communication à détecter une erreur, et même de pouvoir la corriger. À cet effet, la théorie de la détection d'erreur stipule que :

$$M - 1 = D + C$$

où :
M est la distance minimale d'un code ;
D est le nombre de bits pouvant être détecté ;
C est le nombre de bits pouvant être corrigés, sachant que $C \leq D$, c'est-à-dire que toute erreur corrigée doit nécessairement être détectée au préalable.

Considérons ici l'exemple du code formé des mots {0000, 0011, 1100, 1111}, dont la distance minimale M est 2. Sachant que M=2, on trouve M – 1 = 1, ce qui signifie que le mieux qu'on puisse faire est de poser D=1 et C = 0, puisque $C \leq D$. Ce résultat signifie qu'il est possible de détecter une erreur sur un bit lorsqu'elle survient car D = 1 (si l'émetteur envoie 0000, et qu'à la réception, le mot apparaisse comme étant 0100, on sait qu'il y a erreur), mais qu'il est alors incapable de corriger l'erreur détecter car C = 0 (lorsque le récepteur reçoit 0100, il est en mesure de savoir que le mot n'appartient pas au code, mais il ne sait pas pour autant si le mot envoyé était 1100 ou 0000).

On comprend de ce qui précède qu'il est nécessaire de garantir M = 3 pour qu'un système de communication soit en mesure de détecter une erreur sur 1 bit, et de la corriger :

$$M = 3 \Rightarrow M - 1 = 2$$
$$\Rightarrow C + D = 2$$
$$\Rightarrow (D = 2 \text{ et } C = 0) \text{ ou } (D = 1 \text{ et } C = 1)$$

Ce qui précède signifie : si il y a deux bits erronés, le système va détecter l'erreur, mais ne sera pas en mesure de corriger l'erreur. Cependant, si il y a un seul bit erroné, le système détectera l'erreur et sera en mesure de la corriger.

La difficulté réside dans le fait d'être en mesure de trouver un code qui permette d'obtenir une distance minimale M valant 3. Dans ce qui suit, nous allons montrer des méthodes systématiques permettant d'atteindre ce niveau de sécurité.

7.2.3 Quantité d'information

Nous allons nous outiller d'une autre métrique qui réfère au coût de l'information d'un code. Supposons que nous utilisions un code qui comporte n mots, et qu'il soit possible de représenter N mots différents en utilisant l'écriture utilisée par ce code, la quantité d'information[1], noté I, est égale à $I = \log_2(N/n)$.

Plus cette valeur s'approche de 0, et plus le codage est optimal. Supposons que l'on veuille représenter de façon binaire les 26 lettres de l'alphabet. Pour ce faire, on utilise 5 bits :

00000	A	01010	K	10100	U
00001	B	01011	L	10101	V
00010	C	01100	M	10110	W
00011	D	01101	N	10111	X
00100	E	01110	O	11000	Y
00101	F	01111	P	11001	Z
00110	G	10000	Q		
00111	H	10001	R		
01000	I	10010	S		
01001	J	10011	T		

Ici : $n = 26$, $N = 2^5 = 32$:

$$\begin{aligned} I &= \log_2(N/n) \\ &= \log_2(32/26) \\ &= \log_2(1{,}230) = 0{,}2988 \\ I &\approx 0{,}3 \end{aligned}$$

7.2.4 Encodage avec redondance

Le premier type d'encodage est l'encodage avec redondance. Nous avions vu à la section 7.2.1 que répéter chaque bit deux fois permettait d'avoir M = 2. Il est possible de montrer que si l'on répète chaque bit n fois, on aura M = n. Ainsi, pour obtenir un code avec M = 3, il suffit de répéter chaque bit du code 3 fois. Par exemple, pour un code comportant {00, 01, 10, 11}, on peut le modifier de sorte à ce que les mots du codes correspondent respectivement à {000000, 000111, 111000, 111111}. Dans ce cas là, on trouve que la distance minimale M est 3.

[1] On appelle I quantité d'information car elle quantifie l'information supplémentaire (bits) utilisée pour porter l'information globale nécessaire. Dans l'exemple des 26 lettres de l'alphabet, la valeur est proche de 0 (mais non nulle) car des certains mots du code (6 au total) existent sans qu'il y ait de symbole qui leur soit associé.

	000000	000111	111000	111111
000000	-	3	3	6
000111	3	-	6	3
111000	3	6	-	3
111111	6	3	3	-

Distance minimale : M = 3

Il est intéressant de connaître le coût d'un tel encodage. Dans notre cas, la quantité d'information I vaut : $I = \log_2(2^6/2^2) = \log_2(2^4) = 4$. Notons que cette valeur est égale au nombre de bits ajoutés. Cela est dû au fait que le code de départ possédait une quantité d'information nulle que nous avons alourdie de quatre bits supplémentaire.

7.2.5 Encodage avec bit de parité

Il est possible d'ajouter un bit de parité au mot d'un code avant de le transmettre. Supposons que nous disposions de n bits, b_i, en ajoute un bit p pour la parité :

$$b_{n-1}b_{n-2}\ldots b_2b_1b_0 => b_{n-1}b_{n-2}\ldots b_2b_1b_0\text{p}$$

Le bit p est tel que :

- p=1 si le nombre de bits 1 est pair (le plus fréquemment utilisé)
- p=1 si le nombre de bits 0 est pair
- p=1 si le nombre de bits 1 est impair
- p=1 si le nombre de bits 0 est impair

Nous utiliserons dans la suite de l'ouvrage le premier de ceux-là. En ajoutant un bit de parité, on ajoute 1 à M, la distance de Hamming minimale du code. Prenons l'exemple du code {00, 01, 10, 11}, nous aurons le code {000, 011, 101, 111}. Et comme prévu, on trouve M :

	000	011	101	110
000	-	2	2	2
011	2	-	2	2
101	2	2	-	2
110	2	2	2	-

Distance minimale : M = 2

Dans ce cas, un système de communication sera en mesure de détecter une erreur sur un bit, mais ne sera pas en mesure de la corriger. Si nous reprenons la mesure logarithmique établie plus tôt, nous aurons dans ce cas : $\log_2(2^4/2^3) = \log_2(2) = 1$. Cette valeur ne change pas quelle que soit le nombre de bits transmis.

7.2.6 Encodage avec parité orthogonale

La parité orthogonale est une technique permettant d'obtenir une distance de Hamming minimal M = 3. Pour ce faire, on considère un code de longueur n x m bits organisés en n groupes de m bits. On ajoute aux m x n bits, $m+n+1$ bits de parité. Pour ce faire, on utilise une matrice de parité orthogonale.

Afin de bien comprendre la technique, considérons l'exemple suivant, où nous prenons $n = 3$ $m = 4$

0001	1
1101	1
0100	1
1000	1

Les bits à envoyer sont ici : 000111010100. Sur chaque ligne, on calcule le bit de parité. On calcule ensuite le bit de parité sur chaque colonne, et finalement, en ligne ou en colonne, le bit de parité des bits de parité obtenus (que ce soit en ligne ou en colonne, la valeur sera identique).

On trouve toujours $I = m + n + 1$ et $M = 3$. L'avantage de la méthode tient surtout dans la facilité avec laquelle le système est en mesure de corriger l'erreur sur un bit si jamais elle survient. Considérons à nouveau l'exemple précédent.

Nous voulons envoyer 000111010100 partagé sur les mots 0001, 1101 et 0100. Pour ce faire, on utilise la parité orthogonale et envoyons 4 (m+1) mots de 5 bits (n+1) : 00011, 11011, 01001 et 10001. Supposons maintenant qu'un des bits de cette séquence soit affecté par le bruit du canal de communication, et qu'à la réception, nous obtenions 00111, 11011, 01001 et 10001 au lieu de 00011, 11011, 01001 et 10001.

Le récepteur reconstruit la matrice avec les bits reçus :

0011	1
1101	1
0100	1
1000	1

Que ce soit sur la troisième colonne ou la première ligne, on constate que les bits de parité ne correspondent pas au résultat attendu. Il y a donc erreur (détection d'erreur). Pour corriger l'erreur, il faut retrouver le bit qui se trouve à l'intersection de la ligne et la colonne où la parité n'est pas respectée :

0011	1
1101	1
0100	1
1000	1

Il suffit alors d'inverser la valeur de ce bit :

0001	1
1101	1
0100	1
1000	1

Ce qui permet de retrouver le message transmis : 0001, 1101 et 0100 → 000111010100.

7.2.7 Code de Hamming

Hamming a proposé une méthode systématique permettant de créer un code où M = 3. Supposons un code de longueur fixe *n*, Hamming propose d'ajouter p bits de parité, de sorte à respecter l'inégalité suivante :

$$n + p + 1 \leq 2^p$$

Nous obtenons la table (incomplète) suivante :

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…
p	2	3	3	3	4	4	4	4	4	4	4	5	5	5	5	5	…

Les codes sont formés de la façon suivante :

1. Pour chaque mot du code de longueur *n*, on forme un nouveau mot de longueur *n*+p.
2. Chaque bit de parité porte un index *i* qui début par 1 (p_1, p_2, p_3, p_4…)
3. Les bits du nouveau mot (de longueur *n*+*p*) sont indexés selon *j* et on débute par 1 (b_1, b_2, b_3, b_4…)
4. Chaque bit de parité (d'index *i*) prend la position de bit b_j dont l'index *j* est une puissance de 2 (1, 2, 4, 8…) de sorte que $j = 2^{i-1}$
5. Les bits du mot de départ sont placés dans les espaces libres restant
6. Chaque bit de parité d'index *i* positionné au bit de position *j* ($j = 2^{i-1}$) mesure la parité d'une suite de bits qui débutent à l'index *j*, de telle sorte que l'on prend alternativement *i* bits, une fois sur deux.

Le cas particulier qui nous concerne ici est celui où $n = 4$. Les règles précédentes deviennent :

1. Pour chaque mot du code de longueur 4, on forme un nouveau mot de longueur 7.
2. Chaque bit de parité porte un index *i* qui début par 1 (p_1, p_2, p_3)
3. Les bits du nouveau mot sont indexés selon *j* et on débute par *j*=1 (b_1, b_2, b_3, b_4, b_5, b_6, b_7)
4. Chaque bit de parité (d'index *i*) prend la position de bit dont l'index *j* est une puissance de 2 (1, 2, 4) de sorte que $j = 2^{i-1}$
5. Les bits du mot de départ sont placés dans les espaces libres restant d'index 3, 5, 6, 7
6. Chaque bit de parité d'index *i* positionné au bit de position *j* ($j = 2^{i-1}$) mesure la parité d'une suite de bits qui débutent à l'index *j*, de telle sorte que l'on prend *i* bits, puis saute *i* bits, puis prend *i* bits et ainsi de suite… Ce qui donne

   p_1 = parité(b_1, b_3, b_5, b_7) = parité(b_3, b_5, b_7)
   p_2 = parité(b_2, b_3, b_6, b_7) = parité(b_3, b_6, b_7)
   p_3 = parité(b_4, b_5, b_6, b_7) = parité(b_5, b_6, b_7)

Regardons l'exemples suivant. On veut envoyer le mot 0110 :

1. On forme un nouveau mot de longueur 7 :

2. Chaque bit de parité porte un index *i* qui début par 1 (p_1, p_2, p_3)
3. Les bits du nouveau mot sont indexés selon *j* et on débute par 1 (b_1, b_2, b_3, b_4, b_5, b_6, b_7)

b_7	b_6	b_5	b_4	b_3	b_2	b_1

4. Chaque bit de parité (d'index *i*) prend la position de bit dont l'index *j* est une puissance de 2 (1, 2, 4) de sorte que $j = 2^{i-1}$

			p_3		p_2	p_1
b_7	b_6	b_5	b_4	b_3	b_2	b_1

5. Les bits du mot de départ sont placés dans les espaces libres restant d'index 3, 5, 6, 7

0	1	1	p_3	0	p_2	p_1
b_7	b_6	b_5	b_4	b_3	b_2	b_1

6. Chaque bit de parité d'index *i* positionné au bit de position *j* ($j = 2^{i-1}$) mesure la parité d'une suite de bits qui débutent à l'index *j*, de telle sorte que l'on prend *i* bits, puis saute *i* bits, puis prend *i* bits et ainsi de suite… Ce qui donne :

p_1 = parité(b_1, b_3, b_5, b_7) = parité(b_3, b_5, b_7) = parité(0, 1, 0) = 1
p_2 = parité(b_2, b_3, b_6, b_7) = parité(b_3, b_6, b_7) = parité(0, 1, 0) = 1
p_3 = parité(b_4, b_5, b_6, b_7) = parité(b_5, b_6, b_7) = parité(1, 1, 0) = 0

0	1	1	0	0	1	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1

Le code de Hamming permet d'obtenir M = 3 de sorte à pouvoir détecter et corriger une erreur sur un bit. Pour ce faire, un nombre binaire C de longueur p est créé. Dans notre cas, ce nombre est formé des bits

c_1 = parité(b_1, b_3, b_5, b_7)
c_2 = parité(b_2, b_3, b_6, b_7)
c_3 = parité(b_4, b_5, b_6, b_7)

Le nombre $C = c_3c_2c_1$ donne l'index *j* du bit qui a été altéré. Si C = 0, on présume qu'aucun bit n'a été modifié. Si C est différent de 0, on inverse le bit d'index *j* = C.

Reprenons l'exemple précédant. Nous voulons envoyer le mot 0110. Pour cela, on forme le mot 0110011. Le récepteur reçoit 0100011. Le récepteur forme le nombre C en calculant la valeur des bits :

c_1 = parité(b_1, b_3, b_5, b_7) = parité(1, 0, 0, 0) =1
c_2 = parité(b_2, b_3, b_6, b_7) = parité(1, 0, 1, 0) =0
c_3 = parité(b_4, b_5, b_6, b_7) = parité(0, 0, 1, 0) =1

$C = 101_{(2)} = 5_{(10)}$. On inverse b_5 et on trouve : 0110011 qui est le mot envoyé par l'émetteur.