CODAGE / DECODAGE CONVOLUTIF

Au menu :

Allocation dynamique, Lecture/écriture de fichiers,
Utilisation de librairies

Présentation du projet :

On propose d'écrire un programme de codage/bruitage/décodage de signal, permettant de simuler un système de communication. Les fonctions de codage/décodage et ajout de bruit sont déjà écrites.

Partant d'un signal quelconque (image, document texte, archive, son…), on code ce signal avant de le transmettre, puis on le décode. On simulera la transmission du signal en ajoutant du bruit sur le signal codé. Le décodage consiste à retrouver le message d'origine en ayant enlevé le bruit (voir le système de communication ci-dessous).

Le travail à faire est décrit dans les deux parties (A et B) ci après.

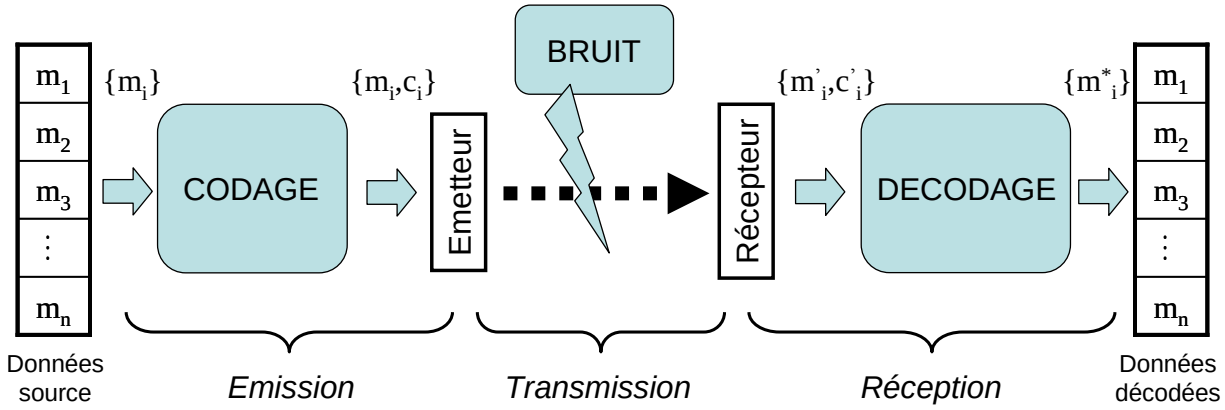


Figure 1 : Système de communication.

Le message original $\{\boldsymbol{m_i}\}$ de taille ***n*** bits (***n/8=N*** octets) est codé grâce au codeur convolutif décrit en annexe. Le signal sortie du codeur, noté $\{\boldsymbol{m_i, c_i}\}$, est de taille 2*(N+1) et est constitué d'une alternance d'un bit d'information ($\boldsymbol{m_i}$) et d'un bit de contrôle ($\boldsymbol{c_i}$) calculé par le codeur. Le signal $\{\boldsymbol{m_i, c_i}\}$ est ensuite émis.

Lors de la transmission un bruit additif perturbe le signal. A la réception, on reçoit le signal bruité, noté $\{\boldsymbol{m'_i, c'_i}\}$. Le décodeur est chargé de retrouver le message original à partir du signal reçu. Si le bruit respecte certaines conditions, le message décodé $\{\boldsymbol{m^*_i}\}$ est égal au message d'origine $\{\boldsymbol{m_i}\}$.

Les fonctions de codage, décodage et de bruit ne sont pas à faire.

➔ ***Chemin réseau vers les ressources du TD***

/home/gehp1/profs/tgrenier/TD_C/TD_C_7_8/

➔ ***Documentation des fonctions :***

/home/gehp1/profs/tgrenier/TD_C/TD_C_7_8/doc/index.html

PARTIE A : Mise en place du projet

1) Écrire la fonction AffichageHexa

Cette fonction (à utiliser abondamment !) affiche en hexadécimal le contenu d'un tableau de caractère (unsigned char). La taille du tableau sera passée en paramètre à la fonction. Une chaine de caractère sera aussi passée à la fonction, elle permettra d'identifier le tableau affiché.

Exemple d'utilisation de la fonction :

- Appel de la fonction *AfficheHexa* :

```
void main()
{
unsigned char message[]={0x0F, 0x44, 0x0F}; // initialisation
AfficheHexa( message, 3, "message"); // on passe une chaine de
                    // caractères correspondant au nom du tableau
}
```

- Affichage obtenu :

```
message [3] : f 44 f
```

Remarque : `cout << hex << int(a);` permet d'afficher la variable *a* en hexadécimal.

2) Ajout des librairies au projet

Les fonctions liées au codage ont déjà été écrites et compilées sous forme d'une librairie partagée. Le but de cette question est de modifier le projet KDevelop précédent pour utiliser ces fonctions.

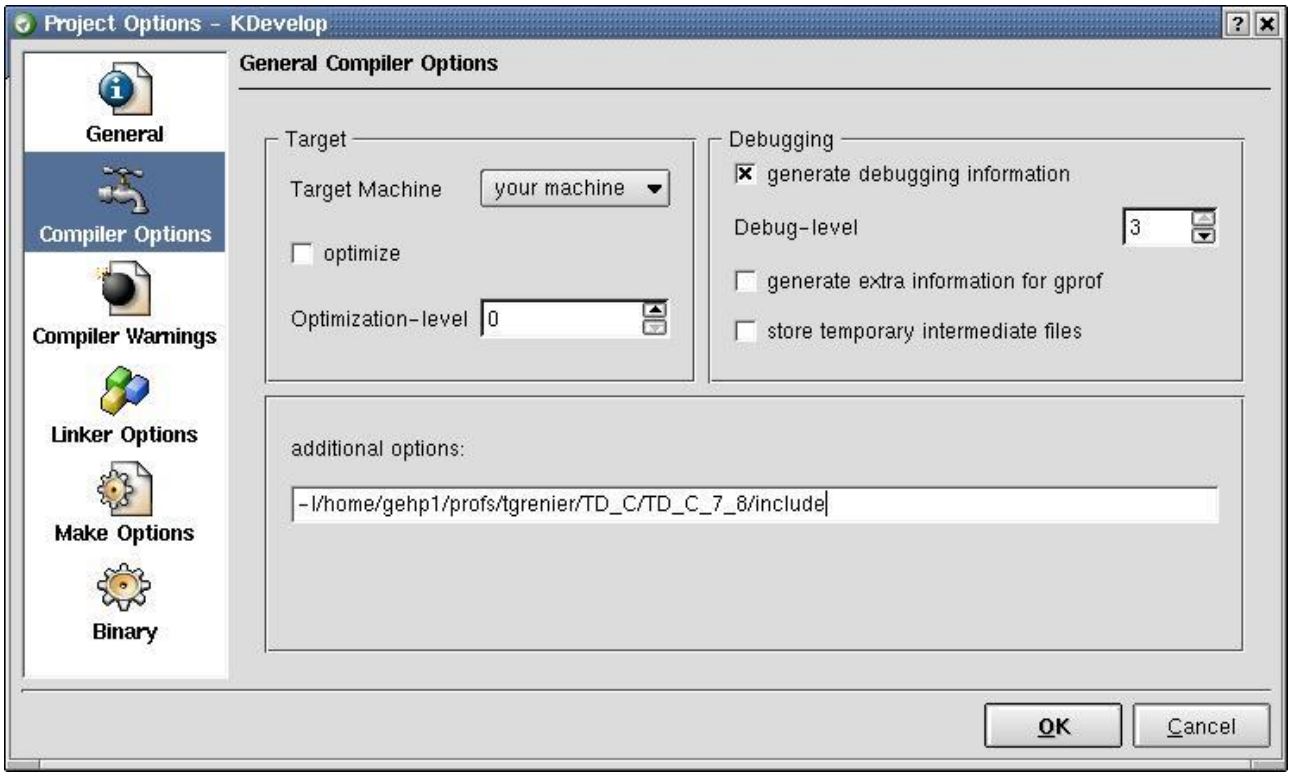
Il y a deux choses à ajouter :

1) **le chemin aux fichiers .h** inclus dans votre main ; nécessaire au compilateur pour inclure les fichiers .h au début du `main` et pour compiler votre code (vérifier la bonne utilisation des fonctions incluses).
2) **le chemin de la librairie et le nom de la librairie utilisée** : nécessaire à l'éditeur de lien pour la création de l'exécutable.

Dans KDevelop, ces 2 ajouts sont à faire dans les options du projet (*Project->Option* ou F7).

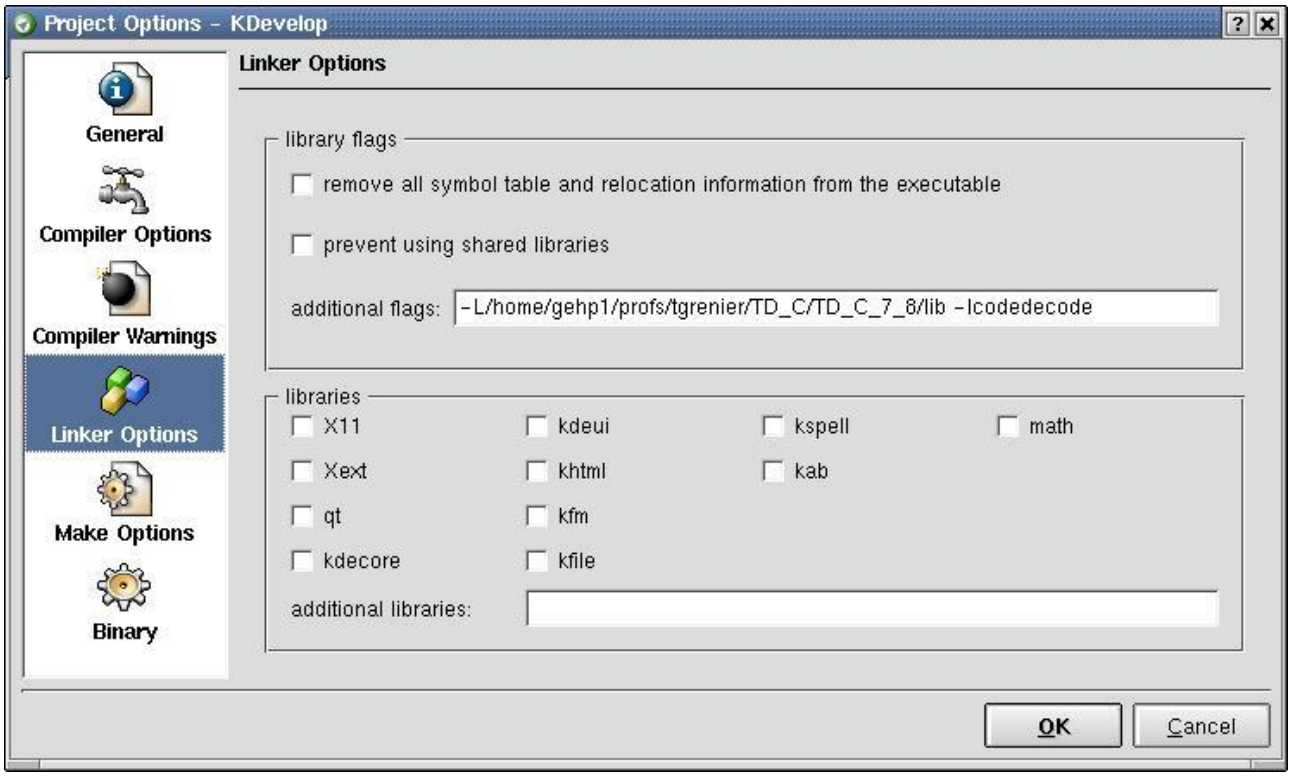
Pour le chemin aux fichiers *.h*, ajouter la ligne suivante dans *Compiler Options* (respecter la casse et les espacements !) :

`-I/home/gehp1/profs/tgrenier/TD_C/TD_C_7_8/include`



Pour le chemin et le nom de la librairie, ajouter la ligne suivante dans *Linker Options* (respecter la casse et les espacements !) :

`-L/home/gehp1/profs/tgrenier/TD_C/TD_C_7_8/lib -lcodedecode`



3) Programme statique

En vous appuyant sur la documentation des fonctions de codage/décodage/bruitage (cf. Annexes) écrire un programme permettant de coder, bruiter puis décoder le contenu d'un tableau statique.

On veillera à respecter les tailles des différents tableaux.

Le programme devra afficher les différents tableaux (notamment le message initial, le message bruité et le message décodé).

Remarque : Pour compiler et construire ce programme, utiliser le projet fait à la question précédente !

PARTIE B : Allocation dynamique et fichier

4) Allocation dynamique

Modifier le programme de la ***question 2*** pour que les allocations soient dynamiques. Les allocations (`malloc`) se font dans la fonction `main`. Ne pas oublier la libération (`free`) des espaces mémoires alloués.

Rappel allocation dynamique :

```
void main()
        {
        unsigned char *tab; // déclaration d'un pointeur
        int taille ;
        cin >> taille ; // demande de la taille
        ...
        tab = new unsigned char[ taille ]; // Allocation dynamique
        ...    // utilisation du tableau
        delete[] tab ; // libération de l'espace mémoire
        ...
        }
```

5) Lecture et d'écriture de fichiers

Dans la librairie deux fonctions d'accès aux fichiers sont écrites :

- **`LireFichier`** : prend en argument une chaine de caractère correspondant au fichier à lire, un tableau de caractères (non signés) lus et la taille du tableau. Le tableau est alloué dynamiquement par cette fonction.
- **`EcrireFichier`** : prend en argument une chaine de caractère correspondant au fichier à écrire, le tableau de caractères (non signés) à écrire et la taille du tableau.

Pour utiliser ces fonctions ajouter :

```
#include "fonctions_fichier.h"
```

➔ Modifier votre programme pour faire une application conviviale permettant de :

- Lire un fichier (signal ou message),
- Enregistrer un signal (après codage),
- Bruiter un signal,
- Enregistrer un message (bruité ou décodé).

On pourra ajouter des fonctions permettant de quantifier le taux de bruitage (nb octets faux/ nb octets total) et le taux de décodage (nb octet modifier par le décodage / nb total d'octets)

6) Tests sur fichiers

Deux exemples de signaux bruités sont donnés :

- Fleur2_signal_bruit.bmp.sgn : le message est une image bmp (bruit : 11% des bits).
- Output_signal_bruit.wav.sgn : le message est un son wav (bruit : 8% des bits).

➔ Comparer qualitativement et quantitativement les messages obtenus. Pour lire les fichiers wav, on trouvera des outils multimédia dans le menu des applications gnome et KDE (xmms…).

Annexes

1- Liste des fonctions :

Détails sur */home/gehp1/profs/tgrenier/TD_C/TD_C_7_8/doc/index.html*

int	**GetTailleInformationControlFromMessage** (int taille_message) *Fonction permettant de déterminer la taille des tableaux "information" et "controle" à partir d'un message de taille taille_message.*
int	**GetTailleMessageFromInformationControl** (int taille_information_control) *Fonction permettant de déterminer la taille du message obtenu à partir deux tableaux "information" et "contrôle" de même taille taille_information_control.*
int	**GetTailleInformationControlFromSignal** (int taille_signal) *Fonction permettant de déterminer la taille des tableaux "information" et "controle" à partir d'un signal recu de taille taille_signal.*
int	**GetTailleSignalFromInformationControl** (int taille_information_control) *Fonction permettant de déterminer la taille du signal à transmettre à partir deux tableaux "information" et "contrôle" de même taille taille_information_control.*
void	**CodageConvolutif** (const unsigned char message[], unsigned char information[], unsigned char control[], int taille_message) *Fonction de codage convolutif systématique.*
void	**DecodageConvolutif** (const unsigned char information[], const unsigned char control[], unsigned char message[], int taille_information_control) *Fonction de decodage convolutif systèmatique.*
void	**MelangeInformationControl** (const unsigned char information[], const unsigned char control[], unsigned char signal[], int taille_information_control) *Melange binaire de l'information et du control pour faire le signal.*
void	**SepareInformationControl** (const unsigned char signal_recu[], unsigned char information[], unsigned char control[], int taille_signal) *Séparation binaire de l'information et du control à partir du signal.*
void	**BruitAdditif** (const unsigned char signal[], unsigned char signal_bruit[], int taille_signal, double p, int offset) *Fonction de bruitage du signal (bruit uniforme). "p" est le pourcentage d'octets bruités. "offset" permet de commencer le bruitage qu'à partir de l'octet "offset".*
int	**LireFichier** (const char fichier[], unsigned char **message, int *taille, int offset) *Fonction de lecture du fichier. le tableau message est alloué par cette fonction !!!*
int	**EcrireFichier** (const char fichier[], const unsigned char message[], int taille) *Fonction d'écriture de fichier.*

2- Schéma du codeur :

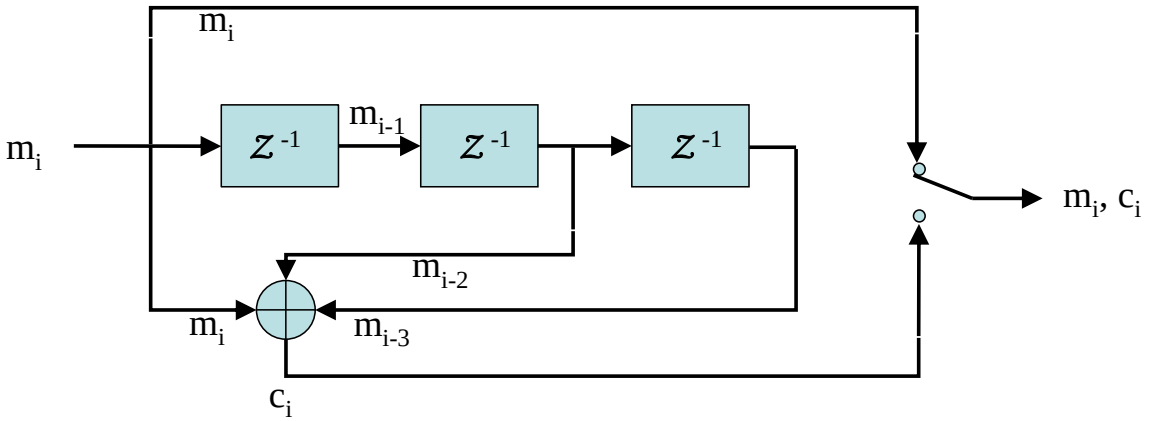


Figure 2 : Codeur convolutif systématique.

Les rectangles $\mathcal{Z}^{-1}$ correspondent aux bascules (initialement à 0) ou fonction retard. L'addition est une addition modulo 2 (réalisée avec des *ou exclusifs*).

Ce codeur est dit systématique car les bits du message sont directement copiés dans les bits d'information m_i signal de sortie. Les bits de contrôles c_i contiennent la redondance.

3- Schéma du décodeur :

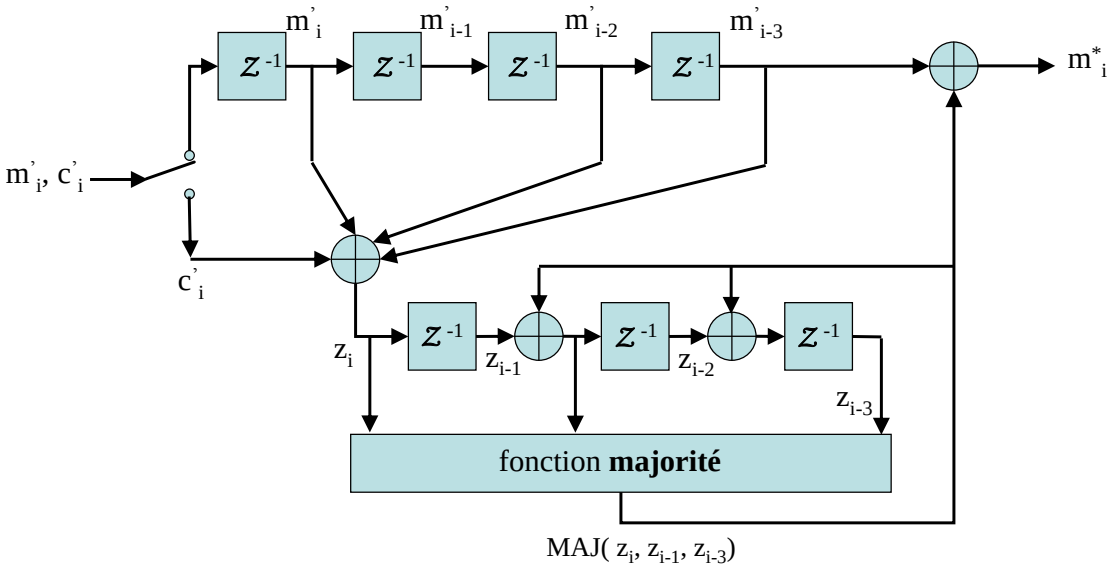


Figure 3 : Décodeur convolutif à fonction majorité.

Le décodeur fonctionne de manière itérative en deux étapes. Initialement, toutes les entrées et sorties des bascules $\mathcal{Z}^{-1}$ sont à zéro, ainsi que la sortie de la fonction majorité.

- La première se réalise quand le commutateur passe en position « haute » : le $i^{ième}$ bit d'information $\boldsymbol{m'_i}$ est envoyé sur la première bascule. Ensuite, toutes les bascules $\mathcal{Z}^{-1}$ sont mises à jour (coup d'horloge) : leur valeur en entrée est copiée en sortie.

- La seconde étape correspond au passage à la position « basse » du commutateur. Le bit $\boldsymbol{c'_i}$ est appliqué sur la quatrième entrée du sommateur modulo 2. La somme $\boldsymbol{z_i}$ est alors calculée ainsi que la valeur de la fonction majorité. Cette fonction retourne la valeur (0 ou 1) majoritairement présente

sur ces entrées (ici : z_i , z_{i-1} et z_{i-3}). Le décodage du i^{ième} bit du signal peut alors se faire en calculant la somme modulo 2 (« ou exclusif ») entre la valeur en sortie de m_{i-3} et de la fonction majorité. On remarquera sur l'exemple donné plus bas qu'il s'agit du (i-3)^{ième} bit du message…

La Figure 4 représente la table de vérité du décodage du signal reçu (bruité) : 0x0D 0x46 0x0F. Après séparation des bits d'information et de contrôle, l'octet d'information vaut 0x21 et celui de contrôle 0x3A. A l'origine, l'information valait 0x30, les bits de contrôle ont été calculés pour cette valeur d'information. Le bruit a modifié les valeurs. Le décodeur doit retrouver la valeur 0x30 pour l'information.

	t	1	2	3	4	5	6	7	8	9	10	11	12
0x21 →	m_i	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**
	m_{i-1}	0	0	0	1	0	0	0	0	1	0	0	1
	m_{i-2}	0	0	0	0	1	0	0	0	0	1	0	0
←	m_{i-3}	0	0	0	0	0	1	0	0	0	0	1	0
0x3A →	c_i	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**
	z_i	0	0	0	1	0	1	1	1	0	1	1	0
	z_{i-1}	0	0	0	0	1	0	1	**0**	1	0	1	**0**
	z_{i-2}	0	0	0	0	0	1	0	**0**	0	1	0	**0**
	z_{i-3}	0	0	0	0	0	0	1	0	0	0	1	0
←	**maj**	0	0	0	0	0	0	1	0	0	0	1	0
→	m^*_i	0	0	0	0	0	1	**1**	0	0	0	**0**	0

0x30

Figure 4 : Exemple du décodage d'un signal bruité.

4- Fonction SépareInformation, explication

A partir d'un signal reçu (tableau de caractères non signés), cette fonction extrait dans **deux tableaux différents** les **bits** d'information et de contrôle. La taille du signal est passée en paramètre. La taille commune aux tableaux d'information et de contrôle sera retournée par la fonction. Le principe de séparation est décrit ci-dessous pour 1 octet (8 bits) de signal. On forme ainsi qu'un demi-octet d'information et un demi-octet de contrôle, un second octet de signal est nécessaire pour obtenir un octet complet d'information et de contrôle (cf . Figure 6).

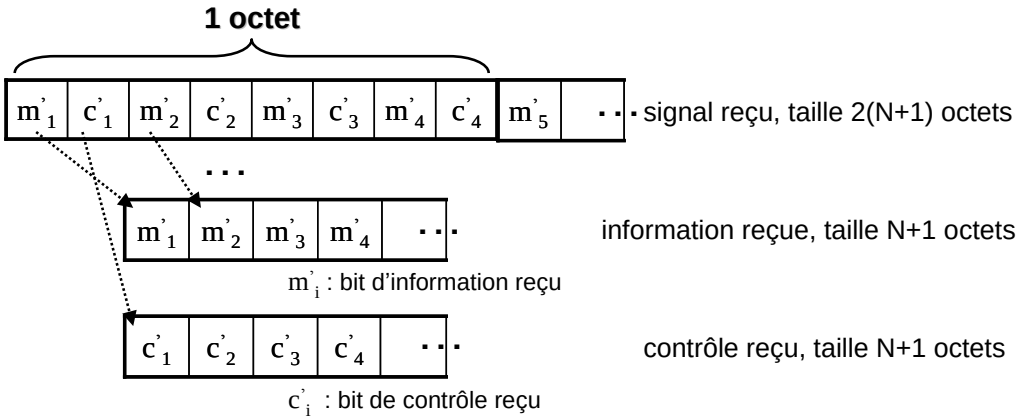


Figure 5 : Principe de la séparation des bits d'information/contrôle du signal reçu.

Il faut noter que *SepareInformationControle* (comme les fonctions de codage/décodage/bruitage) est une fonction qui travaille sur les **bits** des octets des tableaux signal, information et contrôle.

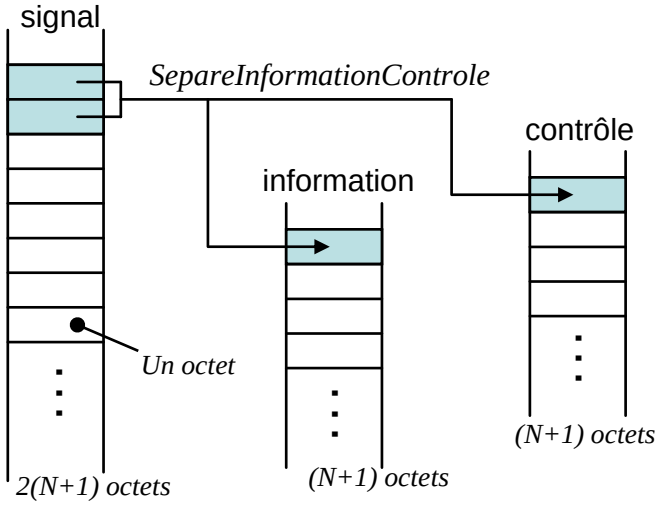


Figure 6: Détails, au niveau octet, du fonctionnement de la fonction SepareInformationControle.