Le contrôle CRC	
Historique des versions	03/05/2004 : Création du document 17/05/2004 : Rajout des références

Sommaire

1 Introduction

Ce document consiste à expliquer l'intérêt et le fonctionnement du contrôle CRC.

Le CRC (Cyclic Redundancy Code) est un code permettant de vérifier l'intégrité d'un mot formé de plusieurs bits.

Pourquoi vérifier cette intégrité ? Beaucoup de système, notamment informatique, utilisent le transfert de données. Par exemple, un réseau informatique : il s'agit d'envoyer des données d'un point A à un point B. Celles ci sont transférées par « paquet » de plusieurs bits qui renferment l'adresse de l'émetteur, l'adresse du destinataire et bien sûr, les données elles mêmes. Mais comment s'assurer, une fois réceptionnées, que ces données sont conformes à celles qui ont été envoyées ? Plusieurs solutions existent.

La première et la plus simple serait de ré envoyer la trame de données, cependant, cela est bien trop coûteux en taille.

D'autres méthodes consistent à rajouter quelques bits de vérification à la trame. Si ces bits sont le résultat d'une fonction mathématique appliqué aux données, il suffit d'appliquer une fonction inverse à la réception pour déterminer si oui ou non la trame est bien la même que celle envoyé. C'est cette méthode qui nous intéresse ici. Une technique simple est d'utiliser un bit de parité (1 si la trame est paire, 0 si elle est impaire); voilà une méthode peu coûteuse en place mais pas forcément très sûre. De manière un peu plus efficace on peut se rajouter un « cheksum » qui n'est autre que la somme des bits constituant de la trame de données. C'est une solution plus coûteuse mais qui permet de détecter déjà beaucoup d'erreurs.

Nous arrivons maintenant à notre méthode qui consiste à calculer un code de redondance cyclique à partir de la trame des données. Ce code se détermine par des calculs relativement simple basés sur l'arithmétique modulaire. Nous verrons plus tard que la qualité de ses résultats est fonction d'un polynôme dit « générateur ». Selon ce dernier, on peut obtenir des probabilités d'intégrité de la trame proche des 100%.

2 Rappel sur l'arithmétique sur les nombres binaire

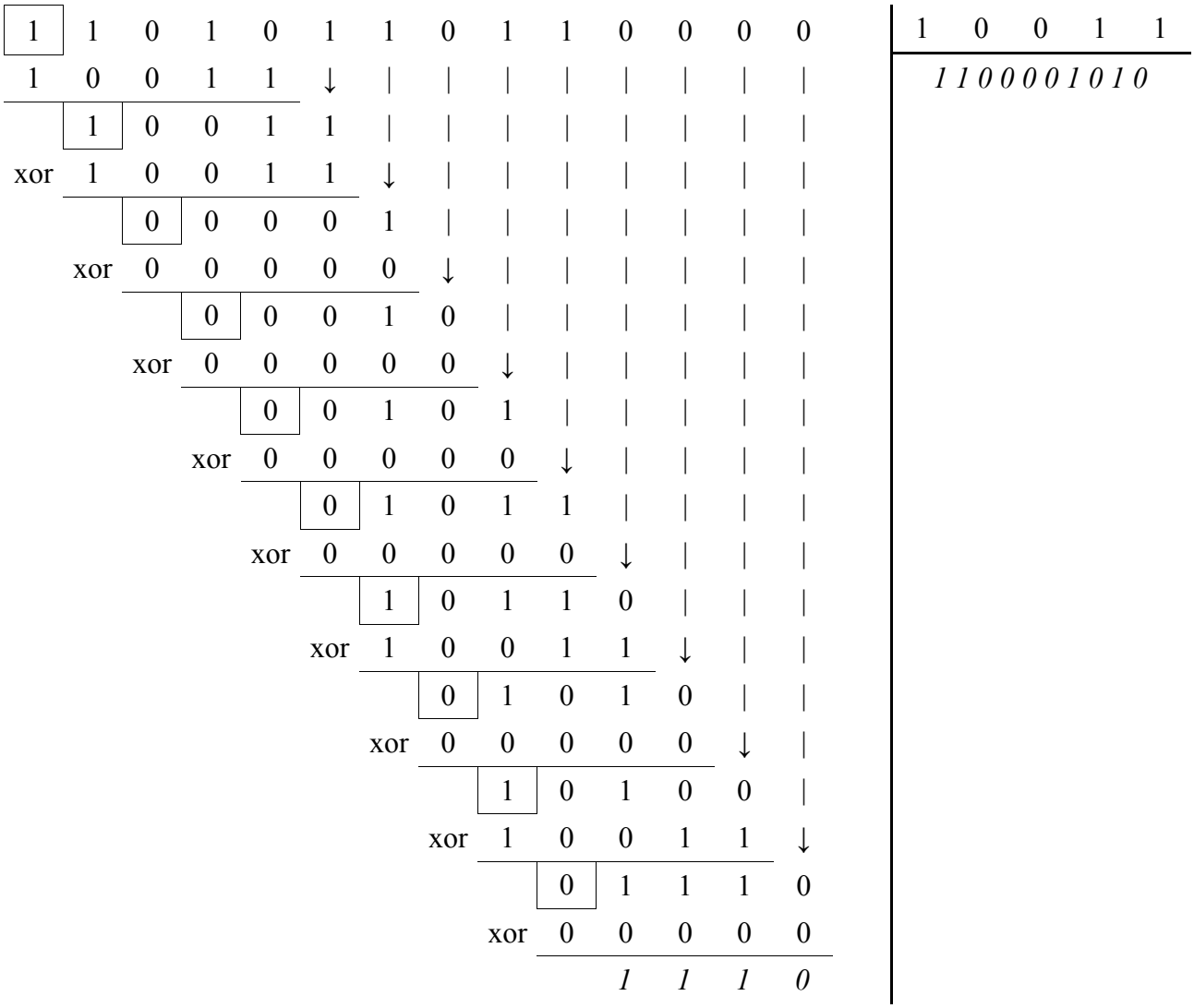
Je ne m'attarderai ni sur les définitions mathématiques, ni sur la théorie de l'arithmétique modulo 2. Cependant quelques petites connaissances dans ce domaine s'impose.

2.1 L'opérateur OU exclusif (XOR)

Cet opérateur est très utilisé dans l'algorithme de création du CRC, en voici ces caractéristiques :

A	*B*	*A* ***xor*** *B*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

2.2 La division entre 2 nombre binaires



Pour ce faire, on sélectionne le bit de poids fort du dividende, ici il vaut 1, il sera le bit de poids fort du quotient. Maintenant, on peut faire (1 x diviseur) = 1 0011. On réalise ensuite un XOR entre le dividende et ce résultat qui est décalé de suffisamment de bit pour commencer au début du dividende. Ensuite, on ajoute au résultat de l'opération le bit suivant correspondant au dividende (on le fait « descendre »).

L'opération s'enchaîne ainsi de suite. On sélectionne le bit de poids fort du résultat précédemment trouvé, ici, c'est un 1. On lui applique donc un XOR avec 1 0011. Le résultat sera 0000 auquel on fera descendre 1, ce qui donne 00001. Ici, le bit de poids fort est un 0, donc (0 x diviseur) = 00000. On refait l'opération avec le XOR.

Ces opérations seront réalisées jusqu'à ce qu'on ne puisse plus « descendre » de bit depuis le dividende, autrement dit, on réalise (taille_du_dividende – taille_du_diviseur + 1) opérations (la taille correspondant au nombre de bits).

3 Fonctionnement

Le calcul va utiliser le principe de la division polynomiale. Ainsi, on va diviser la trame d'entrée A par un polynôme et donc récupérer le CRC. A l'arrivée on redivisera la trame B complète (A+CRC) par ce même polynôme. Si le reste est nul, c'est que la trame de départ est la même que la trame d'arrivée.

Le choix du polynôme générateur est fonction de la qualité du résultat du CRC. Sans rentrer dans les détails mathématiques, le plus simple est d'utiliser ceux qui sont définie comme étant de bonne qualité. Les CRC-16 et 32 donne une fiabilité de près de 100% au niveau de l'intégrité de la trame.

Exemples :

Nom	Polynôme générateur
CRC-4	$x^4 + x^2 + x^1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$
CRC-16 SDLC (CCITT)	$x^{16} + x^{12} + x^5 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-16 Reverse	$x^{16} + x^{14} + x^1 + 1$
SDLC Reverse	$x^{16} + x^{11} + x^4 + 1$
CRC-32 (ethernet)	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$

L'algorithme de base qui renvoi la trame d'origine suivit de son CRC est le suivant :

```
Début
      Décaler, vers la gauche, du degré du polynôme générateur le mot d'entrée
      Faire la division (mot d'entrée / polynôme générateur)
      Faire le calcul (mot d'entrée XOR polynôme générateur)
Fin
```

L'algorithme qu'il faut appliquer à une trame reçu (contenant un CRC) est le suivant :

```
Début
      Faire la division (trame reçu / polynôme générateur)
      Si le reste est nul Alors la trame est correcte
      Sinon retransférer la trame
Fin
```

Voici un exemple de recherche d'un CRC (il réutilise l'exemple de la division vu plus haut). On veut transférer le mot : 11 0101 1011. On décide d'utiliser le polynôme générateur de degré 4 (le degré étant le nombre de bit – 1); ici 10011.

D'abord on décale 11 0101 1011 de 4 rang vers la gauche, résultat : 11 0101 1011 0000.

Ensuite, on cherche le reste de la division de 11 0101 1011 0000 par 10011, résultat : 1110 (c'est le FCS pour **F**rame **C**ontrol **S**equence, nommé par abus de langage CRC).

Enfin on applique un XOR entre le mot d'entrée (décalé de 4 rang) et le reste de la division, 11 0101 1011 0000 XOR 1110 = 11 0101 1011 1110, ce qui correspond bien au mot de départ « suivi » du CRC.

On transfert cette trame. Si on la récupère telle quelle, on obtient 11 0101 1011 1110. On divise ceci par 10011, et on trouve un reste nul, le transfert s'est donc correctement déroulé, aucune perte de données n'est à déplorer.

4 Implémentation en C

Voici une implémentation en langage C du problème. On trouve 3 fonctions :

- ✔ `int get_code_crc(int mot, int polynome_generateur)` qui renvoie le mot passé en paramètre suivit de son CRC a partir du polynôme générateur (en hexadécimal).
- ✔ `int reste_division_ou_exclusif(int dividende, int diviseur)` qui renvoie le reste de la division du dividende par le diviseur.
- ✔ `int degre_int(int mot)` qui renvoie le degré d'un mot binaire (max 32 bits).

```
//-------------------------------------------------------------------------
// Recupere le nombre de bit (degre) d'un mot (short int)
int degre_int(int mot)
{
      int degre = 0;
      int i     = 0;

      for (i = 0; i < (sizeof(int)*8); i++)
      {
            if ((mot & 0x0001) == 1) degre = i+1;
            mot = mot >> 1;
      }

      return degre;
}

//-------------------------------------------------------------------------
// Donne le reste de la division entre 2 nombre binaire (division utilisant
// les OU exclusifs)
int reste_division_ou_exclusif(int dividende, int diviseur)
{
      int i          = 0;
      int tdividende = 0;
      int tdiviseur  = 0;
      int div_interm = 0; // diviseur intermedaire
      int reste      = 0;

      // recherche des degres des operandes
      tdividende = degre_int(dividende);
      tdiviseur = degre_int(diviseur);

      // recherche du reste de la division
      div_interm = diviseur;
      reste = dividende >> (tdividende - tdiviseur);

      for (i = (tdividende - tdiviseur+1); i > 0; i--)
      {
            reste = reste ^ div_interm;

            if (i > 1)
            {
                  reste = reste << 1;
                  reste = reste | ((dividende >> (i-2)) & 0x0001);
                  if (((reste >> (tdiviseur-1)) & 0x0001) == 0) div_interm = 0;
                  else div_interm = diviseur;
            }
      }

      return reste;
}
```

```
//-----------------------------------------------------------------------------
// A partir d'un mot binaire (short int), renvoie ce mot suivit de son CRC
// defini par un polynome generateur (ex : 0x15 = X4 + X2 + 1)
int get_code_crc(int mot, int polynome_generateur)
{
      int crc = 0;

      // decaler le mot vers la gauche du degre-1 du polynome generateur
      mot = mot << (degre_int(polynome_generateur)-1);

      // chercher le reste de la division entre le mot et le polynome generateur
      crc = reste_division_ou_exclusif(mot, polynome_generateur);
      // concatenation du mot et du polynome generateur
      mot = mot ^ crc;

      return mot;
}
```

En reprenant l'exemple précédent, voici comment utiliser ce programme :

```
int main (int argc, char *argv[])
{
      int code = get_code_crc(0x35B, 0x15);
      printf("code + crc : %X  \n", code);


      return 0;
}
```

5 Références

- ✔ Le contrôle CRC, un article de Sylvain Nahas
  http://docs.sylvain-nahas.com/crc.html
- ✔ Le pseudo code de la recherche d'un CRC (en anglais)
  http://www.ibrtses.com/delphi/dcrc.html
- ✔ AMS Standard Cyclic Redundancy Check - CRC16 -Status: Baseline
  http://ams.cern.ch/AMS/Dataformats/node26.html