CRC doda Ci pewności, część 2

Kontynuujemy omawianie metody obliczania CRC (Cyclic Redundancy Codes). W pierwszej części artykułu zapoznaliśmy się z podstawami arytmetyki stosowanej do tego celu. Bazując na tej wiedzy, postaramy się uchwycić istotę metody obliczania CRC.

Po sporej dawce teorii z poprzedniego odcinka, Czytelnicy zapewne spragnieni są bardziej praktycznych ćwiczeń. Wiemy jednak, że nie od razu Kraków zbudowano. Nie uda nam się jeszcze stworzyć gotowej procedury, którą będzie można wykorzystać we własnym programie, jakkolwiek mam nadzieję, że przykłady zawarte w tym odcinku znacznie zbliżą nas do celu.

Przed przystąpieniem do kolejnych ćwiczeń proponuję szybciutko przypomnieć sobie samodzielnie na czym polega dzielenie w arytmetyce CRC. W dalszych rozważaniach dzielnik ilorazu będziemy nazywać *wielomianem generującym* lub krótko *generatorem*. W terminologii angielskiej stosuje się również określenie *poly*. Jego znaczenie jest bardzo istotne we wszystkich algorytmach obliczających CRC. Jak się okaże, szczególnie ważny będzie stopień tego wielomianu. Jest on określony przez pozycję najstarszego współczynnika o wartości 1. Na przykład wielomian 10011 jest stopnia 4 (ozn. W=4), nie 5, bo pozycja najstarszej jedynki jest równa 4 (liczymy od zera). Będziemy go używać w dalszych obliczeniach. W praktyce wykorzystuje się najczęściej wielomiany stopnia 16 lub 32, co ułatwia implementację algorytmów w programach komputerowych.

Mając wybrany generator spróbujmy jeszcze raz podzielić przez niego wielomian reprezentujący pewną transmitowaną wiadomość (obliczenia w arytmetyce CRC). Przed przystąpieniem do obliczeń zastosujemy jednak mały trik polegający na dopisaniu na końcu wiadomości takiej liczby zer, która odpowiada stopniowi generatora, czyli w naszym przypadku 4:
wiadomość oryginalna: 1101011011
generator: 10011
wiadomość przygotowana do obliczeń: 11010110110000

Wykonajmy teraz dzielenie (kropki ułatwiają podpisywanie cyfr):

```
   1100001010
--------------
11010110110000:10011
10011.........
------........
=10011........
 10011........
 ----------...
 =====10110...
     10011...
     -----.
   ==10100.
     10011.
     -----
     ==1110  Reszta z dzielenia
             (suma kontrolna)
```

Sam wynik działania (iloraz) jest dla naszych celów zupełnie nieistotny, możemy go całkowicie zignorować. Najważniejsza jest reszta z dzielenia stanowiąca obliczoną sumę kontrolną. Zazwyczaj jest ona zapisywana do przesyłanej wiadomości (tak jak w powyższym przykładzie zrobiliśmy z zerami) i utworzony w ten sposób zmodyfikowany komunikat jest następnie transmitowany. W omawianym przypadku wysyłana wiadomość miałaby postać: 11010110111110. Na drugim końcu toru transmisyjnego odbiornik ma do wyboru dwie możliwości:

1. Odseparować wiadomość od sumy kontrolnej. Obliczyć sumę kontrolną po wcześniejszym dopisaniu W zer na końcu wiadomości, a następnie porównać obie sumy.

2. Obliczać sumę w biegu (bez dopisywania zer) i sprawdzić, czy obliczona suma będzie równa zero.

Obie powyższe metody są równoważne. W dalszej części artykułu skupimy się na metodzie 2, gdyż z matematycznego punktu widzenia jest nieznacznie prostsza.

Wybór generatora

Wybór odpowiedniego generatora to temat, który nadawałby się bardziej na pracę doktorską niż na krótki artykuł w EP. Rozwiązanie pewnych zagadnień wymagają stosowania niełatwego aparatu matematycznego. Nie będziemy więc wnikać w szczegóły.

Transmitowaną wiadomość oznaczymy literą T. Załóżmy, że T jest wielokrotnością generatora. Zauważmy, że:
- *W* ostatnich bitów wiadomości T to reszta z dzielenia T przez generator,
- po drugie - dodawanie jest w arytmetyce CRC równoważne odejmowaniu, tak więc dodając resztę z dzielenia odkładamy tę wartość do następnego mnożenia. Jeśli teraz przesyłana wiadomość ulegnie przekłamaniu w odbiorniku otrzymamy T⊕E, gdzie E jest wektorem błędu. Zwróćmy uwagę na to, że sumowanie jest prowadzone w arytmetyce CRC (odpowiada np. operacji XOR).

Teraz odebrana informacja poddawana jest operacji dzielenia T⊕E przez G. Podczas gdy T mod G jest równe 0, to (T⊕E) mod G = E mod G. Tak więc rozmiar wybranego przez nas generatora będzie miał istotny wpływ na detekcję błędów. Przyjmijmy na wiarę, że błędy będące wielokrotnością generatora pozostaną nie wykryte. Naszym zadaniem jest znalezienie takiego wielomianu G, aby jego stopień był jak najmniejszy, gwarantując przy tym odpowiednio wysokie prawdopodobieństwo wykrycia błędu w zaszumionym torze transmisyjnym. Popatrzmy teraz z jakimi rodzajami błędów możemy się spotkać.

Błędy pojedyncze

Mówiąc o błędzie pojedynczym mamy na myśli przykładowo E=10000...0000. Taka klasa błędów będzie na pewno wykryta, gdy co najmniej dwa bity generatora G będą równe 1. Gdybyśmy wykonywali mnożenie G polegające (jak pamiętamy) na operacji przesuwania i dodawania pewnej stałej wartości z ustawionym w tym przypadku tylko jednym bitem, nie było by więc możliwe skonstruowanie takiej liczby, w której tylko jeden bit byłby ustawiony. Zawsze pozostaną dwa ostatnie bity.

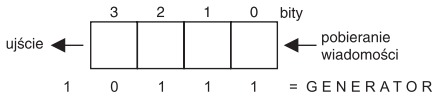
Błędy podwójne

Dla wykrycia wszystkich błędów typu 100...000100...000 (E posiada dwa bity o wartości 1) wybieramy G, które nie będzie wielokrotnością 11, 101, 1001, 10001, 1000001, itd. Przyjmijmy to na wiarę.

Błędy z nieparzystą liczbą bitów

Wykrycie takich błędów jest możliwe przez wybranie G z parzystą liczbę bitów. Zauważmy, że mnożenie CRC jest

Bezpieczna wymiana danych w systemach mikroprocesorowych



Rys. 1

niczym innym, jak wielokrotnym XOR-owaniem pewnej stałej z odpowiednimi przesunięciami. Pamiętamy również, że XOR-owanie to w gruncie rzeczy zwykłe przełączanie bitów. Biorąc to pod uwagę, jeśli będziemy XOR-ować liczbę z parzystą liczbą jedynek zawsze w wyniku obliczeń zostanie nieparzysta liczba jedynek. Na przykład weźmy E=111 i spróbujmy przełączyć wszystkie trzy bity przez powtarzanie XOR-owania z liczbą 11, przesuwając ją po każdym kroku. Otrzymamy E=E XOR 011 (w pierwszym kroku) i E=E XOR 110 (w drugim kroku). Jak widać sztuka się nie udała, jedna jedynka pozostała niezmieniona.

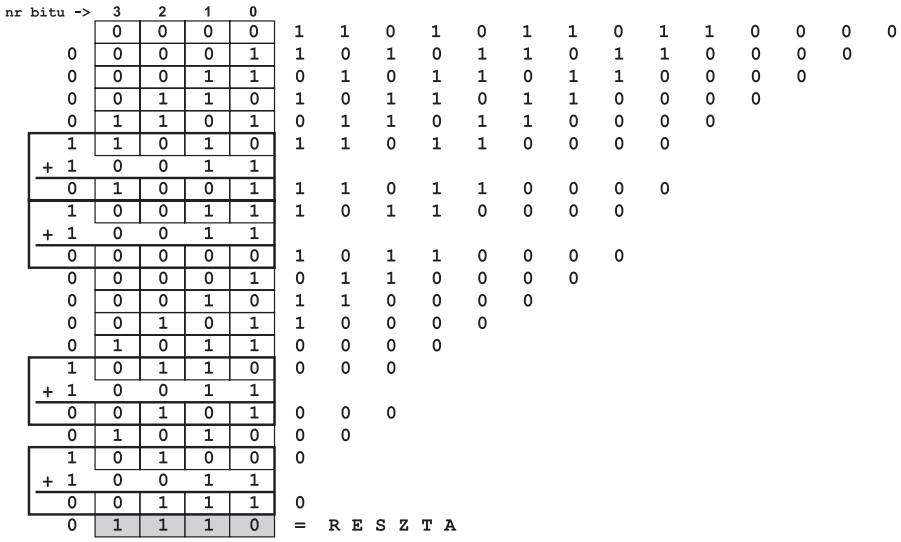
Błędy typu *burst*

Błąd *burst* to ciąg jedynek poprzedzony i zakończony ciągiem zer: E=000...000111...11110000...000. Powyższą zależność można zapisać inaczej: E=(10000...00)(1111111...111), gdzie występuje z zer w lewej części i n jedynek w prawej. Do wykrycia błędu tego typu wystarczy ustawić najmłodszy bit G na 1. Lewa część E nie może być czynnikiem G. Dopóty, dopóki G będzie dłuższe niż prawa część E, błąd będzie wykrywany. Więcej informacji na ten temat można znaleźć w [1].

Biorąc pod uwagę powyższe zależności wybrano kilka generatorów, które są powszechnie stosowane w praktyce:
16-bitowe:
(16,12,5,0) - standard X25
(16,15,2,0) - tzw. CRC-16
32-bitowe:
(32,26,23,22,16,12,11,10,8,7,5,4,2,1,0) - Ethernet

Bezpośrednia implementacja CRC

Ufff, podkład teoretyczny mamy już za sobą, możemy powoli przechodzić do praktyki. Zajmiemy się teraz konkretnymi algorytmami obliczania CRC. Jak wynika z konkluzji poprzedniego podrozdziału, CRC nie jest jednoznacznie wyliczaną sumą kontrolną. W zależności od wymaganego stopnia ochrony danych, stosowanego medium transmisyjnego, możliwości obliczeniowych urządzeń nadawczo-odbiorczych stosuje się różne algorytmy. Na początek zaczniemy od chyba najprostszego z najprostszych, bez wykorzystywania żadnych trików, działającego jednak bardzo wolno. W dalszej części artykułu będziemy go stopniowo komplikować i usprawniać. Pamiętamy o tym, że obliczenia CRC opierają się na operacji dzielenia. Niektóre procesory uwzględniają w swojej liście rozkazów to działanie. Nie będzie ono jednak wykorzystywane, gdyż jak pamiętamy nasze dzielenie jest oparte nie na arytmetyce klasycznej, lecz na arytmetyce CRC. Ponadto dzielna, którą jest transmitowany blok, może osiągać bardzo duże rozmiary. W kolejnych przykładach będziemy



Rys. 2

rozpatrywać transmitowaną wiadomość jako ciąg bajtów. Najstarszy bit (MSB - *Most Significant Bit*) znajduje się na pozycji 7 każdego bajtu. Gdybyśmy potraktowali wiadomość jako ciąg bitów, to najstarszy bit wiadomości odpowiadałby pierwszemu bitowi różnemu od zera w pierwszym bajcie, licząc od najstarszej pozycji. Teraz już możemy naszkicować algorytm dzielenia CRC. Dla ustalenia uwagi przyjmiemy, że W=4, a jako generator wybierzemy G=10111. Do wykonania dzielenia będziemy potrzebować 4-bitowego rejestru (**rys. 1**). Pamiętamy oczywiście, że wiadomość kończy W zerowych bitów.

Algorytm będzie więc następujący (zapiszemy go w postaci pseudoprogramu):

```
Wyzeruj wszystkie bity rejestru.
Dopisz W zerowych bitów do
przesyłanej wiadomości.
while(jest więcej bitów do pobrania)
{
 Przesuń rejestr w lewo o jedną
 pozycję, wczytując następny bit
 wiadomości na pozycję 0.
 If(wychodzący bit ma wartość 1)
 {
   Bitwychodzący_Rejestr=
       Bitwychodzący_Rejestr
       XOR
       Generator
 }
}
```

gdzie: wyrażenie *Bitwychodzący_Rejestr* to słowo, w którym najstarszy bit ma wartość bitu wychodzącego z rejestru, a za nim następują bity reprezentujące wartość rejestru.

Po zakończeniu pętli, rejestr zawiera resztę z dzielenia. Uwaga praktyczna: warunek w instrukcji IF można sprawdzać testując najstarszy bit rejestru przed wykonaniem przesunięcia. Powyższy algorytm będziemy nazywać SIMPLE.

Graficzną ilustrację wykonania tego algorytmu przedstawiono na **rys. 2**. Dane odpowiadają przykładowi przedstawionemu na początku artykułu. Mamy więc: T=11010110110000, G=10011, czyli W=4.

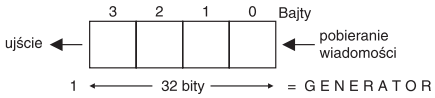
Proponuję teraz samodzielne przećwiczyć algorytm na innym przykładzie. Można przy tym zmienić generator i wartość W, od której algorytm nie zależy.

Implementacja oparta na tablicy

Przedstawiony powyżej algorytm SIMPLE był dość dobrym przykładem bezpośredniego przełożenia teorii w praktykę. Wykorzystując go można już bez trudu napisać program z przeznaczeniem na konkretny procesor. Będzie on oczywiście działał, ale nie zadowoli chyba jego użytkownika. Jego szybkość działania nie będzie największa - zauważmy, że obróbka pojedynczego bitu wymaga wykonania jednego obrotu pętli. Ale „pierwsze koty za płoty“, doświadczenie już jakieś mamy.

Pomyślmy więc co zrobić, żeby zwiększyć wydajność. Pierwszym pomysłem jaki sam się nasuwa będzie przejście z obróbki bitowej na - nazwijmy to na razie - więcej niż bitową. Naturalnymi możliwościami są tu: półbajt (*nibble*) (4 bity), bajt (8 bitów), słowo (16 bitów), długie słowo (32 bity) lub więcej (jeśli będziemy potrafili obsłużyć). My wybierzemy wariant 8-bitowy. Dla niego istnieje już wiele opracowanych algorytmów, ponadto będzie się świetnie nadawał dla małych mikroprocesorów (mikrokontrolerów). Na użytek następnych rozważań zmienimy też dość drastycznie stopień generatora z W=4, na W=32. Pamiętamy, że generator taki będzie miał długość 33 bity (pierwszy bit zawsze równy 1 i 32 bity „aktywne“). Wydłuży się też do 32 bitów rejestr stosowany do obliczeń (**rys. 3**).

Nazwijmy teraz bity najstarszego bajtu (numer 3) naszego rejestru. Będą to t7 (MSB) t6...t0 (LSB). Przyjmijmy też, że 8 najstarszych „aktywnych“ bitów generatora będzie miało oznaczenia: g7



Rys. 3

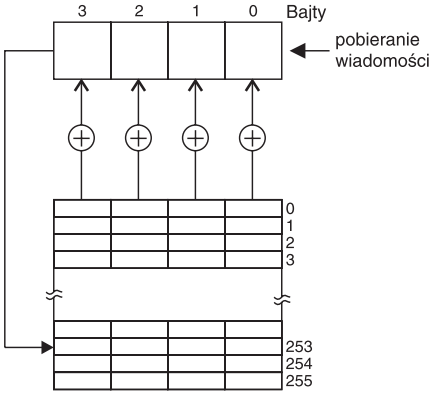
g6...g0. Tak jak było w algorytmie SIMPLE, bit t7 (nazwijmy go bitem szczytowym) będzie określał, czy generator ma być XOR-owany z rejestrem w następnej iteracji. Nastąpi to, gdy bit ten będzie miał wartość 1. Można więc zapisać, że najstarszy bit w następnej iteracji będzie miał wartość: t6⊕t7 * g7, co wyjaśnia poniższa interpretacja pisemna:

```
        t6 t5 t4 t3 t2 t1 t0 ??
⊕       t7 * (g7 g6 g5 g4 g3 g2 g1 g0)
```

Zauważmy, że do obliczania nowej wartości bitu szczytowego (MSB) drugiej iteracji, potrzebne są dwa najstarsze bity w najstarszym bajcie rejestru. Dla trzeciej iteracji będą to trzy bity (t7, t6 i t5), itd. Ogólnie dla k-tej iteracji potrzebnych jest k bitów rejestru. Wykorzystamy to później. Rozważmy przypadek, w którym będziemy wykorzystywać 8 bitów rejestru do obliczania bitu szczytowego dla następnych 8 iteracji. Załóżmy, że będziemy prowadzić 8 następnych iteracji wykorzystując obliczone wartości (które możemy zapisywać w pojedynczym rejestrze i obracać w celu wyłuskiwania każdego bitu). Znowu musimy zauważyć trzy sytuacje:

- Najstarszy bajt rejestru nie ma teraz znaczenia. Nie jest istotne ile razy i z jakim przesunięciem generator jest XOR-owany dla 8 bitów szczytowych, wszystkie będą przesunięte na zewnątrz podczas następnych 8 iteracji.
- Pozostałe bity będą przesunięte o jedną pozycję w lewo, a bajty z prawej strony będą przesunięte na następną pozycję.
- Na czas operacji, rejestr będzie podlegał serii operacji XOR-owania z bitami sterującymi obliczonymi wcześniej.

Teraz rozpatrzmy efekt XOR-owania



Rys. 4

rejestru stałą wartością z różnymi przesunięciami. Na przykład:

```
      0100010 Rejestr
⊕     0000110
⊕     0001100
⊕     0110000
       -------
      0011000
```

Wynik powyższego działania możemy uzyskać na drodze wielokrotnego XOR-owania poszczególnych składników z rejestrem, lub jednokrotnego wykonania tej operacji z wartością stałą równą sumie (XOR) poszczególnych składników. Łącząc całą zdobytą powyżej wiedzę w całość możemy napisać szkic algorytmu:

```
While(są dane w ciągu wejściowym)
{
 Zapamiętaj tymczasowo najstarszy
 bajt rejestru (będzie to bajt
 sterujący)
 Sumuj cały generator z różnymi
 przesunięciami XOR-ując z rejestrem
 odpowiednio z rejestrem sterującym
 Przesuń rejestr w lewo o jeden
 bajt, wczytując na najmłodszą
 pozycję kolejny bajt wiadomości
 XOR-uj zsumowany generator
 z rejestrem
}
```

Jak na razie nie specjalnie widać, żeby algorytm ten był w czymś lepszy niż SIMPLE. Jeśli jednak głęboko go przemyślimy, dojdziemy do wniosku, że większość obliczeń może być wykonana wcześniej, a wyniki mogą być zapisane w odpowiedniej tablicy. Jeśli wykorzystamy to błyskotliwe spostrzeżenie, algorytm można uprościć do postaci:

```
While(są dane w ciągu wejściowym)
{
 Top = bajt szczytowy rejestru
 Rejestr = (Rejestr << 8) |
    następny_bajt_wiadomości
 Rejestr = Rejestr XOR Tablica[Top]
}
```

Nie twierdzę, by powyższe rozważania były proste. Są jednak ważne, ich zrozumienie pozwala bowiem pojąć ideę konstruowania tablicowych algorytmów obliczania CRC. Graficzną interpretację działania algorytmu przedstawiono na **rys. 4**. Możemy jeszcze raz skomentować ten rysunek opisem słownym:

1. Przesuń rejestr o jeden bajt w lewo, dopisując na najmłodszej pozycji kolejny bajt wiadomości
2. Wykorzystaj wychodzący z rejestru bajt do indeksowania tablicy (256 wartości 32-bitowych (dla generatora o W=32 jaki stosowaliśmy w przykładzie).
3. XOR-uj dane z tablicy z rejestrem.
4. Idź do pkt. 1, jeśli nie wykorzystałeś wszystkich bajtów wiadomości.

Teraz już można zrobić pierwszą przymiarkę do napisania programu w jakimś konkretnym języku. Będzie nim oczywiście C. Fragment takiego programu przedstawiono poniżej:

```
unsigned long r;
unsigned char t;
...
r=0;
while(len--)
{
 t=(unsigned char)((r>>24) & 0xff);
 r=(r<<8) | *pMsg++;
 r^=tab[t];
}
```

Zmienna *len* określa liczbę bajtów wiadomości, **Msg* jest wskaźnikiem na bajt wiadomości, który będzie pobierany do obliczeń, *r* to nasz rejestr, *t* jest zmienną tymczasową, a *tab* obliczoną wcześniej tablicą.

Moc języka C polega na bardzo zwięzłym zapisywaniu wyrażeń (co nie zawsze jest wystarczająco zrozumiałe i przez co język ten nie cieszy się popularnością wśród początkujących programistów), można więc nasze obliczenia uprościć do jednej linijki:

```
r=0;
while(len--)
 r=((r<<8) | *pMsg++)
    ^ tab[(unsigned char)((r>>24)
    & 0xff)];
```

Zastosowany tu algorytm będziemy nazywać TABLICOWYM. Charakteryzuje się on dużą wydajnością, nie wymaga wielkich mocy obliczeniowych procesora. Jego największą wadą jest zajmowanie pamięci przez tablice i to, że dla „postronnego" obserwatora może być zupełnie nieczytelny. Trudno znaleźć w nim całą, nie małą jak się mogliśmy przekonać teorię. Ale to jeszcze nie koniec. Więcej w następnym odcinku.

Jarosław Doliński, AVT
jaroslaw.dolinski@ep.com.pl

[1] Artykuł powstał na podstawie publikacji „A painless guide to CRC error detection algorithms", Ross N. Williams. Można ją znaleźć pod adresem http://www.riccibitti.com/crcguide.htm.
[2] Tanenbaum, A.S., „Computer Networks", Prentice Hall, 1981, ISBN: 0-13-164699-0.