# CRC doda Ci pewności, część 3

***Jesteśmy już po pierwszych przymiarkach do napisania własnego programu liczącego CRC. Cel wydaje się być coraz bliżej, ale drogi jakby zaczynają się rozchodzić. Na szczęście każda jest dobra. Dociekliwie będziemy sprawdzać wszystkie tak, by później każdy mógł wybrać najodpowiedniejszą dla siebie.***

## Metoda maglowanej tablicy

Poprzedni odcinek zakończyliśmy napisaniem jednolinijkowego wyrażenia obliczającego CRC. Jak na początek całkiem nieźle. Niby dopiero zaczynamy, a już potrafimy stworzyć tak zwięzły kod. Przypomnijmy więc:

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

Jest jednak pewien problem. Jak widać powyższa pętla operuje na danych będących kolejnymi bajtami pobieranej wiadomości. Chcąc wykorzystać tak napisany program należy dopisać na końcu wiadomości (dopełnić ją) W/8 bajtów o wartości zero. W praktyce może to stanowić niedogodność lub nie, zależy od konkretnych rozwiązań. Jeśli blok danych jest przechwytywany przez inny fragment programu, możemy się spodziewać nawet sporych kłopotów. Jednym z możliwych rozwiązań będzie dopisanie poniższej linii (będzie ona wykonywana po wyjściu z pętli obliczeniowej z przykładu pokazanego wcześniej):

```
for (i=0; i<W/4; i++)
 r = (r << 8) ^ tab[(r >> 24)
    & 0xFF];
```

Linię tę będzie można później zmodyfikować tak, żeby uniknąć konieczności wczytywania zerowych bajtów lub w sposób jawny dopisywać zera jak wyżej. Dla objaśnienia zamysłu jeszcze raz posłużymy się rysunkiem znanym z części 2, który dla wygody powtarzamy poniżej (**rys. 5**). Zauważmy jeszcze dwa fakty:

*Końcówka* - zerowe bajty (będzie ich W/4) pojawiające się na końcu pobieranej wiadomości będą „wpychane" do rejestru z prawej strony, ale ich zerowa wartość nie będzie miała żadnego wpływu na wynik obliczeń. Wynika to z tego, że jak pamiętamy XOR-owanie z zerami nie zmienia bajtu wejściowego. Tak więc, funkcja podstawowa wyliczająca bajty zerowe generuje wyniki wykorzystywane w kolejnym cyklu obliczeń, co powoduje, że po ich zakończeniu wszystkie przesyłane dane „przejdą" przez rejestr.

*Nagłówek* - jeśli po zainicjowaniu rejestr będzie miał zerową wartość, to cztery początkowe iteracje pętli są równoznaczne z przesunięciem czterech pierwszych bajtów wiadomości. Wynika to z faktu, że pierwsze 32 bity sterujące (pobierane kolejno z rejestru) mając wartość zerową, są całkowicie bez wpływu na wynik XOR-owania. Co więcej, nawet jeśli wartość początkowa rejestru nie jest zerowa, to pierwsze cztery bajty iteracji omawianego algorytmu dadzą również jednoznaczny efekt przesunięcia pierwszych czterech bajtów wiadomości i XOR-owania ich z pewną stałą wartością (będącą funkcją początkowego stanu rejestru).

Powyższe fakty połączone z przemiennością operacji XOR ((A ⊕ B) ⊕ C = A ⊕ (B ⊕ C)) oznaczają, że bajty wiadomości nie wymagają przeskoku o W/4 bajtów rejestru, nie będą więc przez niego przepuszczane. Zamiast tego będą XOR-owane z najstarszym bajtem rejestru zanim zostaną użyte do indeksowania tablicy. Na tej podstawie możemy już zmodyfikować nasz algorytm. Jego graficzna interpretacja jest przedstawiona na **rys. 6**, a komentarz do rysunku poniżej:

1. Przesuń rejestr o jeden bajt w lewo, czytając nowy bajt wiadomości.
2. XOR-uj najstarszy bajt, wysunięty właśnie z rejestru z następnym bajtem wiadomości, otrzymując indeks tablicy (z przedziału od 0 do 255).
3. XOR-uj rejestr z pobraną z tablicy daną.
4. Idź do pkt. 1, jeśli nie wykorzystałeś wszystkich bajtów wiadomości.

Rejestr dla powyższego algorytmu musi być zainicjowany taką samą wartością jak w omawianym poprzednio z tym, że wartość początkowa musi być powtórzona w tablicy czterokrotnie. Jeśli w poprzedniej metodzie były wykorzystywane 0, tak samo należy je stosować w tablicach tworzonych dla nowego algorytmu. Można więc powiedzieć, że obie wersje algorytmów będą takie same, dadzą identyczny wynik. Zapis w języku C będzie więc dobrze nam znany:

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

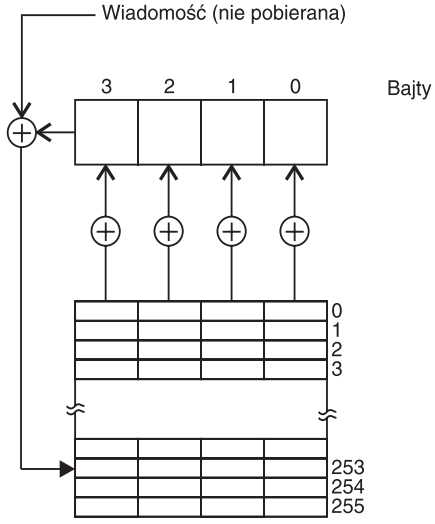
W powyższym kodzie łatwo znajdujemy tablicową implementację obliczania CRC. Maska 0xff może być stosowana i w tym przypadku dla zachowania kompatybilności, jakkolwiek podstawowa pętla wygląda jak wyżej. Zastosowaną tu metodę będziemy nazywać *bezpośrednim algorytmem tablicowym*.

## Odwrócona metoda tablicowa

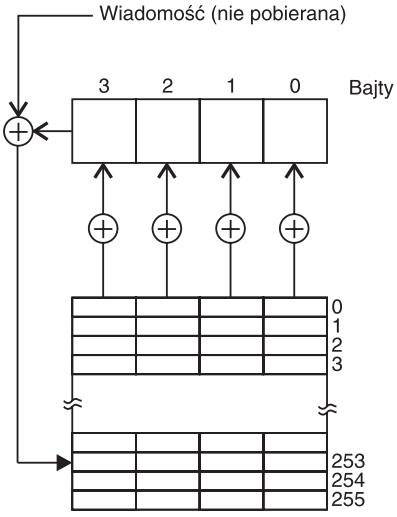
Wydaje się, że powyższa metoda jest już całkowicie zoptymalizowana. Czy na pewno? Zależy jak na to patrzeć. Jeśli uwzględnimy pewne cechy fizycznych układów stosowanych do realizacji transmisji danych, to szybko dojdziemy do wniosku, że nie powiedzieliśmy jeszcze ostatniego słowa. Mimo, że znaleźliśmy się na granicy rozumienia tematu spróbujemy wprowadzić kolejny stopień komplikacji. Potrzebna nam będzie do tego definicja: o pewnej danej (rejestrze) będziemy mówić, że jest odwrócona, jeśli jej bity stanowią odbicie lustrzane względem środka. Na przykład ciąg 0101 jest 4-bitowym odbiciem ciągu 1010. Ciąg 0011 jest odbiciem 1100. I jeszcze nieco bardziej skomplikowany przykład: 0111-0101-1010-1111-0010-0101-1011-1100 to odbicie 0011-1101-1010-0100-1111-0101-1010-1110.

Po co to wszystko? Otóż wprowadzenie odwróconej metody tablicowej może znacznie ułatwić sprzętowe obliczanie CRC w systemach transmisji danych. Jak wiemy większość układów UART wysyła dane w linię począwszy od najmłodszego bitu w bajcie do najstarszego, tymczasem we wcześniejszych rozważaniach zawsze rozpoczynaliśmy analizę od bitu najstarszego. Oczywiście procesor poradzi sobie z takim problemem, i tak będzie odczytywał dane bajtami z bufora UART-u. Są jednak urządzenia, w których kontrola CRC jest realizowana sprzętowo *on-line* przez specjalizowane układy. Odbywa się to na poziomie strumienia danych - bit po bicie. Na potrzeby takich właśnie układów opracowano

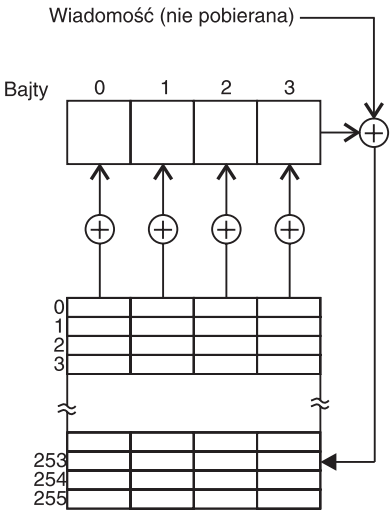
## Bezpieczna wymiana danych w systemach mikroprocesorowych



Rys. 5



Rys. 6



Rys. 7

odwróconą metodę tablicową. Rozpatruje się w niej ciąg bajtów taki sam jak w metodach poprzednich, ale bity w każdym bajcie są odwrócone: bit 7 staje się bitem 0, bit 6 bitem 1, itd. Tym razem dane są przesuwane w rejestrze nie w lewo, lecz w prawo. Wartość początkowa rejestru jest taka sama jak w poprzedniej metodzie, przy czym poszczególne bity są odwrócone zgodnie z powyższym opisem. Odwrócone są też dane zapisane w tablicy. Jedynie dane pozostają bez zmian. Do obliczeń pobierane są bity w kolejności nadchodzenia (czyli od bitu najmłodszego do najstarszego). Zasadę działania algorytmu ilustruje **rys. 7**.

Zauważmy, że i w tym przypadku strumień danych nie musi być przepuszczany przez rejestr. No i jeszcze jeden szczegół. Po zakończeniu obliczeń w rejestrze znajduje się obliczona wartość CRC... oczywiście odwrócona. Powyższy algorytm będziemy nazywali *tablicowym algorytmem odwróconym*.

Oczywiście nasuwa się spostrzeżenie, że przecież można odpowiednio odwrócić bity przed przesłaniem ich do UART-u, bez konieczności opracowywania dodatkowych algorytmów. Nie zawsze jest to jednak możliwe, choćby z uwagi na zachowanie kompatybilności z innymi systemami. Poza tym, wbrew pozorom nie jest to też zadanie łatwe do wykonania dla procesorów. Na ogół nie posiadają one odpowiedniego rozkazu i w takiej sytuacji musiałyby wykonywać „kawałek" bardziej złożonego programu. Dodatkowe angażowanie procesora do takich działań w przypadku stosowania dużych prędkości transmisji mogłoby być nie do przyjęcia.

## Metoda odwróconego generatora

Nie, nie, to jeszcze nie koniec. W nadziei, że wprowadzanie nowych udziwnień może skutkować nieoczekiwanymi efektami końcowymi, tym razem zastosujemy trochę niejasną na „pierwszy rzut oka" zmianę. Będzie ona polegała na odwróceniu wielomianu generującego. Jeśli wartość G=11101 była dobra, to 10111 również powinna się sprawdzić i okazuje się w praktyce, że jest to słuszne założenie. Powyższy pomysł znalazł uznanie w organizacji zajmującej się standaryzacją zagadnień związanych z transmisją danych - CCITT, która do „legalnych" generatorów zaliczyła również generatory odwrócone. Dla uniknięcia zamieszania są one oficjalnie nazywane *reversed*. Mamy więc np.:

X25 standard: 1-0001-0000-0010-0001
X25 reversed: 1-0000-1000-0001-0001

oraz

CRC16 standard: 1-1000-0000-0000-0101
CRC reversed: 1-0100-0000-0000-0011

Zwróćmy uwagę na to, że zostały odwrócone całe generatory włącznie z domyślnym bitem o wartości „1" na najstarszej pozycji, a nie tylko W dolnych bitów. Jest to znacząca różnica. W opisywanym wcześniej tablicowym algorytmie odwróconym stosowany generator był identyczny z tym, jaki wykorzystywaliśmy w metodzie nieodwróconej. W związku z tym musimy pamiętać, że odwrócony algorytm tablicowy nie jest odpowiednikiem algorytmu z odwróconym generatorem.

Czy to już wszystko? Jeśli chodzi o algorytmy, to można powiedzieć, że tak. Pozostało jeszcze kilka zagadnień ogólnych i długo oczekiwane zapewne przykłady, przykłady, przykłady. Na razie głowa chyba nam jednak urosła i trzeba jej dać odpocząć. Do następnego miesiąca!

**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.