Introduktion till feldetekterande och felkorrigerande koder

av: Andreas Larsson
Mattias Andersson

Högskolan Karlskrona Ronneby Karlskrona, 1997

Förord

Fel vid dataöverföringar har, och kommer alltid att förekomma. Det finns därför ett stort behov av felhantering.

Den här rapporten är tänkt att ge en inblick i de mest förekommande metoderna för feldetektering och felkorrigering.

Huvudsakligen kommer CRC och Hamming-koder att behandlas. Rapporten är tänkt att ge en översiktlig bild av dessa metoder, och ej i detalj beskriva kodernas uppbyggnad och matematiska bakgrund.

Vidare beskrivs även de vanligaste tillämpningarna för dessa felhanterings-koder.

Innehållsförteckning

1. Varför behövs felhantering?

Alla har nog någon gång råkat ut för en dålig telefonlinje med brus och knaster. När man talar med någon på en sådan linje kan man för det mesta förstå den andra parten genom att gissa det man inte hörde, men ibland måste man be den andre upprepa meningen för att man inte hörde tillräckligt mycket.

På samma sätt påverkas till exempelvis en modemförbindelse mellan två datorer av linjens kvalitet, men här är det ettor och nollor som kan komma bort eller ändras. Feldetektering och felkorrigering blir här viktig eftersom exempelvis en el-räkning annars kan bli 64.000 kronor, istället för 640, ett program uppbyggt av mer än 1 miljon bitar kan bli obrukbart på grund av att en enda bit ändrar värde.

2. Begrepps definition

I kommunikations sammanhang talas det allt mer om fel vid överföringar och vad man skall göra åt det. I samband med detta dyker en del termer upp som kan behöva förklaras.

- Felkontroll (error control): att man överhuvudtaget bryr sig om att leta efter fel i den mottagna datan.
- Feldetektering (error detection): behandlar bara upptäckandet av fel, inte vilka åtgärder som skall vidtagas.
- Felkorrigering (error correction): försök att åtgärda felet hos mottagarsidan utan att behöva skicka om datan.
- Felåterhämtning (error recovery): innefattar både själva feldetekteringen samt en åtgärd som leder till att felet repareras.

3. Enkel feldetektering och felkorrigering

Hur skall man då upptäcka att fel uppstår vid dataöverföring? Det enklaste sättet är att helt enkelt skicka all information två gånger. På detta sätt skulle sekvensen '101' bli '110011'. Nu skulle mottagaren enkelt kunna kontrollera den mottagna sekvensen, eftersom den vet att alla par bitar skall vara lika. Om den tidigare nämnda sekvensen kommer fram som '110111' vet mottagaren att det blivit ett fel i överföringen. Dock vet den inte om det ursprungliga meddelandet var '101' eller '111'. Mottagaren kan i detta fall i princip bara be om en omsändning av sekvensen.

Ett sätt att förbättra felåterhämtningen är att sända allt tre gånger. Meddelandet '101' skulle då bli '111000111'. Om det kommer fram som '110010111' kan mottagaren anta att det korrekta meddelandet var '101'.

Man inser dock rätt snart att man behöver mer intelligenta kodningsmetoder för att åstadkomma felåterhämtning, efter som dom ovan nämnda metoderna leder till att två respektive tre gånger så stora datamängder måste överföras jämfört med hur mycket man vill överföra.

3.1 Seriell paritet

En bättre metod för feldetektering är att lägga till paritet på datan. Man lägger till en extra bit så att det blir ett jämnt eller udda antal ettor i meddelandet. Detta kallas för jämn respektive udda paritet.

Paritet	Data	Data med paritet
jämn	1001	10010
udda	1001	10011

Mottagaren räknar sedan med XOR för att kontrollera datan. Hela den mottagna sekvensen skall vara lika med noll vid jämn paritet, och ett vid udda:

$$1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$
$$1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1$$

I exemplet ovan använder vi oss av 4-bitar långa meddelanden. Med en paritetsbit adderad på slutet kan vi detektera alla en-bits-fel i det överförda 5-bits meddelandet. Men om det uppträder fel i 2 bitar är det omöjligt att upptäcka:

data	data med paritet	mottagen data
1001	10010	1**1**0**0**0

När mottagaren kontrollerar detta meddelande finner den att
(1 ⊕ 1 ⊕ 0 ⊕ 0 ⊕ 0) = 0, och upptäcker inte felen. Denna metod bör således endast användas där sannolikheten för två på varandra tätt följande fel är mycket liten.

Vidare tillhandahåller den seriella paritetsmetoden inte någon information om vilken bit i det överförda meddelandet som är felaktig.

3.2 Parallell paritet

Parallell paritet bygger på vanlig seriell paritet. Parallell paritet kan upptäcka om det har uppstått något fel och dessutom upptäcka vilket bit som förändrats. Metoden tillämpas på ett block av data, där en paritetsbit läggs till varje rad och kolumn:

									paritet
	1	0	1	0	1	1	1	1	**0**
	1	0	0	0	0	0	1	1	**1**
	0	1	0	0	0	0	0	0	**1**
	1	1	1	1	0	0	0	0	**0**
	1	0	1	1	1	0	0	1	**1**
	0	0	0	0	0	1	1	1	**1**
	1	1	1	1	1	1	1	1	**0**
	0	1	1	1	1	0	0	0	**0**
paritet	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**

Om en bit i blocket har förändrats får vi två paritetsfel, ett i raden och ett i kolumnen. Genom att utnyttja dessa fel, kan vi konstatera vilken bit som blivit fel, och rätta till den. Detta är således en (enkel) typ av felkorrigering.

Men även denna metod har sina starka begränsningar. I exemplet var det 64 + 16 bitar som överfördes och med så pass många bitar är risken stor att mer än en bit blir fel. Om det uppstår jämna multiplar av fel i en rad eller kolumn kommer dessa fortfarande inte upptäckas.

3.3 Kontrollsumma

Ett annat sätt att implementera enkel feldetektering är att använda sig av kontrollsummor (check sums). Dessa tillämpas mest på byte nivå. En vanlig implementation av dessa är att använda sig av modulo 256 beräkning. Man räknar ut värdet av ett antal bytes, lägger på en kontrollsumma sådan att den tillsammans med den ursprungliga datan ger resten 0 vid en modulo 256 beräkning.

															ks
12	40	05	80	FB	12	00	26	B4	BB	09	B4	12	28	74	11	**BB**
12	00	2E	22	12	00	26	75	00	00	FA	12	00	26	25	00	**3A**
74	11	12	00	2E	22	74	13	12	00	2E	22					**B4**

I exemplet är varje byte representerad av sitt hexadecimala värde. Det sista värdet i varje rad är kontrollsumman. Om datan blivit korrupt under transporten kommer detta att visa sig när mottagaren gör modulo 256 beräkning på raderna genom att resten <> 0. Dimensionering av raderna följer två riktlinjer; de skall inte vara så korta att kontrollsumman utgör allt för stor del av den totala datan, de skall inte heller vara för stora, eftersom all data måste sändas om vid ett detekterat fel. Dessutom ökar risken för att felen slipper igenom kontrollen.

Det finns givetvis 1/256 risk att felen leder till att resten blir noll. Säkerheten kan förbättras genom att använda 2 bytes till kontrollsumman.

4. Olika typer av fel

Det finns givetvis flera typer av fel. Hittills har mest en-bits-fel diskuterats, men det finns fler och mer komplicerade. En definition av de vanligaste feltyperna är på sin plats.

- **en-bits-fel**: Endast ett fel i en given datamängd (ram).
- **två-bits-fel**: Endast två fel i en given datamängd.
- **ojämnt antal bit-fel**: Multiplar av ojämnt antal fel (1,3,5,...)
- **jämnt antal bit-fel**: Multiplar av jämnt antal fel (2,4,6,...) Oss veterligen finns det inget protokoll som garanterar feldetektering vid denna typ av fel.
- **skur bit-fel** (burst errors): Begränsat område med ett felaktigt konstant värde i en given datamängd (0000...0000).

5. ARQ vs FEC

Det finns två huvudsakliga strategier när det gäller att ta hand om fel. De två strategierna är : ARQ (Automatic Repeat Request) och FEC (Forward Error Correction), ARQ har generellt sett de flesta fördelarna av de två typerna. Den största fördelen är att antalet extra bitar som behövs till feldetektering är väldigt litet jämfört med antalet bitar som behövs vid felkorrigering. Orsaken till detta är att ARQ bara behöver detektera felen för att sedan be om omsändning av datan. Så länge fel uppkommer ganska sällan, är detta en bra teknik, men ifall felen skulle uppkomma ofta så kan detta leda till att överföringshastigheten sjunker dramatiskt. Alternativet blir då FEC. Felen rättas då till hos mottagaren och datan behöver inte sändas om. Nackdelen är att det krävs fler antal extra bitar till felkorrigeringen än till feldetekteringen. Detta är dock en avvägning som man får göra.

Några av de mest använda FEC algoritmerna är :
- BCH - koder
- Hamming - koder
- Reed-Solomon - koder

Några av de mest använda ARQ algoritmerna är :
- CRC
- Seriell paritet
- Parallell paritet
- Modulo checksum

Paritet och checksum är tidigare behandlat i det här dokumentet och de följande sidorna kommer främst att behandla CRC och Hamming-koder.

6. CRC

CRC har, som tidigare sagts, flera fördelar. De främsta är :

- Extremt stor chans att detektera fel.
- Litet antal extra bitar.
- Lätt att implementera i mjukvara och hårdvara.

CRC är en av de mest använda teknikerna för feldetektering när det gäller datakommunikation. Tekniken används för att skydda block av data, sk frames. Detta sker genom att ta datan som ska överföras och lägga till X antal nollor, där X = polynomets storlek, och sedan dividera den modifierade datan med polynomet. På så sätt får man en rest (= FCS = Frame Check Sequence) över, som läggs till det ursprungliga meddelandet. Meddelandet + FCS:n överförs sedan, och väl framme hos mottagaren skiljs FCS:n och meddelandet åt och X antal nollor läggs till på samma sätt som hos sändaren och en likadan division sker. Om den mottagna FCS:n och den beräknade är identiska, har meddelandet blivit korrekt överfört. Ett alternativ till detta, är att ta hela den mottagna sekvensen och dividera med polynomet. Meddelandet är korrekt mottaget om resultatet av divisionen blir noll.

6.1 Exempel

Nedan följer ett exempel på en CRC-beräkning med ett 4-bitars polynom ($x^4 + x + 1$) som har storlek W = 4.

Ursprungligt meddelande : 1101011011
Polynom: 10011
Meddelande efter att ha lagt till W nollor : 11010110110000
Sedan är det dags att modulo-2 dividera meddelandet med polynomet.

```
                    1100001010  skräptal
Polynom  10011 | 11010110110000  modifierat meddelande
               10011
                10011
                10011
                 00001
                 00000
                  00010
                  00000
                   00101
                   00000
                    01011
                    00000
                     10110
                     10011
                      01010
                      00000
                       10100
                       10011
                        01110
                        00000
                         1110  Rest = kontrollsumman = FCS
```

Vanligtvis läggs resten till efter det ursprungliga meddelandet. Mottagaren kan, efter att ha mottagit det kodade meddelandet, välja mellan två alternativ :

1. Meddelandet separeras från FCS:n och FCS:n ersätts med nollor, varpå mottagaren gör en likadan division som sändaren och jämför den mottagna FCS:n med den beräknade.
2. Mottagaren dividerar det hela kodade meddelandet med polynomet och kontrollerar att resten blir noll.

Beräkningen av kontrollsumman kan slutligen sammanfattas i 3 enkla punkter:

1. Välj ett polynom G med storleken W.
2. Lägg till W antal nollor till meddelandet => M'
3. Modulo-2-dividera M' med G. Resten blir kontrollsumman FCS.

6.2 Typer av CRC

Det används huvudsakligen tre stycken olika CRC - polynom :

- CRC-16
- CRC-CCITT
- CRC-32

CRC-16 och CRC-CCITT är båda 16-bitars polynom. Skillnaden mellan dem är bara själva polynomet. För CRC-16 är det $x^{16}+x^{15}+x^2+1$, och för CRC-CCITT $x^{16}+x^{12}+x^5+1$. Det mest använda polynomet av de två är CRC-CCITT. Den finns nu för tiden implementerad i stort sett alla diskkontrollers. IBM implementerade i sin diskkontroller till sina 8” floppydiskar, CRC-CCITT och gjorde den därmed till en standard.

CRC-32 består av ett 32 -bitars polynom och ger, därmed med sina 16 bitar mer, möjlighet att överföra större datapaket med högre säkerhet. Nackdelen är att det blir en 16 bitar större FCS.

Det bör nämnas att 16-bitars polynom egentligen är 17 bitar långa men de genererar en 16 bitar lång FCS. På samma sätt är ett 32 bitars polynom egentligen 33 bitar långt.

I ett korrekt utformat polynom är alltid MSB och LSB en etta. Polynomet ska också kunna detektera:

- Alla kontinuerliga fel som är kortare än polynomet.
- Alla udda fel.
- Alla 2-bits fel.

I de flesta fallen klarar polynomet av att hitta stort sett alla fel. Det finns dock en liten risk att en del fel undgår att bli detekterade. Detta sker när datan, efter att felen har smugit sig in, genererar en rest som är identisk med den ursprungliga. Med CRC-16 är dock bara sannolikheten 1 på 65536, eller 0.002%, att det ska smyga sig in ett fel i meddelandet. Det finns ingen teknik som absolut garanterar felfrihet men man kan minimera felen till en godtagbar kostnad. De andra teknikerna för feldetektering, som paritet och checksum, ger mycket sämre feldetektering. T.ex. 1-bytes checksum är till 99,29 % säker medan CRC, med bara en byte mer, ger 460 ggr högre säkerhet. I praktiken är CRC:n till och med ännu säkrare p.g.a. av att den upptäcker alla fel där det bara är några enstaka bitar som har blivit fel, och det är dessa situationer som uppkommer oftast.

6.3 Snabbare CRC

CRC kontrollen sker ofta i mjukvaran. För att snabba upp detta, kan man beräkna alla möjliga kombinationer av data i förväg, för att sedan bara behöva leta upp resultatet i en tabell när datan kommer. Nackdelen med detta är att för en 16 bitars CRC i kombination med en byte, så skulle det behövas en tabell som är 34 Megabyte stor. Det är oftast inte praktiskt möjligt med en så stor tabell, utan man blir tvungen att reducera den. Med hjälp av vissa logiska samband, kan man till slut komma fram till att det räcker med en 512 bytes stor tabell istället. Vissa beräkningar måste dock ändå ske när datan anländer hos mottagaren, men de blir betydligt enklare än ursprungligen.

Avslutningsvis kan man om CRC-koder säga att det är ett av de bästa sätten för feldetektering som finns, p.g.a. sin enkla implementation och litet antal feldetekteringsbitar tillsammans med en mycket hög säkerhet.

7. Olika typer av felåterhämtning

Det finns ett antal olika förfarande vid felåterhämtning. I detta avsnitt förklaras de två vanligaste metoderna kortfattat.

- Stop-and-wait ARQ: Detta är den enklaste typen av felåterhämtning som baserar sig på stop-and-wait flödes kontroll. Sändaren kan inte skicka någon ny ram med data förrän mottagaren talat om att den tagit emot föregående ram korrekt. Om mottagaren upptäcker ett fel i ramen, slänger den bort denna och väntar. Om sändaren inte får något svar från mottagaren inom en viss tid, skickar denna om ramen.
  Ett annat fel som kan uppstå är att bekräftelsen inte kommer fram korrekt. Detta tolkar sändaren som att den inte fått någon bekräftelse alls, och skickar alltså om ramen. Varannan ram numreras 1 och varannan 0, på detta vis kan mottagaren upptäcka om den får två kopior av samma ram. Den slänger då den senast mottagna och skickar en ny bekräftelse.

- Go-Back-N ARQ: Baserar sig på sliding windows flödes kontroll. Sändaren skickar ett antal ramar som är sekventiellt numrerade. När mottagaren upptäcker ett fel i en ram, skickar den ett negativt bekräftande till sändaren och kastar sedan alla ramar tills den får en ny kopia på den felaktiga. Sändaren måste gå tillbaka till det nummer som mottagaren returnerat och skicka om alla ramar från och med detta.

De verktyg som används av metoderna är följande:

- ARQ (Automatic Repeat Requests): be om omsändning av data.
- Positivt bekräftande: talar om att datan kom fram korrekt och att mottagaren är beredd att ta emot nästa ram med data.
- Negativt bekräftande: talar om att ramen kom fram felaktigt och ber om omsändning.
- Återsändning efter time-out: om inget svar fås inom en given tid skickas en ny ram.

8 Felkorrigeringens historia

Felkorrigeringskoder har utvecklats i takt med att medierna som används till digital överföring har tillåtit större datamängder. Den första felkorrigeringskoden upptäcktes 1945 av Claude Shannon när han bevisade sin teori som sade att det finns ett sätt att minimera risken för att ett meddelande blir förstört genom att koda meddelandet innan man sänder det.

Den första boken om felkorrigering skrevs 1961 av W Wesley Peterson.

Elwyn Berlekamp och James Massey, upptäckte 1968 en algoritm som klarade av att korrigera flera fel. Det visade sig senare att algoritmen bara var en variation på en gammal algoritm som egyptierna använde, 300 f.Kr., för att hitta största gemensamma nämnare till två olika polynom.

Den första implementerade felkorrigeringskoden kom 1947 från en matematiker, Richard Hamming, som tröttnade på att hans dator ständigt hängde sig när den fick fel indata. Lösning var att datorn automatiskt skulle korrigera felen när de anlände. Hans lösning på detta kom att kallas Hammingkoder, vilka fortfarande används som grund till andra felkorrigeringskoder.

Två av de koderna är Golay-koder som upptäcktes av Marcel Golay. Det som är anmärkningsvärt är att några år tidigare publicerades en av koderna i en finsk tidning (Veikkaajo). Det var en finsk vadslagare som presenterade ett spelsystem, för den finska motsvarigheten till stryktipset (11 rader), som garanterade vinst varje gång. Tyvärr fungerar inte detta systemet på 13 rader. :-(

9 Hamming-koder

Hammingkodernas teknik baseras på införandet av paritetsbitar på sådant sätt att de överlappar varandra och tillsammans täcker maximalt antal databitar. På så sätt kan man korrigera alla en-bits-fel eller detektera alla två-bits-fel. Det finns dock ingen möjlighet att göra både och.

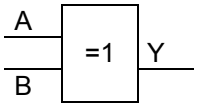
Antalet paritetsbitar ges av Hammings regel. Den är som följer : $d + p + 1 <= 2^P$, där d = antal bitar information som ska överföras och p = antal paritetsbitar.

9.1 Exempel 1

Som exempel kan man ta fram en (7,4) Hamming-kod. 7:an står för antalet bitar som kommer att överföras (d+p) och 4:an är antalet bitar data (d). Vanligtvis använder man större koder i applikationer.

Det finns två olika sätt att räkna fram bitsekvensen som ska skickas iväg. Nedan kommer båda att behandlas. Det första tillvägagångssättet är det mest matematiska.

Man gör en generatormatris som det är tänkt att datan ska multipliceras med. Detta sker genom modulo-2 multiplikation. Modulo-2 multiplikation är en matematisk model av det logiska kommandot XOR. XOR fungerar som följer:



A	B	Y
0	0	0
1	0	1
0	1	1
1	1	0

Generatormatrisen består av två delar: I = enhetsmatrisen och A = paritets-generatormatris.

$$G = [\, I \mid A \,] \quad G = \begin{array}{c} 1\,0\,0\,0 \mid 1\,1\,1 \\ 0\,1\,0\,0 \mid 0\,1\,1 \\ 0\,0\,1\,0 \mid 1\,0\,1 \\ 0\,0\,0\,0 \mid 1\,1\,0 \end{array}$$

När man multiplicerar 4-bit vektorn (d1, d2, d3, d4) = 1001 med G så resulterar detta i ett 7-bit kodord (d1, d2, d3, d4, p1, p2, p3) = 1001001. Det är själva A-delen av matrisen G som genererar de 3 sista paritetsbitarna. A måste innehålla 3 kolumner för att generera 3 paritets bitar. Kolumnerna i A väljs så att de är unika och skapar på så sätt 3 paritetsbitar som får unika egenskaper. Detta möjliggör felkorrigering. Mottagaren har en matchande generatormatris: H.

$$H = [\, A^T \mid I \,]$$

H - matrisen multipliceras med det inkommade kodordet r och skapar därmed produkten s.

$$H * r = s \qquad \begin{array}{c} 1011 \mid 100 \\ 1101 \mid 010 \\ 1110 \mid 001 \end{array} \quad * \quad \begin{array}{c} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{array} \quad = \quad \begin{array}{c} 0 \\ 0 \\ 0 \end{array}$$

Om alla elementen i s är noll, då är meddelandet korrekt mottaget. Fel på den inkomna bitsekvensen skapar paritetsfel och genom att analysera paritetsbitarna kan man hitta felet och korrigera det, så länge det rör sig om en-bits-fel.

9.2 Exempel 2

Ett annat sätt att få fram kodordet, och mindre matematiskt, är att beräkna varje paritetsbit för sig. Båda sätten att räkna fram kodordet ger dock egentligen samma resultat. Den största skillnaden ligger i att paritetsbitarna placeras på olika bitpositioner i kodordet av orsaker som beskrivs senare.

Låt oss säga att man vill koda bitsekvensen 1100 till ett 7-bitars kodord (x_1 x_2 x_3 x_4 x_5 x_6 x_7). Först fyller man i de 4 bitar som skall kodas enligt följande:

x_1 x_2 x_3 x_4 x_5 x_6 x_7
- - 1 - 1 0 0

Sedan är det dags att fylla i paritetsbitarna. Vi använder jämn paritet. I 1:a position fyller man i pariteten till x_1, x_3, x_5, x_7. I 2:a positionen fyller man i pariteten till x_2, x_3, x_6, x_7 och slutligen fyller man i position 4:a med pariteten till x_4, x_5, x_6, x_7.

$x_1 \oplus 1 \oplus 1 \oplus 0 = 0 \quad x_1 = 0$
$x_2 \oplus 1 \oplus 0 \oplus 0 = 1 \quad x_2 = 1$
$x_4 \oplus 1 \oplus 0 \oplus 0 = 1 \quad x_4 = 1$

Detta ger att kodsekvensen för 1100 blir 0111100. Om man sedan antar att det blir fel på biten i position 6, ger detta resultatet att alla paritetskontroller som innehåller denna bit kommer att misslyckas.

x_4 x_5 x_6 x_7 = 1
x_2 x_3 x_6 x_7 = 1
x_1 x_3 x_5 x_7 = 0

Värdena av paritetskontrollerna blir det binära talet 110 = 6 vilket också är de position som blivit felaktig.

Detta exemplet visar att det kan finnas fördela med att placera paritetsbitarna på särskilda positioner i kodordet. Felet hade dock kunnat detekteras och korrigeras ändå, men fördelarna är rätt tydliga. När det gäller val av antal paritetsbitar och datamängd, så är (7,4) inget bra val. Man strävar efter att sända hela dataord s.k. bytes i ett och samma paket och då är (12,8) att föredra. Kommunikationen blir också enklare p.g.a. att man kan låta en paritetsbyte övervaka två databyte och slipper därmed ojämnt antal bitar.

Sammanfattningsvis kan man säga att felkorrigerande koder kan underlätta i vissa fall p.g.a. att man slipper omsändningar, dock är nackdelen att de bara klarar av en-bits-fel och det är mycket extra data som skall överföras. 1 paritetsbyte per 2 byte data, jämfört med CRC: 2 byte FCS per 128 byte data. Vilken typ av felhantering man skall välja är en bedömningsfråga från fall till fall.

10. Reed-Solomon

En mycket använd typ av felkorrigeringskoder är de så kallade Reed-Solomon koderna. Det speciella med dessa är att de är näst intill perfekta i den mening att den

extra kod som krävs är minimal oavsett vilken nivå på felkorrigering man önskar uppnå. Dom är vidare förhållande vis väldigt enkla att avkoda. Dom är mycket lämpliga att använda för korrigering av skur bit-fel samt multipla slumpmässiga fel.

Av denna anledning används just Reed-Solomon koder för felkorrigering på CD-skivor, eftersom den största orsaken till fel på dessa är repor och smuts, vilka ger upphov till skur bit-fel. På CDDA skivor används två Reed-Solomon koder, båda på 8 bitar. Dessa arbetar med 28 respektive 32 bitar data inklusive själva kodordet. Båda kan korrigera 2 fel per kodord. Genom att kombinera dessa två koder på ett speciellt vis kan CD-spelaren klara av att korrigera skur bit-fel på över 4000(!) bitar.

Reed-Solomon specificeras även av European Telecommunication Standard för överföring av digital-TV via satellit, samt i Digital Video Transmission Standard för kabel-TV sändningar.

11. Framtida användning

Allt eftersom kommunikationsmediarna utvecklas och förbättras minskar risken för överföringsfel. I alla fall räknat i fel per överförda bitar. Men samtidigt sker det även en ökning i överföringshastighet. Detta leder till ett den så kallade fel-hastigheten, dvs fel per tidsenhet, bara minskar marginellt. Detta innebär att man även i fortsättningen kommer att behöva felkorrigering i de olika medierna.

Telefonnäten, som en gång användes för att överföra enbart röstmeddelanden, börjar i allt större utsträckning användas för datakommunikation. Hastigheten på överföringen är förhållandevis låg och fel-frekvensen hög. Vid denna typ av kommunikation vill man korrigera så mycket av felen som möjligt hos mottagaren, eftersom återsändning av datan tar för lång tid. Felkorrigeringen har tidigare kunnat göras i antigen mjukvara eller hårdvara tack vare de låga överföringshastigheterna, men allt eftersom hastigheterna kryper högre och högre växer behovet att speciell hårdvara för felhantering.

I satellitsystem förekommer det mycket störningar på grund av varierande atmosfäriska förhållande och kravet på FEC är hårt på grund av de stora avstånden och de fördröjningar som detta medför. Vidare skulle en betydande del av bandbredden gå förlorad vid omsändning av data. I dessa system används Reed-Solomon koder som ett 'yttre' skydd, och innanför dessa ligger det ytterligare FEC koder. Tillsammans bildar dessa ett effektivt skydd mot fel.

European Telecommunication Standard specificerar t.ex. användandet av Reed-Solomon koder för överföring av digital television.

Mikrovågslänkar används i allt större utsträckning för punkt-till-punkt kommunikation, internet kopplingar, mobiltelefoni med mera. Bruset på dessa varierar med avseende på atmosfäriska förhållanden, men även under de bästa förhållandena måste man ha felhantering. Hastigheten på mikrovågslänkarna ligger på mellan 1-8 Mbyte/s och felhanteringen måste skötas av speciell hårdvara.

Fiberoptiknäten expanderar med stor hastighet i hela världen. Dessa är väldigt störtåliga, men samtidigt används väldigt höga överföringshastigheter på 300 Gbit/s och mer, vilket leder till en ansenlig fel-hastighet. Om man räknar med att 1 bit per 100 miljarder bitar blir fel, innebär detta att med 300 Gbit/s kommer det att uppträda 3 fel i sekunden. Även här krävs alltså intelligent FEC.

Källförteckning

Francis Yein Chei Fung: A survey of the theory of error-correcting codes
(http://www.math.harvard.edu/~hmb/issue1.1/codes.html)

Ross N. Williams: CRC explained
(http://www.holli.com/~jbuchana/CRC_explained.html)

Introduction to error control
(http://www.pluscom.ru/mac/school/RAD/web/networks/1994/error_control/intro.html)

Terry Ritter: The great CRC mystery
(http://www.io.com/~ritter/ARTS/CRCMYST.HTM)

ECC for communications
(http://members.aol.com/mnecctek/comm.html)