De werking van de Hamming code

De Hamming code is een FEC of Forward Error Correction. Een FEC is een systeem waarbij de data zo wordt gecodeerd, dat de detectie en correctie van fouten in de datastroom in de ontvanger kan worden opgelost. Zo is het mogelijk om in een simplex systeem de informatie redelijk betrouwbaar over te sturen.

In deze publicatie zal ik de Hamming code als FEC verder uitwerken. Je zult zien dat we de Hamming code niet direct kunnen toepassen, er zit namelijk nog een addertje onder het gras.

In een eerdere publicatie heb je kunnen lezen dat een draadloos kanaal een alles behalve optimaal transmissiemedium is. We moeten rekening houden met allerlei storingen die ervoor kunnen zorgen dat de informatie van de zender niet of vervormd bij de ontvanger aankomt.
In de meeste gevallen is de storing van korte duur of niet van invloed zodat de gegevensstroom niet onderbroken of verminkt zal worden. Als de storing langer duurt of de invloed van de storing is groter, dan is het mogelijk dat er een fout wordt gemaakt bij het reconstrueren van de data. Dan spreken we over een bitfout; een ‘0’ wordt gezien als een ‘1’ of een ‘1’ wordt gezien als een ‘0’. De kans waarbij dit voorkomt in relatie tot het aantal bits dat wel goed wordt gereconstrueerd noemen we de BER of *Bit Error Rate* van de verbinding.

In een simplex verbinding heeft de ontvanger geen mogelijkheid om aan de zender door te geven dat een bit fout is ontvangen. Een parity of CRC heeft voor een simplex verbinding dus niet zoveel zin. Bij een parity of CRC moet er een mechanisme zijn om aan de zendende node te ‘vertellen’ dan er een fout is gedetecteerd. Zo kan de zender het frame nog een keer sturen. Bij een parity of CRC weet de ontvanger dat er een fout is in een bit is opgetreden, maar weet nog niet waar die fout is opgetreden. Het kan zelfs de partity zelf zijn. De Hamming code kan dit probleem gedeeltelijk oplossen.

Werking

De Hamming code heeft als eigenschap dat het een aantal databits voorziet van een flink aantal extra bits die op een specifieke manier van de databits zijn afgeleid. De payload wordt dus flink groter en hier moeten we rekening mee houden. In dit voorbeeld gaan we uit van een Hamming (7,4) code. Hierbij worden aan 4 databits nog 3 codebits toegevoegd. Deze code is vrij inefficiënt in vergelijking met bijvoorbeeld de Hamming (11,7) of Hamming (20,15). De Hamming (7,4) heeft maar liefst 43% redundantie. De Hamming (20,15) nog maar 25%.
Een andere eigenschap van de Hamming code is dat we 1 bitfout kunnen detecteren en corrigeren en dat we 2 bitfouten alleen kunnen detecteren. In ons geval dus 1 bitfout op de 7 bits: 4 databits + 3 bescherming bits. En dat betekent dat we een beschermingsverhouding hebben van 1:7. De kans dat er een bit fout is, is dus veel kleiner dan een Hamming (20,15) waarbij de beschermingsverhouding nog maar 1:20 is; bijna 3x zo klein dus. De winst van de verminderde redundantie van een grotere Hamming code zit dus in een gereduceerde bescherming van de databits.
Een ander aspect van een grotere Hammning code is de omvang van de coder en de decoder. Vooral de decoder is een flinke klus. De Hamming code heeft hiervoor wel een zeer vernuftig mechanisme, maar we kunnen stellen; hoe groter de Hamming code, hoe meer klopwerk we moeten doen.

De Hamming (7,4) code bestaat uit 4 databits en 3 redundante bits. En dat komt mooi uit want met twee van deze codes kunnen we dus 8 bits beschermen. Voor 8 bits hebben we dan 6 redundante bits nodig.

De 3 redundante bits bij de Hamming code worden parity genoemd; P1,P2 en P3.

	databits				parity		
	D3	D2	D1	D0	P3	P2	P1
0	0	0	0	0	0	0	0

P1 is de parity van de databits D0,D1 en D3.
P2 is de parity van de databits D0,D2 en D3.
P3 is de parity van de databits D1,D2 en D3.

De totale serie ziet er uit zoals in Tabel 1.
Er vallen meteen twee problemen op. De Hamming code voor de waarde 0 en F, leveren een opmerkelijk resultaat op.

1	0	0	0	1	0	1	1
2	0	0	1	0	1	0	1
3	0	0	1	1	1	1	0
4	0	1	0	0	1	1	0
5	0	1	0	1	1	0	1
6	0	1	1	0	0	1	1
7	0	1	1	1	0	0	0
8	1	0	0	0	1	1	1
9	1	0	0	1	1	0	0
A	1	0	1	0	0	1	0
B	1	0	1	1	0	0	1
C	1	1	0	0	0	0	1
D	1	1	0	1	0	1	0
E	1	1	1	0	1	0	0
F	1	1	1	1	1	1	1

Tabel 1

Databits en Hamming code.

De ontvanger heeft als taak de verzonden bits te reconstrueren, maar kan dat alleen als de data informatie bevat over de kloksnelheid. Er is immers geen systeem, anders dan de data zelf, om aan de ontvanger het kloksignaal door te geven. Soms kunnen we de ontvanger een beetje in de juiste richting bijsturen, maar helemaal strak krijgen we het nooit. De ontvanger heeft een klokfrequentie nodig voor bitsyncronisatie.
Omdat de ontvanger de klokfrequentie reconstrueert uit de data, moeten we er voor zorgen dat de data veel wisselende bits bevat. Het liefst willen we dat er maximaal 3 ‘1’-en of 3 ‘0’-en achter elkaar in de code voorkomen. De resultaten voor 0 en F, maar ook de resultaten voor 3 en C voldoen hier dus niet aan.

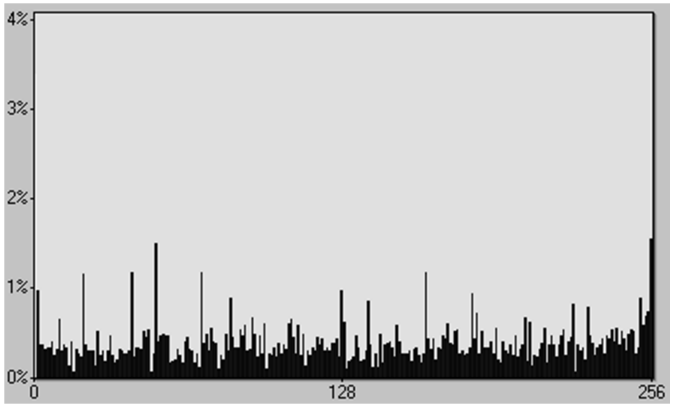
De werking van de Hamming code berust op het principe van afgeleide overtolligheid. De code is zo goed doordat er niet zomaar bits zijn toegevoegd, maar de bits zijn afgeleid van het origineel. Elk resultaat is dus uniek en er worden maar een paar codes uit het hele pallet van mogelijke codes gebruikt.
Op 6 databits worden 3 Hamming codebits toegevoegd. Het hele pallet aan codes is dus 128 (2^7). Het aantal Hamming codes is echter maar 16. De Hamming afstand (Hamming distance) is daarmee even groot als het aantal toegevoegde bits: 3. De Hamming afstand betekend dat er minstens 3 bits fout gereconstrueerd moeten zijn, voordat de code op een andere code lijkt.

Voorbeeld:

De Hamming code voor 8 [P1,P2,D0,P3,D1,D2,D3] = 1101001
De Hamming code voor 9 [P1,P2,D0,P3,D1,D2,D3] = <u>0011001</u>
Verschil: 1110000 = 3 bits

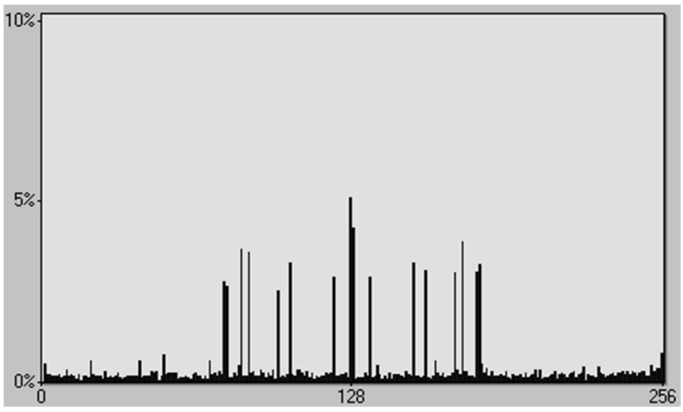
De volgorde van de kolommen maakt in deze berekening niets uit, zolang ze maar voor elk getal hetzelfde zijn.

Op een grafische manier is het nog beter te zien dat de Hamming code functioneert.



Figuur 1

Ongecodeerde data.



Figuur 2

Gecodeerde data met ruis.

In het histogram van Figuur 1 wordt de data van een afbeelding (jpg) weergegeven. Het is goed te zien dat de data een stochastisch (willekeurig) gedrag vertoont.
Het lijkt nog het meeste op ruis. Als een ontvanger een datastroom met deze inhoud te verwerken krijgt en een fout wordt gemaakt bij de reconstructie, dan is het vrijwel onmogelijk om de fout te detecteren laat staan te corrigeren.

In figuur 2 heb ik dezelfde hoeveelheid data gecodeerd met de Hamming code en daaraan evenveel ruwe data aan toegevoegd als ruis.
Duidelijk is te zien dat de Hamming codes ver boven de ruis uitsteken. Het histogram leert ons dat de decoder veel eenvoudiger en selectiever mag zijn. Hiermee is te beredeneren dat er ook een mate van versterking plaatsvindt.

Dit gaat helaas ten koste van de hoeveelheid data die per tijdseenheid getransporteerd kan worden. Bij gelijke data rate kan immers maar ongeveer de helft van de data worden verzonden.
Merk op dat de horizontale afstand (in Figuur 2) tussen de ‘paaltjes’ geen relatie heeft tot de selectiviteit of versterking.

De verbeterde Hamming code.

De manier om wat aan de tegenvallende bitsynchronisatie van de Hamming code te doen, is een procedure die *whitening* wordt genoemd. Whitening wordt hier gerealiseerd door de code zo te manipuleren zodat het teken (geïnverteerd of niet) en de volgorde van uitgezonden bits beter op de ideale vorm voor de data lijkt.
De ideale vorm voor de data is om en om een ‘0’ en een ‘1’. Als we deze code door een serieel kanaal sturen dan vormt dit een blokgolf. Hieruit kan de ontvanger eenvoudig het kloksignaal reconstrueren. Door om en om een ‘0’ en een ‘1’ te sturen kunnen we helaas geen informatie meer toevoegen aan de data. Het zal dus een compromis moeten worden tussen zoveel mogelijk bitwisselingen en de informatie in de vorm van de volgorde van de uitgezonden bits.

Een vorm van whitening is ook te vinden in de Manchester codering. De Manchester codering maakt gebruik van de dubbele data rate met als voordeel een inherent kloksignaal in de data. Manchester codering mist echter de eigenschappen van de Hamming code.

Het originele idee achter de Hamming code is pas compleet als de bits in de juiste volgorde achter elkaar worden gezet. Alleen op die manier heeft de code mathematische eigenschappen waarmee we op een vrij eenvoudige wijze een fout bit kunnen detecteren en corrigeren. Neem dit even van mij aan, ik ga hier niet verder op in. De code is te kort om hiervan veel voordeel van te hebben.
De verbeterde Hamming code maakt dus maar gedeeltelijk gebruik van de Hamming code; de parity.

Om whitening toe te kunnen passen moeten we een keuze maken: of we voegen extra stappen toe aan de coder voor de zender, of we passen een andere manier van coderen en decoderen toe.

Een van de mogelijkheden om de data te coderen is om in de firmware voor de microcontroller een (en)coder te schrijven. De coder moet eerst elke parity uitrekenen en dan de bits in de juiste volgorde zetten. Whitening kunnen we doen door de hele byte met een XOR functie te bewerken. De decoder in de ontvanger moet eerst de data met dezelfde XOR functie bewerken, dan de fouten herkennen en daarna het foute bit vervangen. Daarna kan de code worden gedecodeerd.

Een andere mogelijkheid van coderen is met behulp van een LUT, een *Look Up Table*. De LUT kunnen we maken door een lijst op te stellen van elke code voor elke mogelijkheid. Maar in feite bestaat deze LUT al. Tabel 1 bevat de LUT voor de zender. We hoeven de code alleen nog iets aan te passen voor de verbeterde bitsynchronisatie.
De 4 databits vormen de ingang van de LUT. De uitgang zijn 7 bits; de databits en de Hamming parity bits.

In de ontvanger ziet de LUT er iets groter uit. Het aantal mogelijkheden is door het toevoegen van bits immers gegroeid van 16 (2^4) naar 128 (2^7). Met het overzenden van 8 bits tegelijk worden het er zelfs 256 (2^8). Dat wordt een flinke tabel!

Met een beetje puzzelen kunnen we, door kolommen in de juiste volgorde te zetten en enkele kolommen te inverteren (de XOR functie), een heel eind in de juiste richting komen. We kunnen echter niet voorkomen dat er 4 ‘0’-en of 4 ‘1’-en achter elkaar komen te staan. In een datastroom waarbij alle bits achter elkaar staan zou dit dan inhouden dat er een flinke kans is dat er 8 ‘0’-en of 8 ‘1’-en achter elkaar worden uitgezonden. Bijvoorbeeld een byte met 4 ‘0’-en aan het einde en daarna een byte met 4 ‘0’-en aan het begin. De kans is groot dat de ontvanger hierbij de bitsynchronisatie verliest.
Als we de tabel zouden kunnen aanpassen zodat er nog maar maximaal 3 ‘0’-en of 3 ‘1’-en achter elkaar voorkomen, dan wordt de kans op het verlies van bitsyncronisatie al een stuk kleiner.

We gebruiken de Hamming code maar we zenden groepen van 8 bits uit. Dan blijft er dus 1 bit over waar we nog niets mee doen. Dit bit kunnen we gebruiken om het bitsynchronisatieprobleem nog kleiner te maken. In Tabel 2 wordt dit bit ‘S’ genoemd.

whitening	0	0	1	-	1	1	1	0	
Kolom: Ingangs-waarde:	P2	D3	D1	**S**	P3	D2	P1	D0	Uitgangs waarde
0	0	0	1	0	1	1	1	0	0x2E
1	1	0	1	0	1	1	0	1	0xAD
2	0	0	0	1	0	1	0	0	0x14
3	1	0	0	1	0	1	1	1	0x97
4	1	0	1	1	0	0	1	0	0xB2
5	0	0	1	1	0	0	0	1	0x31
6	1	0	0	1	1	0	0	0	0x98
7	0	0	0	1	1	0	1	1	0x1B
8	1	1	1	0	0	1	0	0	0xE4
9	0	1	1	0	0	1	1	1	0x67
A	1	1	0	0	1	1	1	0	0xCE
B	0	1	0	0	1	1	0	1	0x4D
C	0	1	1	0	1	0	0	0	0x68
D	1	1	1	0	1	0	1	1	0xEB
E	0	1	0	1	0	0	1	0	0x52
F	1	1	0	1	0	0	0	1	0xD1

Tabel 2

Aangepaste tabel: de LUT voor de zender.

Van de mathematische eigenschappen van de Hamming code is nu helaas niet zoveel meer over. Ik heb de kolommen door elkaar gegooid en enkele kolommen geïnverteerd. Door gebruik te maken van een LUT kunnen we toch van de beschermende eigenschappen van de Hamming code gebruikmaken en het levert een vrij eenvoudige decoder op.

De LUT voor de zender is klein. Tabel 2 geeft in de rechter kolom de corresponderende resultaten weer.

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0		2		5		E	C			6		4		F	8	
x1		5	5	5		F		5		F		5	F	F		
x2		E		4	E	E		E		4	4	4		E		4
x3		7		5		E	9			3		4		F	D	
x4	2	2		2		2	8			2	8		8		8	8
x5		2		5	B		9			3	1			F	8	
x6		2	0			E	9			3		4	A		8	
x7		3	9		9		9	9	3	3		3		3	9	
x8		6	C		C		C	C	6	6		6		6	C	
x9		7		5	B		C			6	1			F	D	
xA		7	0			E	C			6		4	A		D	
xB	7	7		7		7	D			7	D		D		D	D
xC		2	0		B		C			6	1		A		8	
xD	B		1		B	B	B		1		1	1	B		1	
xE	0		0	0	A		0		A		0		A	A	A	
xF		7	0		B		9			3	1		A		D	

Tabel 3

Matrix voor de ontvanger LUT.

In de ontvanger wordt de tabel iets groter, 16x groter om precies te zijn. Om deze tabel te maken moeten we eerst alle codes uitrekenen die 1 bitfout hebben ten opzichte van de juiste code. De ontvanger LUT geeft dan voor die

waarden de juiste waarde terug. Als we alles hebben uitgerekend zetten we de waarden in een 16x16 matrix.

De lichtgrijze vlakjes in Tabel 3 stellen de locaties voor waarbij de ontvanger geen fouten heeft gemaakt. De overige vakjes met getallen en letters stellen de locaties voor waarbij er 1 bitfout is gemaakt. De twee donkergrijze vakjes stellen de bytes voor die door de data verbindingslaag worden gebruikt om aan te geven dat er een pauze volgt (0xAA) of dat er een commando byte volgt (0x55). Beide getallen zijn dus uitgezonderd van de Hamming code of de gereconstrueerde waarden met 1 bitfout.

Voorbeeld:

De zender moet de decimale waarde 10 uitzenden. Uit de LUT van de zender volgt dat voor de waarde 10 (0x0A), een byte met waarde 0xCE uitgezonden wordt. Dit is het resultaat van de LUT.
In de ontvanger wordt de waarde (in dit voorbeeld) fout gereconstrueerd tot 0x8E. Hierbij is dus bit 6 niet juist ontvangen en van een ‘1’ in een ‘0’ is veranderd. Uit Tabel 3 blijkt dat een ingang van 0x8E (8 horizontaal en E verticaal) de waarde A staat. Het resultaat van de decoder LUT is dus A en dat is de waarde 10.
De ontvanger LUT heeft nu de ontvangen byte niet alleen gedecodeerd maar ook het foute bit gecorrigeerd. Zo eenvoudig werkt het.

We kunnen de ontvanger LUT nog iets uitbreiden. We maken immers maar gebruik van de helft van de tabel. De uitgang van de decoder is een byte met een waarde tussen 0 en 15; de laagste 4 bits. De hoogste 4 bits zijn altijd ‘0’. Door een van deze bits te definiëren kunnen we nog een kunstje aan de decoder meegeven. Door aan elke gecorrigeerde waarde met 1 bitfout een extra bit van de uitgangswaarde toe te voegen, wordt het mogelijk om het aantal bitfouten te tellen. Het bit blijft bij de juiste uitgangswaarde gewoon 0. De juiste uitgang van de decoder kunnen we krijgen door de hoogste bits te negeren.
Door van een bericht het aantal juist ontvangen bytes en het aantal gecorrigeerde bytes te tellen, kan de BER wordt uitgerekend. Als er bijvoorbeeld op een bericht van 1000 bytes er 3 waarden met 1 bitfout zijn ontvangen en gecorrigeerd, dan is de BER $3x10^{-3}$.

De LUT bevat ook nog een heleboel vakjes waar geen waarde in staat. Dat zijn de tabelwaarden waarbij 2 bitfouten zijn opgetreden bij de reconstructie in de ontvanger. Hierin moet een waarde worden geplaatst waarmee we in de decoder kunnen aangeven dat deze uitzondering heeft plaatsgevonden. Bijvoorbeeld de waarde 0xFF. Helaas kunnen we hier niet zoveel mee doen, behalve registreren.

De uiteindelijke ontvanger LUT ziet er uit zoals in de onderstaande tabel.

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	FF	12	FF	15	FF	1E	1C	FF	FF	16	FF	14	FF	1F	18	FF
x1	FF	15	15	05	FF	1F	FF	15	FF	1F	FF	15	1F	0F	FF	FF
x2	FF	1E	FF	14	1E	0E	FF	1E	FF	14	14	04	FF	1E	FF	14
x3	FF	17	FF	15	FF	1E	19	FF	FF	13	FF	14	FF	1F	1D	FF
x4	12	02	FF	12	FF	12	18	FF	FF	12	18	FF	18	FF	08	18
x5	FF	12	FF	15	1B	55	19	FF	FF	13	11	FF	FF	1F	18	FF
x6	FF	12	10	FF	FF	1E	19	FF	FF	13	FF	14	1A	FF	18	FF
x7	FF	13	19	FF	19	FF	09	19	13	03	FF	13	FF	13	19	FF
x8	FF	16	1C	FF	1C	FF	0C	1C	16	06	FF	16	FF	16	1C	FF
x9	FF	17	FF	15	1B	FF	1C	FF	FF	16	11	FF	FF	1F	1D	FF
xA	FF	17	10	FF	FF	1E	1C	FF	FF	16	AA	14	1A	FF	1D	FF

xB	17	07	FF	17	FF	17	1D	FF	FF	17	1D	FF	1D	FF	0D	1D
xC	FF	12	10	FF	1B	FF	1C	FF	FF	16	11	FF	1A	FF	18	FF
xD	1B	FF	11	FF	0B	1B	1B	FF	11	FF	01	11	1B	FF	11	FF
xE	10	FF	00	10	1A	FF	10	FF	1A	FF	10	FF	0A	1A	1A	FF
xF	FF	17	10	FF	1B	FF	19	FF	FF	13	11	FF	1A	FF	1D	FF

Tabel 4

Matrix voor de ontvanger LUT.