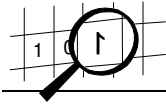


# 1 VERZEICHNISSE

## 1.1 INHALT

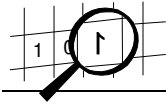
|  |    |
|--|----|
| 1 Verzeichnisse.....                           | 2  |
| 1.1 Inhalt.....                                | 2  |
| 1.2 Abbildungen.....                           | 3  |
| 1.3 Literaturverzeichnis.....                  | 4  |
| 1.4 Abkürzungen.....                           | 5  |
| 2 Aufgabenstellung.....                        | 6  |
| 3 Einleitung .....                             | 9  |
| 4 Konfigurationsdaten .....                    | 11 |
| 4.1 Ressourcen .....                           | 11 |
| 4.2 Implementation auf dem DSP .....           | 13 |
| 4.3 Aufbau des DSP – Programms .....           | 14 |
| 4.4 Kommunikation zwischen dem DSP und PC..... | 15 |
| 4.5 Oberfläche und deren Funktionen.....       | 17 |
| 5 Verwaltungsdaten.....                        | 19 |
| 5.1 Der 15-7 BCH Code.....                     | 19 |
| 5.1.1 Bemerkungen.....                         | 23 |
| 5.2 Der 1023-943 BCH Code .....                | 24 |
| 6 Auswertung .....                             | 25 |
| 6.1 Binominalverteilung.....                   | 25 |
| 6.2 Der Bitfehlergenerator.....                | 26 |
| 6.2.1 Erste Variante.....                      | 26 |
| 6.2.2 Zweite Variante .....                    | 27 |
| 6.3 Der 15-11 Hamming – Code .....             | 28 |
| 6.4 Der 16-11 – Code .....                     | 32 |
| 6.5 Gesamter Konfigurationsdatenblock .....    | 34 |



|  |    |
|--|----|
| 6.6 Der 15-7 BCH Code.....   | 36 |
| 6.6.1 WSK, dass ein Fehler nicht korrigiert werden kann .....      | 36 |
| 6.6.2 WSK, dass ein Fehler falsch korrigiert wird (Desaster) ..... | 37 |
| 6.7 Der 1023-943 BCH Code .....                                    | 39 |
| 6.7.1 WSK, dass ein Fehler nicht korrigiert werden kann .....      | 40 |
| 6.7.2 WSK, dass ein Fehler falsch korrigiert wird (Desaster) ..... | 41 |
| 7 Schlusswort.....   | 43 |

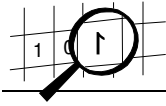
## 1.2 ABBILDUNGEN

|   |    |
|---|----|
| Abbildung 1 Blockschema .....   | 9  |
| Abbildung 2 Decodierung der Konfigurationsdaten.....                        | 11 |
| Abbildung 3 Ablauf einer DSP – Programmentwicklung .....                    | 13 |
| Abbildung 4 Oberfläche des Decodierprogrammes für Windows 95 .....          | 17 |
| Abbildung 5 15-7 BCH Windowstestprogramm.....                               | 22 |
| Abbildung 6 Wahrscheinlichkeitsfunktion $f(x)$ .....                        | 26 |
| Abbildung 7 Performancekurve 15-11 Code bei Korrektur.....                  | 29 |
| Abbildung 8 Performancekurve 15-11 Code bei Erkennung .....                 | 31 |
| Abbildung 9 Performancekurve 16-11 Code .....                               | 33 |
| Abbildung 10 WSK, dass gewisse Anzahl Worte in einem Block fehlerhaft ..... | 34 |
| Abbildung 11 WSK, dass gewisse Anzahl Worte in einem Block fehlerhaft ..... | 35 |
| Abbildung 12 Performancekurve 15-7 BCH $p_i(p_b)$ .....                     | 36 |
| Abbildung 13 Performancekurve 15-7 BCH $p_d(p_b)$ .....                     | 38 |
| Abbildung 14 Performancekurve 15-7 BCH $p_d(p_b)$ .....                     | 39 |
| Abbildung 15 Performancekurve 1023-943 BCH $p_i(p_b)$ .....                 | 40 |
| Abbildung 16 Performancekurve 1023-943 BCH $p_d(p_b)$ .....                 | 42 |
| Abbildung 17 Performancekurve 1023-943 BCH $p_i(p_b)$ und $p_d(p_b)$ .....  | 42 |



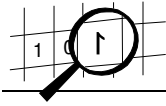
## 1.3 LITERATURVERZEICHNIS

- [1] Shu Lin, Daniel J. Costello: *Error Control Coding.- Fundamentals and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1983
- [2] W. Wesley Peterson: *Prüfbare und korrigierbare Codes*, R. Oldenbourg Verlag, München und Wien, 1967
- [3] Joachim Swoboda: *Codierung zur Fehlerkorrektur und Fehlererkennung*, R. Oldenbourg Verlag, München und Wien, 1973
- [4] *User's Guide 'DSP21k Toolkit'*, Bittware, May 1996 Edition
- [5] *ADSP-21000 Family 'C Tools Manual'*, Analog Devices, Third Edition Aug. 1995
- [6] Lothar Papula: *Mathematik für Ingenieure und Naturwissenschaftler Band 3*, Vieweg Braunschweig/Wiesbaden, 1994
- [7] Yves Gross und Tobias Läubli: *Semesterarbeit SS98-1 'Forward Error Correction'* September 1998



## 1.4 ABKÜRZUNGEN

- n : Anzahl Bits eines Codewortes
- m : Anzahl der Prüfbits
- k : Anzahl der Informationsbits
- t : maximale Anzahl korrigierbare Fehler
- h : Hammingdistanz
- u : Information (k Bits)
- $\hat{u}$  : Information nach decodieren (evtl. fehlerhaft)
- v : Codewort (n Bits)
- r : Codewort nach decodieren (evtl. fehlerhaft)
- S** : Syndromvektor
- p : Generatorpolynom
- G : Generatormatrix
- H : Kontrollmatrix (n Bits)
- GF : Galoisfeld
- $\alpha^i$  : Galoiselement
- DSP : Digitaler Signal Prozessor



## 2 AUFGABENSTELLUNG

**Diplomarbeit** für Herrn Y. Gross und Herrn T. Läubli

### **Codierung zur Erkennung und Korrektur von Bitfehlern**

Aufbauend auf die Resultate der vorangegangenen Studienarbeit sollen in dieser Diplomarbeit

- eine Fehlerschutzcodierung für Konfigurationsdaten auf einer DSP-Karte und
- eine Fehlerschutzcodierung für Verwaltungsdaten auf einem PC implementiert werden.

Bei den Konfigurationsdaten handelt es sich um 127 x 11 Bit Nutzdaten, die in einem Speicherbereich von 128 x 16 Bit gespeichert werden. Mit einem CRC-Code soll zunächst aus den 127 x 11 Bit ein weiteres 11 Bit breites Wort und danach mit einem (15,11)-Hamming-Code und einem zusätzlichen Paritätsbit aus den jeweils 11 Bit je ein 16 Bit breites Wort erzeugt werden. Mit diesem Code lässt sich ein einzelner Bitfehler pro 16 Bit breites Datenwort korrigieren, und deren zwei können noch erkannt, aber nicht mehr korrigiert werden. Tritt letzterer Fall bei mindestens einem Datenwort auf, so kann wenigstens ein Aufstarten eines Gerätes mit falschen Daten verhindert werden. Der CRC-Code, ebenfalls ein - jedoch gekürzter Hamming-Code geeigneter Länge, wird lediglich zur Fehlererkennung verwendet, um die Wahrscheinlichkeit für ein Aufstarten mit falsch korrigierten Daten tief zu halten.

Bei den Verwaltungsdaten handelt es sich um 27 x 16 Bit Nutzdaten, die in einem Speicherbereich von 32 x 16 Bit gespeichert werden. Als Fehlerschutzcodierung soll ein BCH-Code geeigneter Länge verwendet werden, und dessen Nutzung zur Fehlerkorrektur bzw. lediglich zur Fehlererkennung soll in Abstimmung mit der Leistungsfähigkeit des Codes der ersten Teilaufgabe erfolgen.

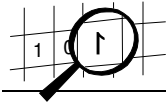
### **Aufgabe**

#### 1. Fehlerschutzcodierung der Konfigurationsdaten

In einem ersten Schritt ist das in der Studienarbeit erstellte Programm gemäss Beurteilungsblatt zu überprüfen und zu überarbeiten.

Für den (15, 11)-Hamming-Code, den um ein Paritätsbit erweiterten (16, 11)-Code und schliesslich für den gesamten Blockcode sind für den Fall zufällig verteilter Bitfehler separate Simulationsprogramme zum Ermitteln der Leistungsfähigkeit als Funktion der Bitfehlerrate zu erstellen. Die ermittelten Simulationswerte sind mit theoretischen, auf der Wahrscheinlichkeitsrechnung beruhenden Werten zu vergleichen.

Die Decodierung des Blockcodes ist auf einem digitalen Signalprozessor zu implementieren. Zusammen mit einem PC-Programm für die Codierung ist eine Vorführanordnung zu realisieren, welche die Leistungsfähigkeit und Grenzen des Codes durch Setzen einzelner Bitfehler auf praktische Weise erproben lässt. Dazu sollen zusätzlich auch Zwischenresultate der Verarbeitung sichtbar gemacht werden.



Die einzelnen Codes und die Ermittlung ihrer Leistungsfähigkeit sind ebenso wie das Erarbeiten der Vorführeinrichtung übersichtlich und anschaulich zu dokumentieren.

## 2. Fehlerschutzcodierung der Verwaltungsdaten

Ausgehend vom theoretischen Erarbeiten der Grundlagen in der Studienarbeit ist zunächst ein Programm für den (15, 7)- BCH-Code zu erstellen, mit dem zwei Bitfehler in einem 15 Bit breiten Datenwort korrigiert werden können. Mit diesem Programm ist sicherzustellen, dass die praktische Anwendung der Theorie umfassend verstanden wird. Dazu muss das Programm so gestaltet werden, dass die Leistungsfähigkeit und Grenzen des Codes durch Setzen einzelner Bitfehler auf praktische Weise erprobt werden können und dass zugleich auch ein Einblick in Zwischenresultate der Verarbeitung ersichtlich ist.

Im einem nächsten Schritt erfolgt eine Erweiterung zu einem Simulationsprogramm, das für den Fall zufällig verteilter Bitfehler die Leistungsfähigkeit des Codes als Funktion der Bitfehlerrate für eine grosse Stichprobe ermitteln lässt. Die so ermittelten Simulationswerte sind schliesslich mit theoretischen, auf der Wahrscheinlichkeitsrechnung beruhenden Werten zu vergleichen. Sowohl das Programm selbst wie auch die Ermittlung der Leistungsfähigkeit des Codes sind auf übersichtliche und anschauliche Weise zu dokumentieren.

Schliesslich ist die Lösung der eigentlichen Aufgabe, die Fehlerschutzcodierung der 27 x 16 Bit Nutzdaten mit einem BCH-Code geeigneter Länge, in der gleichen Art und Weise zu erarbeiten und zu dokumentieren wie soeben für den (15, 7)- Code ausführlich dargelegt wurde.

## Literatur

- [1] Shu Lin, Daniel J. Costello: Error Control Coding: Fundamentals and Applications, Prentice Hall, Englewood Cliffs, NJ, 1983
- [2] W. Wesley Peterson, E.J. Weldon: Error-Correcting Codes, 2<sup>nd</sup> edition, MIT Press, Cambridge, MA, 1972

## Bericht

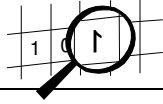
Über die Arbeit ist ein Bericht zu verfassen. Dabei sind die Richtlinien der Abteilung für Elektrotechnik für das Erstellen von Berichten einzuhalten. Alle verwendeten Quellen sind im Literaturverzeichnis des Berichts anzugeben (Hinweis im Text). Der Bericht ist im Doppel abzugeben; ein Exemplar verbleibt am Labor für Digitale Signalverarbeitung des ITR. Die erstellten Programme und der Text des Berichts sind auf einer beschrifteten Diskette (mit Nummer der Studienarbeit) beizulegen.

## Termine

|  |                      |
|--|----------------------|
| Ausgabe der Aufgabenstellung:                | 19.10.1998           |
| Abgabe der Zusammenfassung der Diplomarbeit: | 4.12.1998            |
| Abgabe des Berichts zur Diplomarbeit:        | 4.12.1998, 17.00 Uhr |

## Kontaktadresse

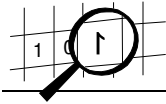
Bernafo AG  
A. Schaub



Eichtalstr. 55  
8634 Hombrechtikon  
Tel. 055 - 264 13 51  
Fax. 055 - 264 13 54

Rapperswil, 15. Oktober 1998

A. Schaub



### 3 EINLEITUNG

In der zweiten Semesterarbeit haben wir uns bereits mit dem Thema 'Forward Error Correction' beschäftigt. Sie ist Grundlage für diese Diplomarbeit und Voraussetzung für den Bericht, in dem kaum mehr auf die theoretischen Aspekte eingegangen wird.

Die Arbeit besteht im Wesentlichen aus drei Aufgaben:

- Die Fehlerschutzcodierung für Konfigurationsdaten auf dem DSP
- Die Fehlerschutzcodierung für Verwaltungsdaten auf dem PC
- Die Auswertung der Codes

Wie bereits in der Aufgabenstellung beschrieben handelt es sich bei den Konfigurationsdaten um 127·11 Bit Nutzdaten, die in einem Bereich von 128·16 Bit gespeichert werden. Zum Schutz der Nutzdaten stehen also für 11 Bits jeweils 5 Bits, und für die 127 Worte noch ein zusätzliches Wort, zur Verfügung. Die 27·16 Nutzbits der Verwaltungsdaten werden mit 5·16 Bits geschützt. Die Art der Codierung dieser Daten wurde in der Semesterarbeit ausführlich dokumentiert. In der Diplomarbeit ging es nun darum diese Anwendungen zu implementieren. Die Konfigurationsdaten werden auf dem DSP gespeichert und dort decodiert. Weiter wurde eine Windowsoberfläche programmiert, über welche man die Daten auf dem DSP manipulieren und visualisieren kann. Auf dem 15-7 BCH Code aufbauend, entstand der 1023-943 BCH Code für die Verwaltungsdaten. Zur Veranschaulichung wurde auch für den 15-7 BCH Code eine Windowsoberfläche erstellt. Dabei hat sich Tobias Läubli vorwiegend mit den Konfigurationsdaten auseinandergesetzt, während sich Yves Gross den Verwaltungsdaten widmete.

Eine praktische Anwendung könnte schematisch etwa folgendermassen aussehen:

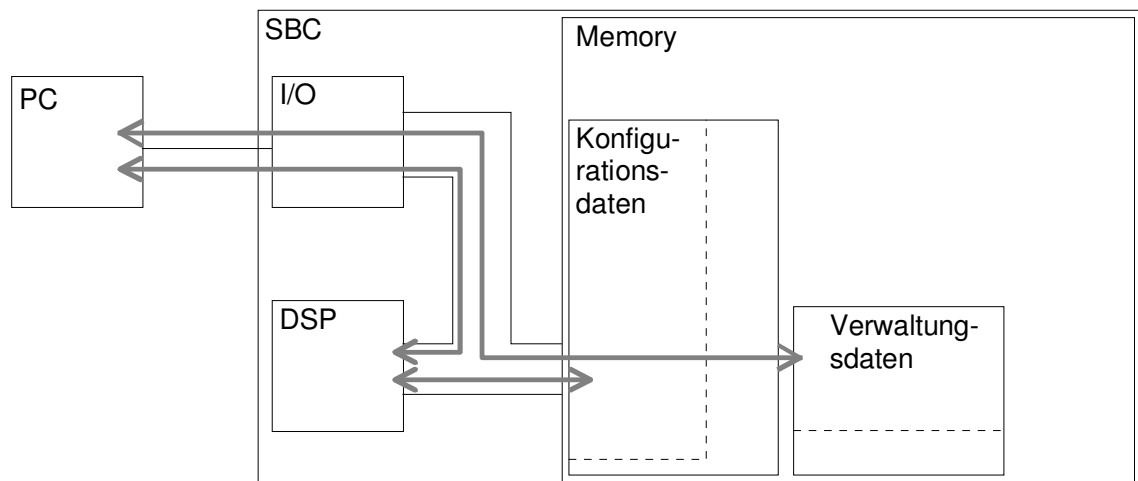
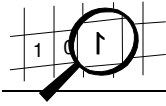
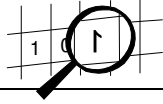


Abbildung 1 Blockschema

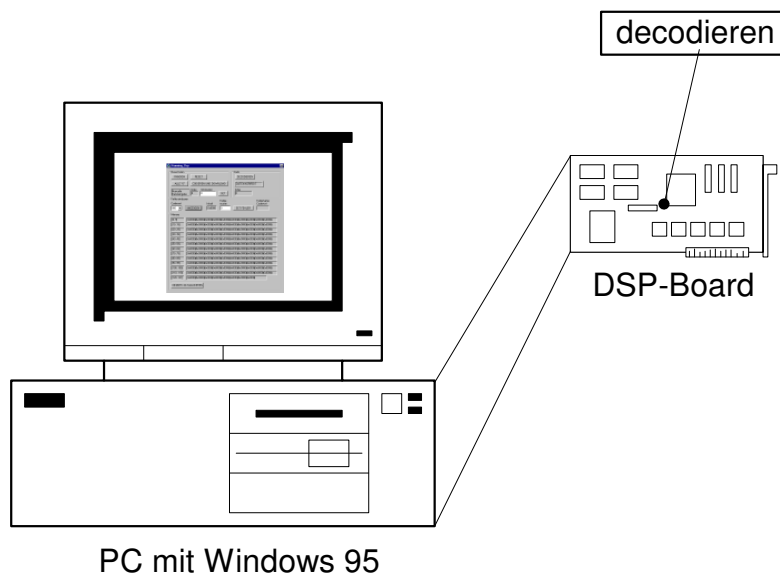


Ein wichtiger Bestandteil ist auch die Auswertung der verschiedenen Codes. Es wird die Leistungsfähigkeit, d.h. die Zuverlässigkeit der Codes untersucht. Dazu werden in die Daten zufällig Fehler, mit unterschiedlichen Wahrscheinlichkeiten eingebaut. Nach dem Decodieren der fehlerhaften Daten kann festgestellt werden, ob die Fehler korrigiert, erkannt oder nicht erkannt wurden. Mit Hilfe der Wahrscheinlichkeitsrechnung können die Resultate zu einem grossen Teil vorhergesagt oder bestätigt werden, was interessante Vergleiche zulässt.



## 4 KONFIGURATIONSDATEN

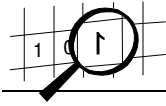
Die Aufgabe dieses Teil war, die in der vorgegangenen Semesterarbeit geschriebenen und getesteten Programme in einer Windows – Anwendung zusammen mit einem DSP einzusetzen. Die Decodierung findet im DSP statt. Alle anderen Funktionen werden im PC ausgeführt.



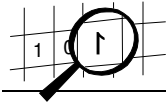
**Abbildung 2** Decodierung der Konfigurationsdaten

### 4.1 RESSOURCEN

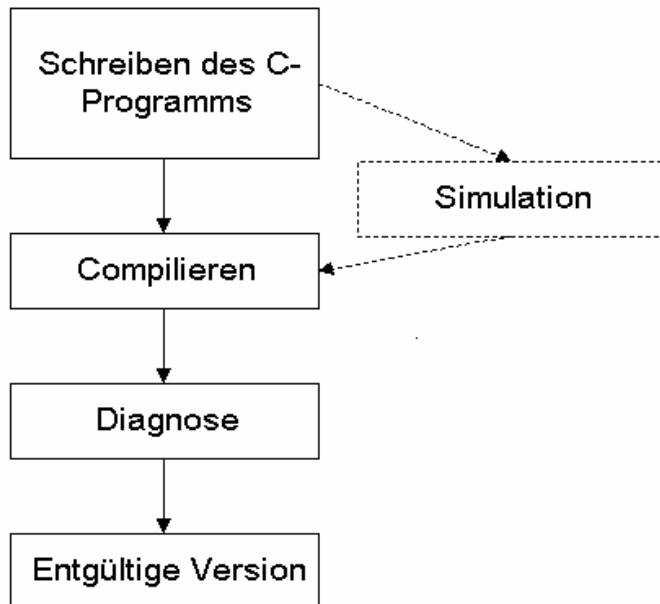
- Hardware:
- IBM kompatibler PC mit freiem ISA – Steckplatz
  - BlackTip Board → DSP – Board ( ISA – Karte ) mit dem Flieskoma – DSP ADSP-21062 ( SHARC ) von der Firma AnalogDevices
- Software:
- Entwicklungs – Umgebung Release 3.3 "ADDS-210XX-SW-PC" von AnalogDevices:
    - C-Compiler
    - Assembler
    - Linker
    - Simulator



- DSP21K Toolkit Release 4.03:
  - Diagnose Tool "DIAG21K" von Bittware
  - Host Interface Library ( HIL ) mit C-aufrufbaren Funktionen für DOS- und Windows – Programme
  - Treiber zur Ansteuerung der Karte
  
- Microsoft Visual C++ 5.0
  
- Betriebssystem: Windows 95



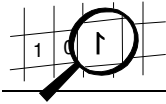
## 4.2 IMPLEMTATION AUF DEM DSP



**Abbildung 3** Ablauf einer DSP – Programmentwicklung

Der oben gezeichnete Ablauf wird in der Praxis mehrmals durchlaufen, da man selten das perfekte Programm auf Anhieb schreiben kann. Um auftretende Fehler zu orten, kann auf das Diagnosetool 'DIAG21K', welches im DSP21K Toolkit beinhaltet ist, zurückgegriffen werden. Mit diesem Tool kann man direkt auf das Memory des Prozessor und auf den Prozessor selber zugreifen.

Compilieren: Der verwendete Compiler ist der GNU – C – Compiler 'G21K'. Es ist eigentlich nicht nur ein Compiler, sondern ein Programm mit C – Preprocessor, Compiler, Assembly – Preprocessor, Assembler und Linker. Das bedeutet, dass mit dem Compiler 'G21K' direkt aus dem C – Code ein ausführbares Programm für den DSP erstellt werden kann.



## 4.3 AUFBAU DES DSP – PROGRAMMS

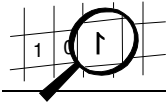
Der C - Code des Decoders konnte grösstenteils aus dem C++ - Code unserer Semesterarbeit abgeleitet werden. Der Unterschied besteht darin, dass wir in der letzten Arbeit objektorientiert programmierten und das aktuelle Programm nun funktionell programmiert werden musste.

Das Programm beinhaltet folgende Funktionen:

- |  |  |
|--|--|
| - <b>void main()</b>                         | Hauptprogramm  |
| - <b>int decodieren_wort( int* codewort)</b> | Decodierfunktion für den (16,11) – Code.<br>Rückgabewert: bei Fehler → 1<br>kein Fehler → 0                  |
| - <b>int decodieren_block()</b>              | Decodierfunktion für den gesamten Block ( (2047,2036) – Code).<br>Rückgabewert: bei Fehler→1<br>keinFehler→0 |

Der Sourcecode der verschiedenen Funktionen des Decoders können im Anhang nachgesehen werden.

Die Variablen, die vom PC aus bearbeitet oder eingesehen werden, müssen im Sourcecode global deklariert werden.



## 4.4 KOMMUNIKATION ZWISCHEN DEM DSP UND PC

Die Kommunikation zwischen dem DSP – Board und dem PC geschieht über Funktionen, die in der Host Interface Library (HIL) definiert sind. Im folgenden sind die Funktionen die in unserer Arbeit eingesetzt sind beschrieben. Alle weiteren Funktionen können im User's Guide des 'DSP21k Toolkit' nachgesehen werden.

Wichtige Variablen:

DSP21K \* **board** → Zeiger auf Struktur mit DSP – Boardinformationen.

Wichtige Funktionen:

DSP21K \* **dsp21k\_open**(int board\_num):

Diese Funktion öffnet das DSP – Board mit der Nummer 'board\_num'. Ebenfalls liefert sie einen Pointer auf die Struktur 'board' zurück, die alle Informationen über das Board enthält und von allen anderen Funktionen benötigt wird.

int **dsp21k\_dl\_exe**(DSP21K \* board, char \* file) :

Mit dieser Funktion wird der Prozessor zuerst zurückgesetzt, und dann wird das ausführbare Programm mit dem Namen 'file' (Extension '\* . 21k') in den Prozessor geladen.

int **dsp21k\_start**(DSP21K \* board):

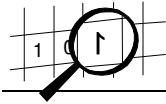
Löst den Start des Programms, welches sich im Prozessor befindet, aus.

int **dsp21k\_close**(DSP21K \* board):

Schliesst das Board und gibt den Speicher frei.

long **dsp21k\_get\_addr**(DSP21K \* board, char \* name):

Gibt die DSP - Adresse der Variablen 'name' zurück. Die Funktion wird in den Methoden, die auf den DSP – Speicher zugeifen, gebraucht.



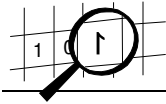
---

void **dsp21k\_dl\_ints**(DSP21K \* board, long dsp\_addr, unsigned int count, int \* val ):

Herunterladen eines Arrays auf den DSP. Die DSP – Adresse 'dsp\_addr' kann mit der Funktion dsp21k\_get\_addr herausgefunden werden. Die Variable 'count' gibt an, wieviele Werte eines Arrays auf den DSP heruntergeladen werden sollen. 'val' ist ein Pointer auf das erste Element des Arrays.

void **dsp21k\_ul\_ints**(DSP21K \* board, long dsp\_addr, unsigned int count, int \* val ):

Mit dieser Funktion werden Arrays vom DSP in den PC geladen. Die DSP – Adresse 'dsp\_addr' kann mit der Funktion dsp21k\_get\_addr herausgefunden werden. Die Variable 'count' gibt an, wieviele Werte eines Arrays auf dem DSP geladen werden sollen. 'val' ist ein Pointer auf das erste Element des Arrays.



## 4.5 OBERFLÄCHE UND DEREN FUNKTIONEN

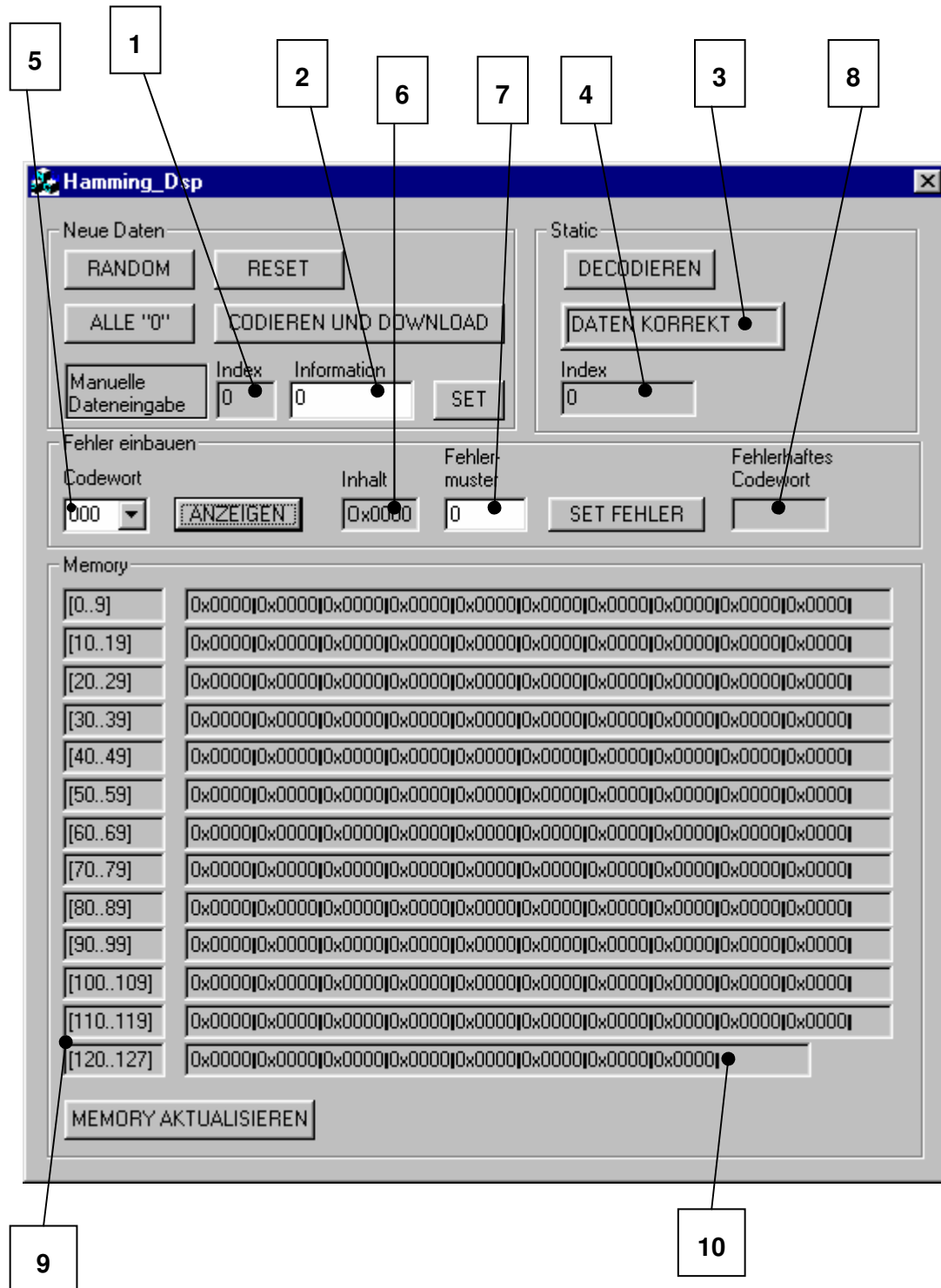
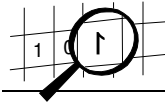


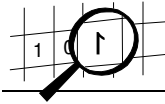
Abbildung 4 Oberfläche des Decodierprogrammes für Windows 95



Mit dieser Oberfläche können folgende Funktionen ausgeführt und Daten visualisiert werden.

| Button                | Funktion  |
|-----------------------|---|
| RESET                 | -setzt das Memory auf dem DSP und das DSP – Board zurück  |
| RANDOM                | -generiert zufällig Daten mittels Funktion 'randdata()' aus der Klasse 'dataset'  |
| ALLE "0"              | -Daten werden auf Null gesetzt  |
| SET                   | -mit dieser Funktion können die Daten manuell gesetzt werden<br>-im Feld 2 kann man die Daten (dezimal) eingeben<br>-durch Anklicken des SET – Buttons wird das Datenwort mit dem Index von Feld 1 mit den Daten von Feld 2 beschrieben → der Index (Feld 2) wird um eins inkrementiert |
| CODIEREN UND DOWNLOAD | -die Daten, die mit 'RANDOM', 'ALLE "0" ' ODER 'SET' erzeugt worden sind werden auf dem PC codiert und danach auf das DSP – Board heruntergeladen   |
| DECODIEREN            | -die Daten, die sich auf dem DSP – Board befinden werden decodiert<br>-je nach Ergebnis werden die Ausgaben 'DATEN KORREKT' oder 'FEHLER!!!' im Feld 3 gemacht und im Feld 4 wird der Index des fehlerhaften Wortes angegeben   |
| ANZEIGEN              | -durch Anklicken dieses Buttons wird das Datenwort mit dem Index, der in der Liste 4 ausgewählt worden ist, im Feld 5 ausgegeben (Hex)  |
| SET FEHLER            | -hier können manuell Fehler gesetzt werden<br>-das Datenwort, das in der Liste 4 ausgewählt worden ist, wird mit dem im Feld 6 eingegebenen Fehlermuster (dezimal) XOR – verknüpft und auf das DSP – Board heruntergeladen und in Feld 7 ausgegeben                                     |
| MEMORY AKTUALISIEREN  | -die Daten vom DSP – Board werden auf den PC geladen und im Feld 9 dargestellt<br>-die Indizes sind in den Feldern 8 dargestellt<br>-diese Funktion wird auch beim Anklicken anderer Buttons ausgeführt   |

**Tabelle 1** Beschreibung der Funktionen des Decodierprogrammes



## 5 VERWALTUNGSDATEN

Laut Aufgabenstellung sind die Verwaltungsdaten mit einem BCH – Code zu schützen. Dazu wird der 1023-943 Code verwendet. Wir wissen, dass mit einem bestimmten Code nicht viele Fehler korrigiert, und gleichzeitig nicht korrigierbare Fehler mit grosser Sicherheit erkannt werden können. Zwischen Fehlererkennung und Fehlerkorrektur muss deshalb immer ein geeigneter Kompromiss gefunden werden. Weil ein unerkannter Fehler in den Verwaltungsdaten nicht die gleich fatalen Folgen nach sich zieht wie in den Konfigurationsdaten, werden die Coderessourcen vorwiegend für die Fehlerkorrektur verwendet.

Um den Einstieg in die Implementation des BCH-Codes zu erleichtern, ist es sinnvoll zuerst den 15-7 Code zu betrachten. Die einzelnen Schritte sind auf dem Papier leicht nachvollziehbar.

### Einige Grundlegende Zusammenhänge des BCH-Codes

|                      |                         |
|----------------------|-------------------------|
| Codewortlänge:       | $n = 2^i - 1$           |
| Anzahl Prüfbits:     | $m = n - k = i \cdot t$ |
| min. Hammingdistanz: | $h_{\min} = 2t + 1$     |
| Galoisfeld:          | $GF(2^i)$               |

### 5.1 DER 15-7 BCH CODE

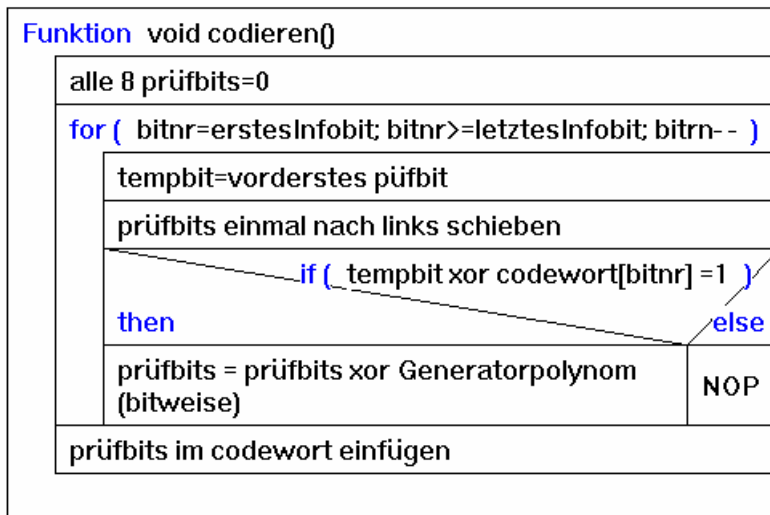
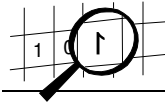
In der vorangegangenen Studienarbeit haben wir schon Klassen geschrieben, um uns die Implementation des BCH – Codes zu erleichtern. Dabei handelt es sich um eine Klasse 'gf', die den einfachen Umgang mit Galoiselementen ermöglicht und um deren Unterklasse 'polynom', die lediglich der Klasse 'gf' dient. Die Galoiselemente werden ja auch als Polynome dargestellt.

Die aus der Theorie bekannten Methoden für die Codierung resp. Decodierung müssen nun in C++ Code umgesetzt werden. Bei der Implementation haben wir jedoch darauf geachtet, dass die Erweiterung auf den grossen 1023-943 Code möglichst problemlos ausfällt. Zur Veranschaulichung haben wir die wichtigsten Funktionen sehr allgemein im Nassi-Shneiderman Diagramm dargestellt.

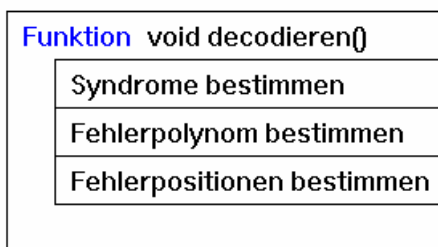
Ein Codewort setzt sich aus den Informationsbits und den Prüfbits folgendermassen zusammen:

|    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| I  | I  | I  | I  | I  | I | I | I | P | P | P | P | P | P | P |

In der Funktion 'codieren' werden zu den Informationsbits die zugehörigen Prüfbits bestimmt. Der Inhalt der alten Prüfbits wird dabei nicht berücksichtigt. Sie werden einfach überschrieben.



Das Decodieren erfolgt in drei Hauptschritten, die wir in der letzten Studienarbeit genau dokumentiert haben. Mittels Pseudocode, dargestellt im Struktogramm könnten diese etwa so aussehen.

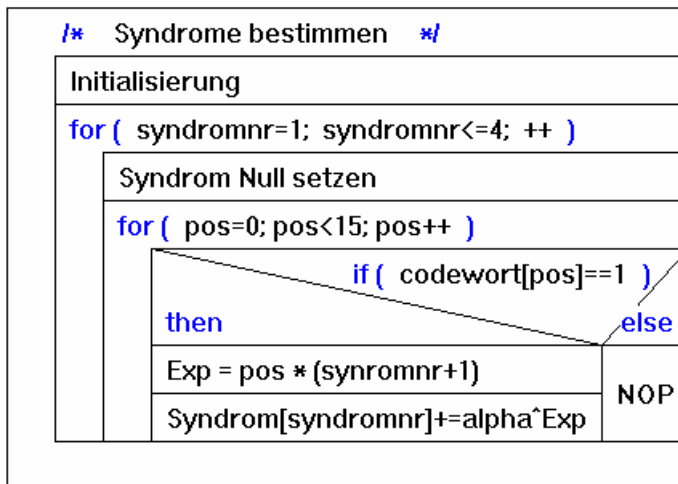
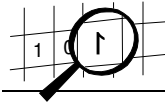


**Zur Erinnerung:**

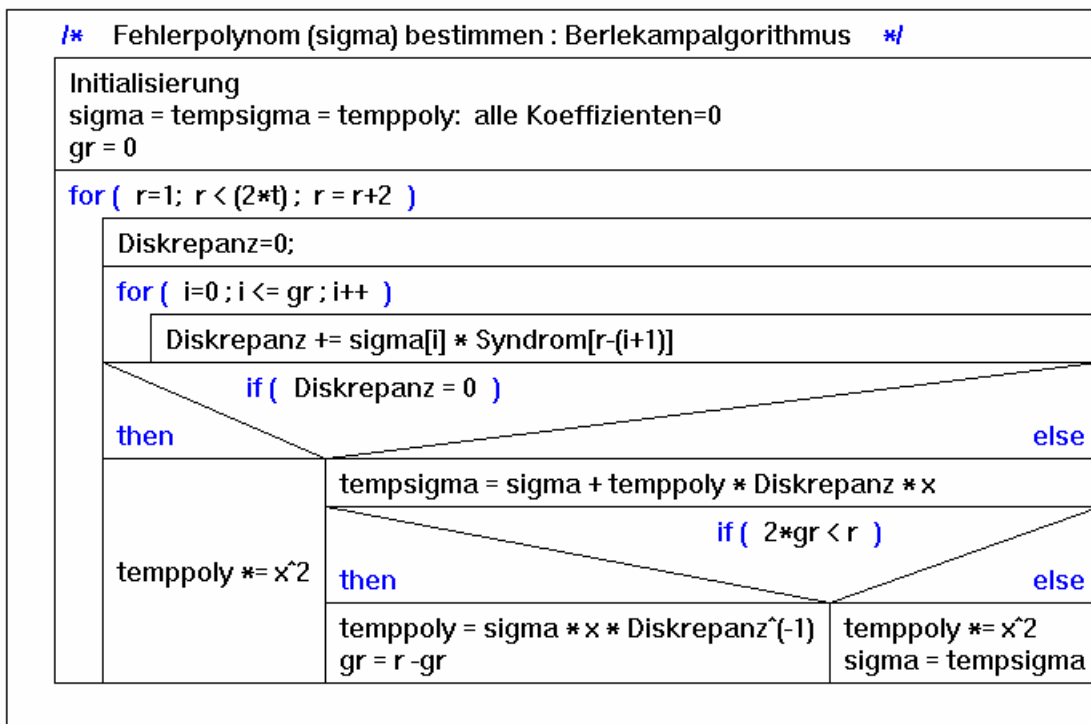
Anzahl Syndrome =  $2 \cdot t = 2 \cdot 4 = 4$   
**S** := Syndromvektor[S<sub>1</sub>, S<sub>2</sub>, S<sub>3</sub>, S<sub>4</sub>]  
r(x) := Codewort

$$\mathbf{H} = \begin{bmatrix}
 \alpha^{14} & \alpha^{13} & \alpha^{12} & \alpha^{11} & \alpha^{10} & \alpha^9 & \alpha^8 & \alpha^7 & \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha & 1 \\
 \alpha^{28} & \alpha^{26} & \alpha^{24} & \alpha^{22} & \alpha^{20} & \alpha^{18} & \alpha^{16} & \alpha^{14} & \alpha^{12} & \alpha^{10} & \alpha^8 & \alpha^6 & \alpha^4 & \alpha^2 & 1 \\
 \alpha^{42} & \alpha^{39} & \alpha^{36} & \alpha^{33} & \alpha^{30} & \alpha^{27} & \alpha^{24} & \alpha^{21} & \alpha^{18} & \alpha^{15} & \alpha^{12} & \alpha^9 & \alpha^6 & \alpha^3 & 1 \\
 \alpha^{56} & \alpha^{52} & \alpha^{48} & \alpha^{44} & \alpha^{40} & \alpha^{36} & \alpha^{32} & \alpha^{28} & \alpha^{24} & \alpha^{20} & \alpha^{16} & \alpha^{12} & \alpha^8 & \alpha^4 & 1
 \end{bmatrix}$$

$$\mathbf{S} = \mathbf{r}(X) \cdot \mathbf{H}^T$$



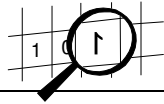
Die Galoiselemente werden meistens als Koeffizienten in Polynomen verwendet. Wir entwarfen deshalb eine Klasse 'gf\_poly', welche es uns erlaubt den Berlekamp-Algorithmus einigermaßen übersichtlich zu programmieren. Im untenstehenden Struktogramm wären die Objekte 'sigma, tempsigma und tempoly' Instanzen von eben dieser Klasse 'gf\_poly', wogegen das Objekt 'Diskrepanz' nur ein Gaoiselement darstellt.



Im letzten Schritt der Decodierung werden die Nullstellen des Fehlerpolynoms 'sigma' gesucht. Dazu werden sämtliche Galoiselemente des Galoisfeldes ( $2^4$ ) der Reihe nach im Fehlerpolynom eingesetzt. Die Fehlerpositionen entsprechen dann den Potenzen der inversen Nullstellen.

**Zur Erinnerung:**

Das Inverse eines Galoiselmentes:  $\frac{1}{\alpha^x} = \alpha^{-x+15}$



$\alpha^{15} = \alpha^0$ , das Inverse von  $\alpha^0$  ist also  $\alpha^0$

```
/* Fehlerpositionen bestimmen */
for ( exp = 0; exp < n; exp++ )
  for ( j = 0; j < Grad v. sigma; j++ )
    potNullst += sigma[j] * alpha^(exp+j)
    if ( potNullst = 0 )
      then
        if ( exp = 0 )
          then Fehlerpos = exp
          else Fehlerpos = n-exp
      else NOP
```

Für das Handling eines einzelnen Codeblocks haben wir ebenfalls eine Klasse definiert: 'code15\_7'. So können die Daten gesetzt (bestimmt oder zufällig), codiert, Fehler eingebaut und wieder decodiert werden. Von einem Hauptprogramm aus können dann die Methoden auf verschiedene Codeblocks angewendet werden.

Der C++ Code für sämtliche Klassen, Haeder- und Programmdateien findet man im Anhang.

Zu Präsentationszwecken waren wir bemüht, eine übersichtliche Windowsoberfläche zu erstellen. Am Beispiel des 15-7 Code soll man die Daten setzen, codieren, mit Fehler versehen und decodieren können.

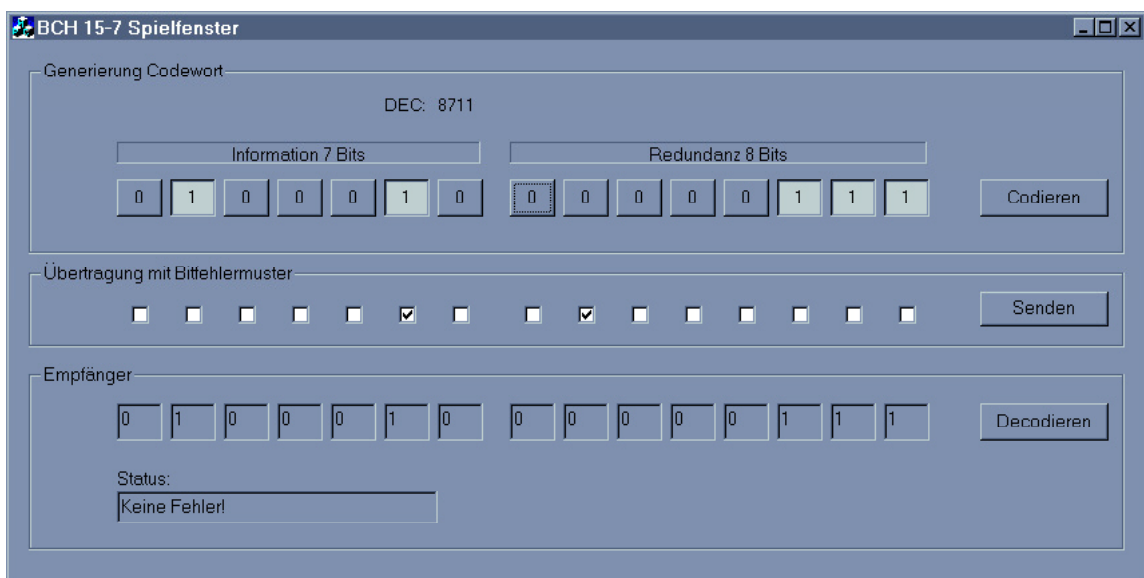
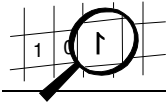


Abbildung 5 15-7 BCH Windowstestprogramm



## 5.1.1 BEMERKUNGEN

Aufgrund der beschränkten Codewortlänge kann man mit dem 15-7 Code recht gut experimentieren. Zuerst sei darauf hingewiesen, dass Fehlererkennung, Fehlerkorrektur, Fehlerpolynom und Syndrome (d.h. die ganze Decodierung) nur vom Bitfehlermuster, nicht aber vom Codewort abhängig ist. Tests können also alle mit dem selben Codewort, z.B mit Null, durchgeführt werden.

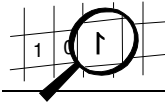
Aus der Theorie wissen wir, dass höchstens zwei Fehler zuverlässig korrigiert werden können. Um unser Programm zu prüfen, haben wir sämtliche mögliche Bitfehlermuster simuliert und aufgenommen.

| Anzahl Fehler<br>k | Anz. Mögliche<br>Bitfehlermuster<br>$\binom{15}{k}$ | korrigiert | erkannt | Desaster |
|--------------------|---|------------|---------|----------|
| 0                  | 1   | 1          | 0       | 0        |
| 1                  | 15  | 15         | 0       | 0        |
| 2                  | 105   | 105        | 0       | 0        |
| 3                  | 455   | 5          | 270     | 180      |
| 4                  | 1365  | 0          | 810     | 555      |
| 5                  | 3003  | 0          | 1500    | 1503     |
| 6                  | 5005  | 0          | 2500    | 2505     |
| 7                  | 6435  | 0          | 3240    | 3195     |
| 8                  | 6435  | 0          | 3240    | 3195     |
| 9                  | 5005  | 0          | 2500    | 2505     |
| 10                 | 3003  | 0          | 1500    | 1503     |
| 11                 | 1365  | 0          | 810     | 555      |
| 12                 | 455   | 0          | 270     | 180      |
| 13                 | 105   | 0          | 0       | 105      |
| 14                 | 15  | 0          | 0       | 15       |
| 15                 | 1   | 0          | 0       | 1        |

**Tabelle 2** 15-7BCH Auswertung aller Bitfehlermuster

Das Erste was uns auffiel, waren die vielen Fälle, in denen der Decodieralgorithmus Fehler erkennen konnte. Wir hatten eigentlich mehr Desasterfälle erwartet. Das zeigt, dass die Hammingdistanz meistens grösser als fünf ist. Wir haben dies zur Kenntnis genommen.

Was uns jedoch neugierig machte, waren die fünf Fälle, in denen sogar drei Fehler korrekt korrigiert wurden. Der Decodieralgorithmus ist also noch ein wenig besser als erwartet. Dies liegt an unserem Programm. Die Theorie besagt nämlich, dass man sich weitere Berechnungen ersparen kann, wenn der Grad des Fehlerpolynomes grösser als zwei ist, weil dann auch sicher mehr als zwei Fehler im Codewort sind und diese somit nicht mit Sicherheit korrigiert werden können. Bei der Programmierung unserer Decodierung sind wir dieser Regel nicht nachgekommen, sondern haben auch Fehlerpolynome dritten Grades nach deren Nullstellen untersucht. Offensichtlich kommen fünf solche Fehlerpolynome vor, bei denen je drei verschiedene Galoiselemente eine Nullstellen sind. Wir haben diese fünf Fälle etwas genauer betrachtet und nachstehend einige Eigenschaften aufgeführt.



| Bitfehlermuster:   | Syndromvektor:          | Fehlerpolynom:               |
|--------------------|-------------------------|------------------------------|
| [0000100'00100001] | [ 0 0 $\alpha^0$ 0 ]    | $\alpha^0 x^3 + \alpha^0$    |
| [0001000'01000010] | [ 0 0 $\alpha^3$ 0 ]    | $\alpha^3 x^3 + \alpha^0$    |
| [0010000'10000100] | [ 0 0 $\alpha^6$ 0 ]    | $\alpha^6 x^3 + \alpha^0$    |
| [0100001'00001000] | [ 0 0 $\alpha^9$ 0 ]    | $\alpha^9 x^3 + \alpha^0$    |
| [1000010'00010000] | [ 0 0 $\alpha^{12}$ 0 ] | $\alpha^{12} x^3 + \alpha^0$ |

Aus diesen Eigenschaften sind zwar gewisse Gesetzmässigkeiten zu erkennen, um jedoch detaillierte Begründungen zu finden, müsste man sich wohl in der Galois- und Restklassentheorie besser auskennen.

## 5.2 DER 1023-943 BCH CODE

Nach den Betrachtungen am 15-7BCH Code waren die Voraussetzungen gegeben um sich dem eigentlichen Problem, dem 1023-943BCH Code zu widmen. Aus der letzten Studienarbeit wissen wir, dass man mit diesem Code bis zu  $t=8$  Fehler korrigieren kann.

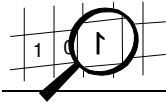
|                      |   |
|----------------------|---|
| Codewortlänge:       | $n = 2^i - 1 = 1023 \rightarrow i = 10$                 |
| Anzahl Prüfbits:     | $m = n - k = i \cdot t = 80 \rightarrow t = 8$          |
| min. Hammingdistanz: | $h_{\min} = 2t + 1 = 2 \cdot 8 + 1 = 17$                |
| Galoisfeld:          | $GF(2^i) = GF(2^{10}) \rightarrow$ im Anhang aufgeführt |

Die C++ Klassen konnten relativ einfach umgeschrieben werden:

```
'gf' → 'gf10'  
'gf_poly' → 'gf10_poly'  
'code15_7' → 'code1023_943'
```

Nicht ganz unerwartet traten jedoch Probleme auf. Die meisten Memberfunktionen wurden viel zu langsam und somit unbrauchbar. Der Hauptgrund dafür waren die Galoiselement – Objekte. Nach jeder Operation musste die Zugehörigkeit von Potenz und Koeffizientenvektor neu bestimmt werden, was natürlich einiges an Rechenaufwand benötigte. Wir haben uns darum entschieden eine global definierte Galoistabelle zu generieren, in welcher diese Zugehörigkeiten festgehalten sind. Diese Variante erhöht zwar den Speicheraufwand, ermöglicht jedoch den langen Code innerhalb eines angemessenen Zeitraumes zu decodieren.

Bis jetzt war ausschliesslich vom 1023-943BCH Code die Rede, obwohl es sich ja bei den Verwaltungsdaten lediglich um 512 ( $32 \cdot 16$ ) Bit handelt. Die Algorithmen werden auch auf 1023Bit angewendet, wobei die Stellen 513 bis 1023 mit Nullen aufgefüllt werden. Das müsste nicht so sein, macht aber Sinn, weil damit die Fehlererkennungsrate erhöht werden kann. Ergibt der Decodieralgorithmus nämlich einen oder sogar mehrere Fehler an den Stellen über 512, ist klar, dass in der Übertragung oder auf dem Speichermedium Fehler entstanden sind.



## 6 AUSWERTUNG

Um nun eine Aussage über die Leistungsfähigkeit der Codes zu machen, müssen die Decodierergebnisse als Funktion der *Bitfehlerrate* aufgezeichnet werden. So entsteht die sogenannte Performancekurve. Man will wissen, in wievielen Fällen die Fehler nicht korrigiert, fälschlicherweise korrigiert oder nicht erkannt werden (Desaster). Es handelt sich dabei um die Restfehlerwahrscheinlichkeit, die zu einem grossen Teil auch über die Theorie der *Binominalverteilung* bestimmt werden kann.

### 6.1 BINOMINALVERTEILUNG

Zufallsexperimente mit nur zwei möglichen Ausgängen (z.B. Fehler erkannt oder Fehler nicht erkannt) führen zur Binominalverteilung. Bei einem solchen Experiment tritt das Ergebnis A mit der Wahrscheinlichkeit p und das zu A komplementäre Ergebnis  $\bar{A}$  mit der Wahrscheinlichkeit  $q = p-1$  ein.

Wird dieses Experiment n-mal nacheinander durchgeführt, dann gilt für die diskrete Zufallsvariable X (Anzahl der Versuche, aus denen das Ergebnis A resultiert) die Binominalverteilung mit der Wahrscheinlichkeitsfunktion

$$f(x) = P(X = x) = \binom{n}{x} \cdot p^x \cdot q^{n-x} \quad (x = 0, 1, 2, \dots, n)$$

und der zugehörigen Verteilungsfunktion

$$F(x) = P(X \leq x) = \sum_{k \leq x} \binom{n}{k} \cdot p^k \cdot q^{n-k} \quad (x \geq 0)$$

(für  $x < 0$  ist  $F(x) = 0$ ). n und p sind dabei die Parameter der Binominalverteilung. Die Kennwerte dieser Verteilung lauten:

Mittelwert:  $\mu = n \cdot p$

Varianz:  $\sigma^2 = n \cdot p \cdot q = n \cdot p \cdot (1-p)$

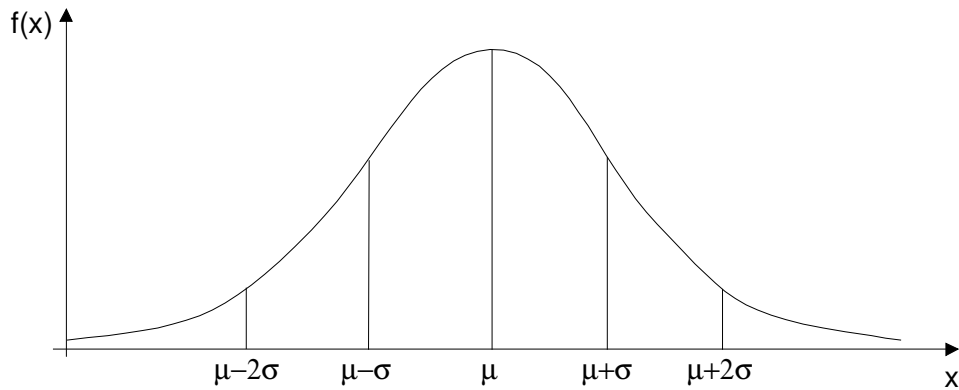
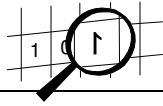
Standardabweichung:  $\sigma = \sqrt{n \cdot p \cdot q} = \sqrt{n \cdot p \cdot (1-p)}$

wobei

p: Konstante Wahrscheinlichkeit für das Erhalten des Ergebnisses A beim Einzelversuch ( $0 < p < 1$ ).

q: Konstante Wahrscheinlichkeit für das Erhalten des zu A komplementären Ergebnisses  $\bar{A}$  beim Einzelversuch ( $q = p-1$ ).

n: Anzahl der Ausführungen des Experiments



**Abbildung 6** Wahrscheinlichkeitsfunktion  $f(x)$

68.3% aller Messwerte liegen im Intervall  $[\mu-\sigma, \mu+\sigma]$

95.5% aller Messwerte liegen im Intervall  $[\mu-2\sigma, \mu+2\sigma]$

99.7% aller Messwerte liegen im Intervall  $[\mu-3\sigma, \mu+3\sigma]$

## 6.2 DER BITFEHLERGENERATOR

Die Bitfehlerrate  $p_b$  ist die Wahrscheinlichkeit, dass ein Bit falsch ist.

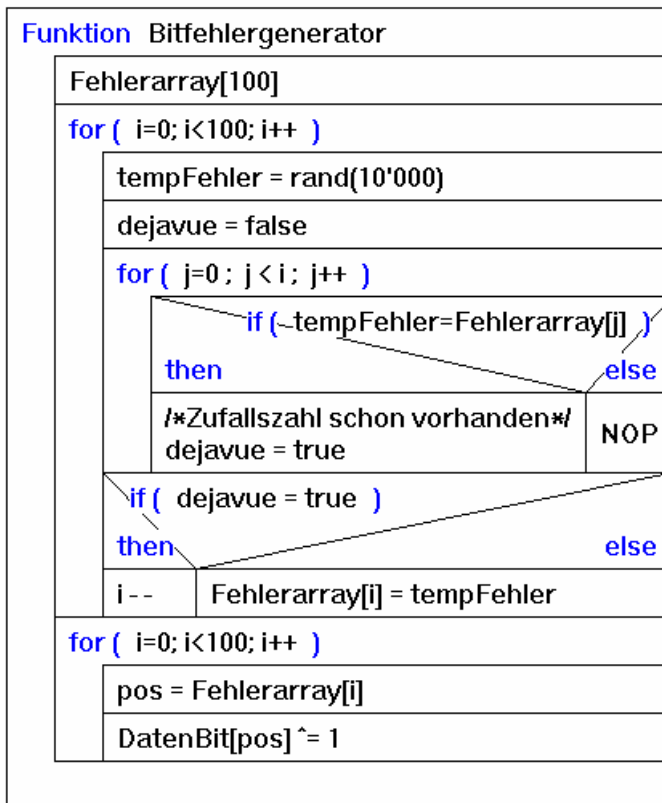
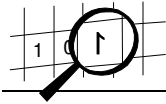
Beispiel:

$p_b = 10^{-2}$ : Im Durchschnitt ist jedes 100ste Bit falsch. Oder, die Wahrscheinlichkeit dass ein bestimmtes Bit falsch ist, beträgt 0.01.

Für die Realisierung eines 'Bitfehlergenerators' gibt es verschiedene Möglichkeiten. Wie im obigen Beispiel gezeigt kann die Bitfehlerrate schon unterschiedlich formuliert werden, was natürlich auch unterschiedlich umgesetzt werden kann.

### 6.2.1 ERSTE VARIANTE

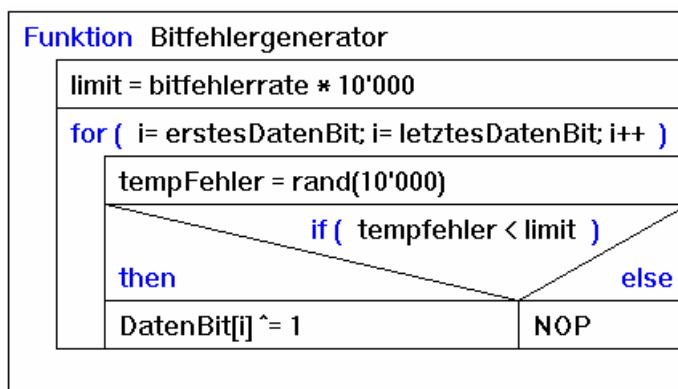
Um eine Bitfehlerrate von  $p_b = 10^{-2}$  zu erhalten werden z.B. 100 Zufallszahlen aus einem Bereich von 10'000 generiert. Diese 100 Zufallszahlen müssen natürlich voneinander verschieden sein. Sie geben die Fehlerstellen aus einem Datensatz von 10'000 Bits an. Ein zugehöriges Struktogramm könnte etwa so aussehen:

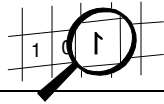


Diese Variante hat einige Nachteile. Sie ist aufwendig, sowohl von der Implementation, wie auch vom Rechen- und Speicheraufwand her. Dazu kommt, dass die Zufälligkeit bei mehreren Durchgängen ein wenig verfälscht wird, weil ja in jedem 10'000er-Block genau 100 Fehler sind. Die Fehler werden künstlich gleichverteilt.

### 6.2.2 ZWEITE VARIANTE

Es wird davon ausgegangen, dass jedes einzelne Bit aus einem Datensatz mit der Wahrscheinlichkeit  $p_b$  falsch ist. Das untenstehende Struktogramm zeigt das Beispiel ebenfalls mit der Bitfehlerwahrscheinlichkeit von  $p_b = 10^{-2}$ :





Diese Variante wird der ersten Variante in allen Problempunkten gerecht. Sie kommt auch der Praxis näher, weil vor jedem Datenbit entschieden wird, ob dieses richtig oder falsch ist. Es kommt deshalb nicht auf die Anzahl der Durchgänge an.

## 6.3 DER 15-11 HAMMING – CODE

Beim (15,11) – Code muss man grundlegend zwei Fälle unterscheiden, da man mit diesem Code entweder einen Fehler korrigieren oder bis maximal zwei Fehler erkennen kann. Nur mit einem zusätzlichen Paritybit kann man einen Fehler korrigieren und gleichzeitig zwei Fehler erkennen. Diese Möglichkeit wurde ebenfalls in unserer Arbeit mit dem (16,11) – Code implementiert und ist im nächsten Kapitel beschrieben.

Einige wichtige Daten des (15,11) – Codes:

-Anz. Bit des Codewortes:  $n = 15$

-Anz. Informationsbit:  $k = 11$

-min. Hammingdistanz:  $d_{\min} = 3$

-Anz. korrigierbarer Fehler:  $t = \frac{d_{\min} - 1}{2}$

-Anz. erkennbarer Fehler:  $e = d_{\min} - 1$

### Fehlerkorrektur:

Beim (15,11) – Code beträgt die minimale Hammingdistanz  $h_{\min} = 3$ . Damit beträgt die maximale Anzahl korrigierbarer Fehler  $t = 1$ .

Restfehlerwahrscheinlichkeit

$$p_d = 1 - \sum_{i=0}^t \binom{n}{i} \cdot p_b^i \cdot (1 - p_b)^{n-i}$$

Für die Simulationen wurde jeweils pro Durchgang zehn Millionen Durchläufe gemacht. Pro Durchlauf wurden die einzelnen Bit eines Codewortes nach einem Zufallsprinzip abhängig von der Bitfehlerrate "gedreht" und danach wurde das Codewort decodiert und nach Fehlern überprüft.

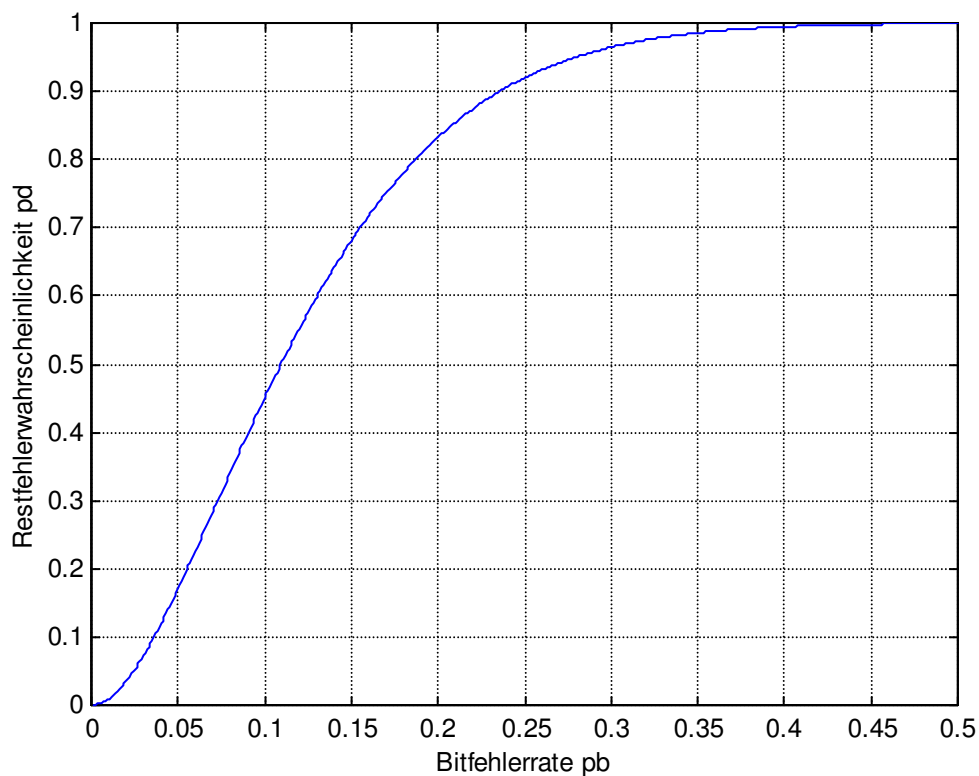
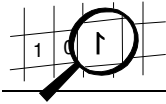


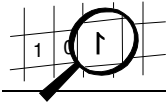
Abbildung 7 Performancekurve 15-11 Code bei Korrektur

| Stützwerte $p_b$ | gezählt (Messwerte) | gerechnet $n \cdot p_d(p_b)$ | Differenz | $\sigma(p_b, n)$ |
|------------------|---------------------|------------------------------|-----------|------------------|
| 0.001            | 979                 | 1'041                        | 62        | 32               |
| 0.005            | 24'954              | 25'138                       | 184       | 158              |
| 0.01             | 95'974              | 96'298                       | 324       | 308              |
| 0.05             | 1'708'759           | 1'709'525                    | 766       | 1'190            |
| 0.1              | 4'508'070           | 4'509'698                    | 1'628     | 1'574            |
| 0.5              | 9'995'094           | 9'995'117                    | 23        | 70               |

Aus der obigen Tabelle kann man ersehen, dass die Differenz zwischen den Werten der Berechnung und der Simulation alle innerhalb von  $2\sigma$ , was gut tolerierbar ist.

#### Fehlererkennung:

Beim (15,11) – Code beträgt die minimale Hammingdistanz  $h_{\min} = 3$ . Damit beträgt die maximale Anzahl erkennbarer Fehler  $e = 2$ .



Restfehlerwahrscheinlichkeit

$$p_d = 1 - \sum_{i=0}^e \binom{n}{i} \cdot p_b^i \cdot (1-p_b)^{n-i}$$

Bei den Simulationen wurde bemerkt, dass zum Teil auch mehr als zwei Fehler erkannt werden können. Wir haben dann alle Fehlermuster durchgetestet, um festzustellen, welche Muster der Decoder erkennt. Die Resultate sehen folgendermassen aus:

| Anzahl Fehler<br>k | Anzahl<br>Bitfehlermuster | erkannt | Desaster |
|--------------------|---------------------------|---------|----------|
| 0                  | 0                         | 0       | 0        |
| 1                  | 15                        | 15      | 0        |
| 2                  | 105                       | 105     | 0        |
| 3                  | 455                       | 427     | 28       |
| 4                  | 1365                      | 1260    | 105      |
| 5                  | 3003                      | 2814    | 189      |
| 6                  | 5005                      | 4725    | 280      |
| 7                  | 6435                      | 6035    | 400      |
| 8                  | 6435                      | 6000    | 435      |
| 9                  | 5005                      | 4690    | 315      |
| 10                 | 3003                      | 2835    | 168      |
| 11                 | 1365                      | 1281    | 84       |
| 12                 | 455                       | 420     | 35       |
| 13                 | 105                       | 98      | 7        |
| 14                 | 15                        | 15      | 0        |
| 15                 | 1                         | 0       | 1        |

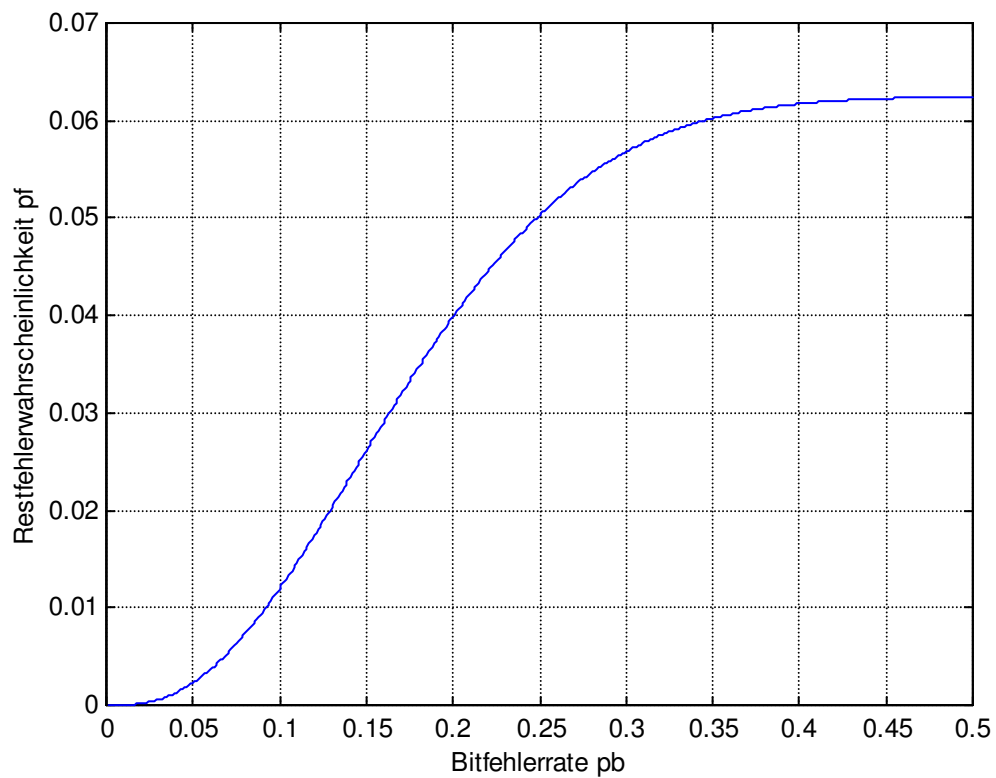
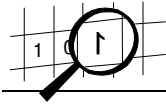
**Tabelle 3** 15-11 Code Auswertung aller Bitfehlermuster

Es muss nun auch die Formel angepasst werden:

$$p_d = 1 - \left( \sum_{i=0}^e \binom{n}{i} \cdot p_b^i \cdot (1-p_b)^{n-i} \right) + \underbrace{427 \cdot p_b^3 \cdot (1-p_b)^{n-3}}_{p(3)} + \underbrace{1260 \cdot p_b^4 \cdot (1-p_b)^{n-4} + \dots}_{p(4)}$$

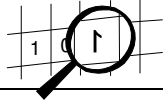
Die Faktoren vor p(3), p(4) und so weiter können in der Tabelle nachgeschaut werden und zwar sind es die Werte in der Spalte 'erkannt'.

| Stützwerte<br>p <sub>b</sub> | gezählt<br>(Messwerte) | gerechnet<br>n · p <sub>d</sub> (p <sub>b</sub> ) | Differenz | σ(p <sub>b</sub> ,n) |
|------------------------------|------------------------|---|-----------|----------------------|
| 0.001                        | 0                      | 0   | 0         | 0                    |
| 0.005                        | 32                     | 34  | 2         | 6                    |
| 0.01                         | 240                    | 258   | 18        | 16                   |
| 0.05                         | 22819                  | 23029   | 210       | 151                  |
| 0.1                          | 119373                 | 119899  | 526       | 344                  |
| 0.5                          | 624013                 | 624695  | 682       | 765                  |



**Abbildung 8** Performancekurve 15-11 Code bei Erkennung

Die simulierten Werte sind auch in dieser Simulation im tolerierbaren Bereich.



## 6.4 DER 16-11 – CODE

Durch das Einfügen eines einfachen Paritybit beim (15,11) – Code, kann die Restfehlerwahrscheinlichkeit um einiges gesenkt werden. Mit diesem Code kann man nun einen Fehler korrigieren und alle geraden ( 2, 4, 6, .... ) Vorkommnisse von Fehlern pro Codewort erkennen.

Einige wichtige Daten des (16,11) – Codes:

-Anz. Bit des Codewortes:  $n = 16$

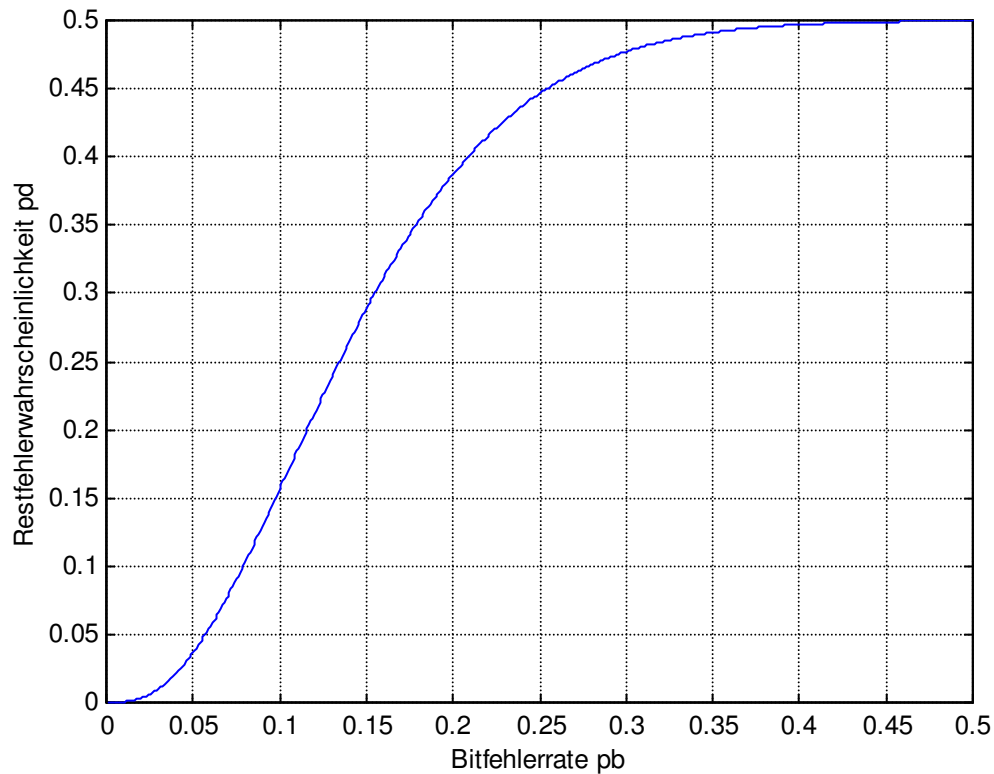
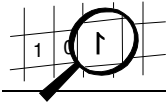
-Anz. Informationsbit:  $k = 11$

-min. Hammingdistanz:  $d_{\min} = 3$

-Anz. korrigierbarer Fehler:  $t = \frac{d_{\min} - 1}{2}$

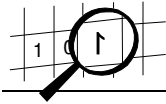
Restfehlerwahrscheinlichkeit

$$p_d = 1 - \sum_{i=0}^t \binom{n}{i} \cdot p_b^i \cdot (1-p_b)^{n-i} - \sum_{j=1}^8 \binom{n}{2 \cdot j} \cdot p_b^{2j} \cdot (1-p_b)^{n-2j}$$

**Abbildung 9** Performancekurve 16-11 Code

| Stützwerte<br>$p_b$ | gezählt<br>(Messwerte) | gerechnet<br>$n \cdot p_d(p_b)$ | Differenz | $\sigma(p_b, n)$ |
|---------------------|------------------------|---------------------------------|-----------|------------------|
| 0.001               | 6                      | 6                               | 0         | 2                |
| 0.005               | 650                    | 656                             | 6         | 25               |
| 0.01                | 4'904                  | 4'918                           | 14        | 70               |
| 0.05                | 371'820                | 367'160                         | 4'660     | 595              |
| 0.1                 | 1'605'660              | 1'565'004                       | 40'656    | 1'149            |
| 0.5                 | 5'311'075              | 4'997'559                       | 313'516   | 1'581            |

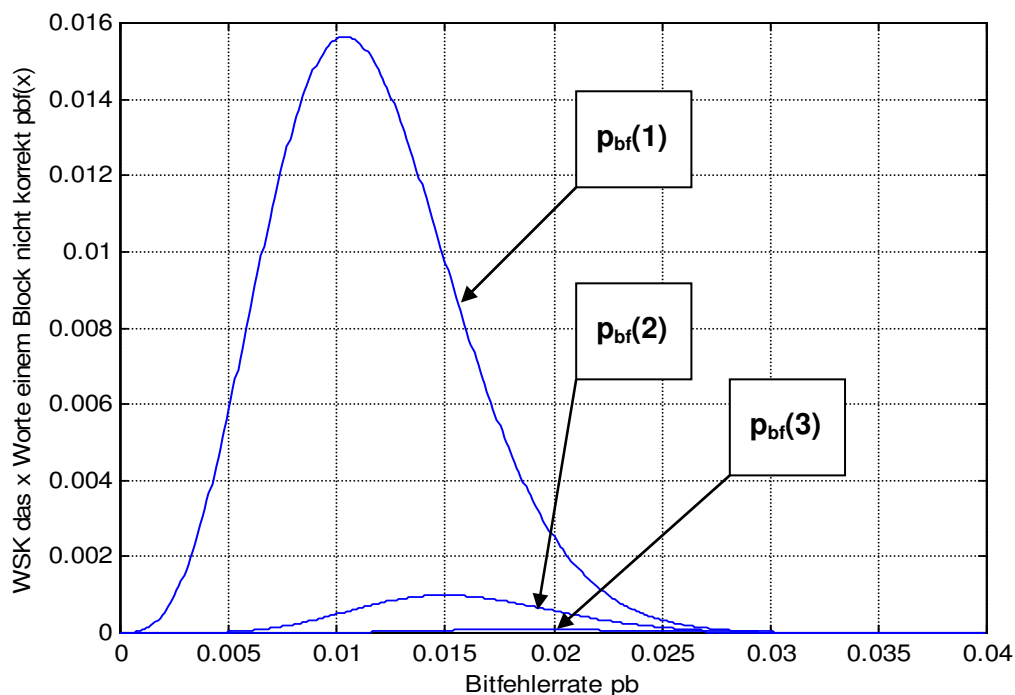
Auffällig bei diesem Code ist, dass die Restfehlerwahrscheinlichkeit nie über einen Wert von 0.5 steigt. Es ist ebenfalls ersichtlich, dass die simulierten Resultate bei den grossen Bitfehlerraten relativ stark von den gerechneten Werten abweichen.



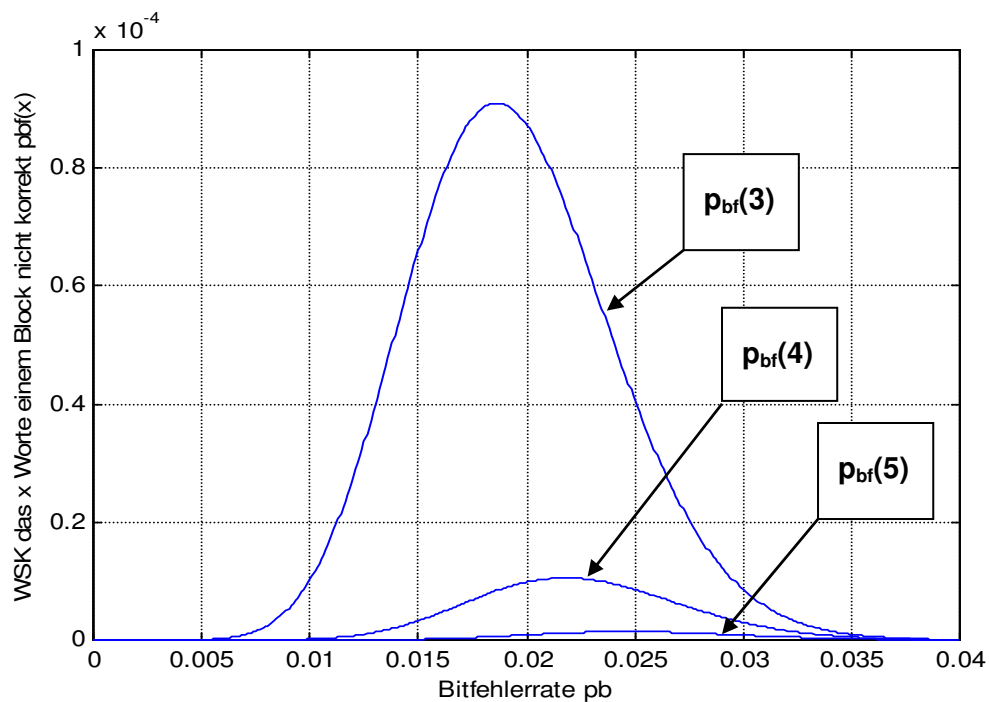
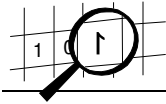
## 6.5 GESAMTER KONFIGURATIONSDATENBLOCK

Die Codierung des gesamten Konfigurationsdatenblock ist aus zwei Codes zusammengesetzt, aus dem (16,11) – Code und einem etwas längeren, dem (2047,2036) – Code. Für den (16,11) – Code ist die Restfehlerwahrscheinlichkeit aus dem obigen Kapitel ersichtlich. Mit dem (2047,2036) – Code werden nun noch die 127x11 Informationsbit überprüft. Durch diese Massnahme können wir die Restfehlerwahrscheinlichkeit der gesamten Konfigurationsdaten erheblich senken. Es ist nicht möglich diesen Wert theoretisch herzuleiten, da der Code nicht ausgelastet ist. In Simulationen konnten wir jedoch feststellen, dass fast keine Fehler den Decoder unbemerkt durchlaufen. Im Durchschnitt sind es etwa fünf Blöcke aus zehn Millionen, die der Decoder fehlerhaft an die Anwendung weitergibt.

Untenstehend sind noch zwei Diagramme dargestellt, die den theoretischen Wert aufzeigen, wie viele Codewörter pro Block fehlerhaft sind in Abhängigkeit der Bitfehlerrate. Aus diesen Diagrammen können wir feststellen, dass eine Bitfehlerrate von  $10^{-2}$  am heikelsten ist. Dies ist damit zu begründen, dass bei kleineren Bitfehlerraten die Fehler meistens korrigiert werden können, und bei grösser werdenden Bitfehlerraten die Chance steigt, dass Fehler erkannt werden.

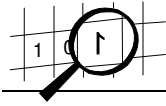


**Abbildung 10** WSK, dass gewisse Anzahl Worte in einem Block fehlerhaft



**Abbildung 11** WSK, dass gewisse Anzahl Worte in einem Block fehlerhaft

Man sieht das  $p_{bf}(1)$  bei einer Bitfehlerrate von  $10^{-2}$  recht gross ist. Mit einer Simulation wollten wir wissen, was der lange Code erkennen kann, falls ein Codewort fehlerhaft ist. Dazu haben wir nacheinander in allen 128 Codewörtern alle möglichen Bitfehlermuster eingebaut. Zu unserer Zufriedenheit haben wir festgestellt, dass solange nur ein Codewort im gesamten Konfigurationsdatenblock fehlerhaft ist, der lange Code alle Fehler erkennt. Gerne hätten wir auch noch die Simulationen zu  $p_{bf}(2)$ ,  $p_{bf}(3)$  usw. durchgeführt, jedoch wären die Simulationen kaum fertig geworden, da sie sehr viel Rechenzeit benötigen.



## 6.6 DER 15-7 BCH CODE

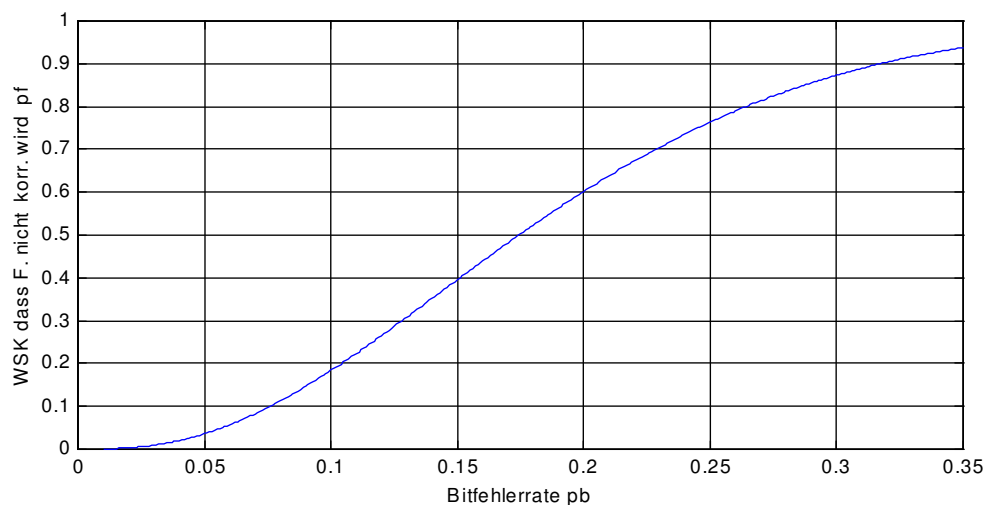
Vom 15-7 BCH Code lohnt es sich zwei Performancekurven zu untersuchen. Es ist dies einerseits die Wahrscheinlichkeit  $p_f$ , dass ein Fehler nicht korrigiert werden kann, und andererseits die Wahrscheinlichkeit  $p_d$ , dass ein Fehler fälschlicherweise korrigiert wird (Desaster). Beide Performancekurven können sowohl rechnerisch, wie auch statistisch mit dem Programm bestimmt werden. Die Resultate werden anschliessend im Rahmen unserer Möglichkeiten diskutiert.

### 6.6.1 WSK, DASS EIN FEHLER NICHT KORRIGIERT WERDEN KANN

Wir wollen zuerst eine Vorhersage treffen. Um die Kurve zu berechnen, sind sämtliche Parameter bekannt. Wir wissen, dass ein Codewort nicht rekonstruiert werden kann, wenn dieses mehr als zwei Fehler aufweist. Um einen kurzen Rechenausdruck zu erhalten, rechnen wir die WSK, dass ein Fehler korrigiert werden kann, und ziehen das Ergebnis von eins ab (komplementäres Ergebnis).

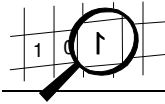
$$p_f = 1 - \sum_{k=0}^2 \binom{15}{k} \cdot p_b^k \cdot (1-p_b)^{(15-k)} = 1 - \{p_b(1-p_b)^{15} + 15 \cdot p_b(1-p_b)^{14} + 105 \cdot p_b^2(1-p_b)^{13}\}$$

Diese Gleichung sei nachfolgend als Funktion im Intervall  $[0, 0.35]$  dargestellt.



**Abbildung 12** Performancekurve 15-7 BCH  $p_f(p_b)$

Es gilt nun, mit dem Programm Simulationen durchzuführen und mit den berechneten Werten zu vergleichen. Wir haben zu einigen Bitfehlerraten jeweils 100'000 Codeworte decodiert, und davon die nicht korrigierten gezählt.



| Stützwerte<br>$p_b$ | gezählt<br>(Messwerte) | gerechnet<br>$100'000 \cdot p_f(p_b)$ | Differenz | $\sigma(p_b, n)$ |
|---------------------|------------------------|---------------------------------------|-----------|------------------|
| 0.05                | 3554                   | 3621                                  | 67        | 59               |
| 0.1                 | 18220                  | 18408                                 | 188       | 123              |
| 0.15                | 39250                  | 39580                                 | 330       | 155              |
| 0.2                 | 59786                  | 60201                                 | 415       | 155              |
| 0.25                | 76089                  | 76389                                 | 300       | 135              |
| 0.3                 | 87159                  | 87322                                 | 163       | 105              |

Auf den ersten Blick scheinen die Messresultate sehr genau mit den berechneten Werten übereinzustimmen. Weichen sie doch nur um wenige Prozente voneinander ab. Betrachtet man jedoch die Differenzen, stellt man fest, dass sie immer zwischen  $\sigma$  und  $3\sigma$  betragen was eher unbefriedigend ist. Sie dürften kaum grösser als  $\sigma$  sein.

Der Fehler liegt für einmal nicht an den Messresultaten, sondern an der Rechnung. Darin haben wir nämlich nicht berücksichtigt, dass unser Decodieralgorithmus in fünf Fällen sogar drei Fehler zu korrigieren vermag. Wir wollen deshalb die Rechnung an dieser Stelle vervollständigen.

$$p_f = 1 - \left\{ p_b(1-p_b)^{15} + 15 \cdot p_b(1-p_b)^{14} + 105 \cdot p_b^2(1-p_b)^{13} + \frac{5}{\binom{15}{3}} \binom{15}{3} \cdot p_b^3(1-p_b)^{12} \right\}$$

Auf die korrigierte Graphik verzichten wir, weil von Auge kein Unterschied festzustellen ist. Die Gegenüberstellung der neu berechneten Werten mit den Simulationen sei jedoch noch aufgeführt.

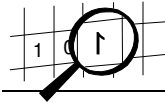
| Stützwerte<br>$p_b$ | gezählt<br>(Messwerte) | neu gerechnet<br>$100'000 \cdot p_f(p_b)$ | Differenz<br>z | $\sigma(p_b, n)$ |
|---------------------|------------------------|---|----------------|------------------|
| 0.05                | 3554                   | 3586                                      | 32             | 59               |
| 0.1                 | 18220                  | 18265                                     | 45             | 123              |
| 0.15                | 39250                  | 39337                                     | 87             | 155              |
| 0.2                 | 59786                  | 59923                                     | 137            | 155              |
| 0.25                | 76089                  | 76144                                     | 55             | 135              |
| 0.3                 | 87159                  | 87130                                     | 29             | 105              |

Dieses Ergebnis sieht nun schon viel genauer aus. Wie man sieht ist die Differenz nie grösser als  $\sigma$ .

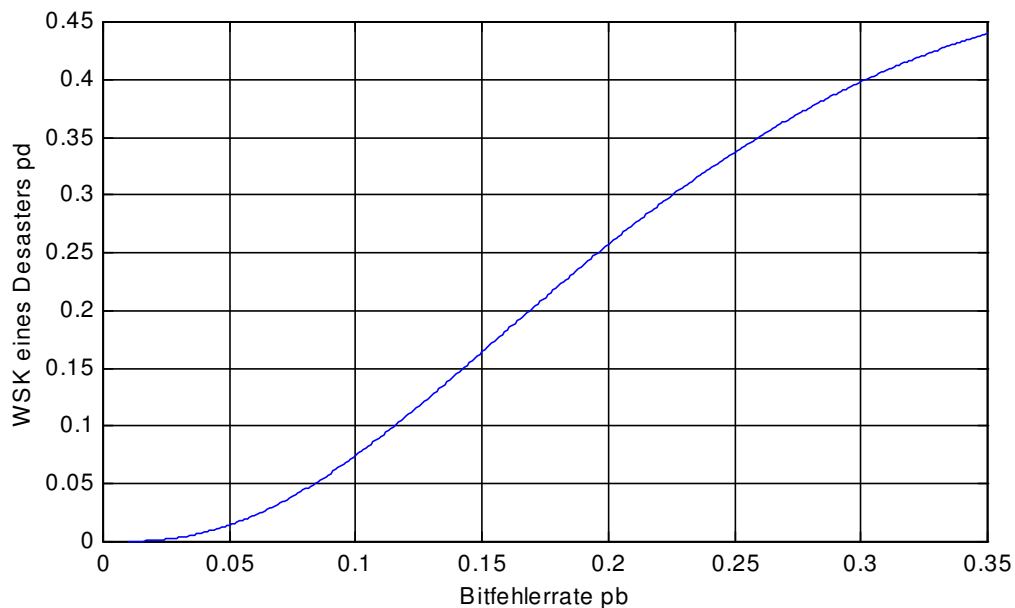
### 6.6.2 WSK, DASS EIN FEHLER FALSCH KORRIGIERT WIRD (DESASTER)

Hier ist es schwieriger eine Vorhersage zu treffen, das heisst die Desasterkurve rein rechnerisch zu bestimmen. Dank Binominalverteilung kann die WSK  $p(k)$  (k Fehler in einem Codewort) bestimmt werden, wieviele davon jedoch Desasterfälle sind, ist nicht so einfach zu berechnen. Im Falle des 15-7 Code jedoch ist die genaue Anzahl der Desasterfälle schon bestimmt worden (siehe Tabelle 2). Sie gehen in der Rechnung genormt als Gewichte der zugehörigen Fehlerwahrscheinlichkeiten ein.

$$p_f(p_b) = \frac{180}{455} p(3) + \frac{555}{1365} p(4) + \frac{1503}{3003} p(5) + \dots + \frac{185}{455} p(12) + p(13) + p(14) + p(15)$$



Zum Vergleich sei die Funktion im selben Intervall graphisch dargestellt, wie das erste Beispiel:



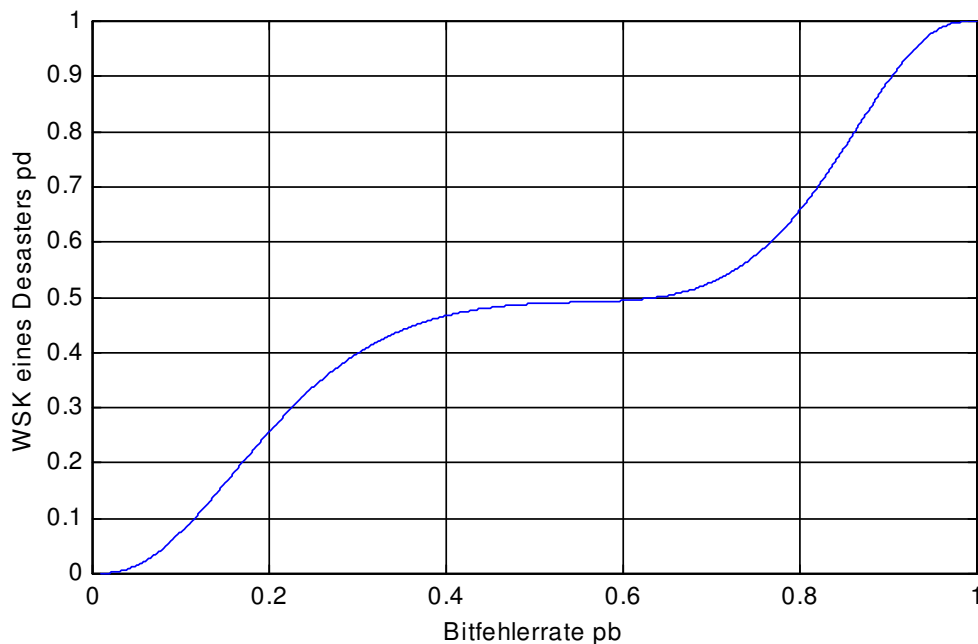
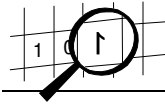
**Abbildung 13** Performancekurve 15-7 BCH  $p_d(p_b)$

Auch in diesem Beispiel haben wir mit unserem Programm für die selben Stützwerte 100'000 Durchgänge laufen lassen und die Desasterfälle gezählt. Zur Kontrolle seien die gerechneten und gezählten Funktionswerte wieder tabellarisch dargestellt:

| Stützwerte $p_b$ | gezählt (Messwerte) | gerechnet $100'000 \cdot p_d(p_b)$ | Differenz | $\sigma(p_b, n)$ |
|------------------|---------------------|------------------------------------|-----------|------------------|
| 0.05             | 1458                | 1443                               | 15        | 38               |
| 0.1              | 7434                | 7461                               | 27        | 83               |
| 0.15             | 16333               | 16429                              | 96        | 117              |
| 0.2              | 25690               | 25736                              | 46        | 138              |
| 0.25             | 33875               | 33735                              | 140       | 150              |
| 0.3              | 39798               | 39813                              | 15        | 155              |

Man sieht, dass die Ergebnisse auch hier sehr gut korrespondieren.

Abgesehen von diesem Vergleich ist es aber auch interessant die ganze Desaster-Performancekurve zu betrachten. Das heisst die Funktion  $p_d(p_b)$  im im Intervall  $[0, 1]$ .



**Abbildung 14** Performancekurve 15-7 BCH  $p_d(p_b)$

In dieser Abbildung ist eine Symmetrie zu erkennen. Sie ist allerdings ein wenig verzerrt, was leicht zu erklären ist. Für kleinere Bitfehlerraten ist die Kurve flacher als bei grösseren. Man vergleiche dazu die Steilheit der Kurve an den Stellen  $p_b = 0.2$  und  $p_b = 0.8$ . Bei kleinen Bitfehlerraten kommt die Fehlerkorrektur zum tragen, bei welcher natürlich keine Desasterfälle vorkommen. Die erwähnte Symmetrie ist übrigens auch in Tabelle 2 zu erkennen. Die wohl eher unerwartete S-Form der Kurve hat damit zu tun, dass der Code nicht 'dichtgepackt' ist. Das heisst, es kommt vor, dass in einem gültigen Codewort fünf Bits verändert werden können, ohne dass daraus wieder ein gültiges Codewort resultiert. Auch dies geht aus Tabelle 2 eindeutig hervor. Diese Eigenschaft, also sozusagen die Architektur des Codes, ist verantwortlich für die Form der Performancekurve.

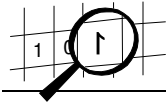
Interessant ist auch, dass die Kurve für hohe Bitfehlerraten gegen 1 geht. Daraus lassen sich einige Schlüsse ziehen:

- Ist die Bitfehlerrate  $p_b = 1$  (d.h. alle Bits sind mit Sicherheit falsch), dann ist die Wahrscheinlichkeit eines Desasters  $p_d = 1$  (d.h. mit Sicherheit ein Desaster).
- Wenn alle Bits eines Codewortes  $C$  falsch sind, erhält man wieder ein gültiges Codewort  $\bar{C}$ . Das muss so sein, da es ansonsten kein Desasterfall wäre.
- Daraus lässt sich schliessen, dass zum Inversen der Informationbits auch das Inverse der Prüfbits gehört.

Diese letzte Aussage lässt sich bequem in unserem Windowsprogramm überprüfen.

## 6.7 DER 1023-943 BCH CODE

Vom 1023-943 BCH Code wollen wir ebenfalls die Performancekurven untersuchen. Der grösse des Codes wegen, ist dies jedoch nicht so einfach wie beim 15-7 BCH Code. Es können nicht alle Bitfehlermuster geprüft werden weil es schlicht zu viele wären. Alleine 50 Fehler könnten auf  $8 \cdot 10^{69}$  verschiedene Arten in den 512 Datenbits



verteilt werden. Auch die Programmalgorithmen können unmöglich von Hand nachvollzogen werden.

Auch hier sind es zwei Kurven, die uns interessieren.

### 6.7.1 WSK, DASS EIN FEHLER NICHT KORRIGIERT WERDEN KANN

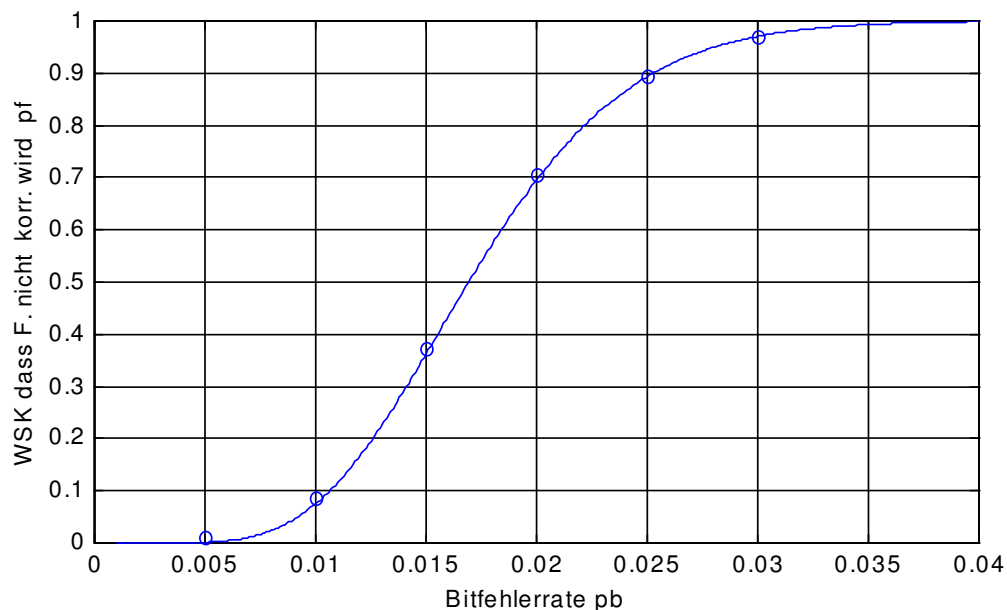
Die Überlegung zur Berechnung dieser WSK  $p_f$  in Abhängigkeit der Bitfehlerrate, ist die selbe wie beim 15-7 BCH Code. Es ist viel weniger aufwendig, die komplementäre WSK  $p_k$ , dass ein Fehler korrigiert werden kann, zu bestimmen.

$$p_f(p_b) = 1 - p_k(p_b)$$

wobei

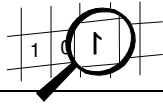
$$p_k(p_b) = \sum_{k=0}^8 \binom{512}{k} \cdot p_b^k \cdot (1 - p_b)^{512-k}$$

Auch in dieser Formel schafft die Grösse des Codes Probleme. Der erste Faktor wird bei  $k > 6$  so gross, dass Matlab darauf hinweist, dass das Resultat nicht genau sein könnte. Im Gegensatz dazu wird der letzte Faktor für grosse Bitfehlerraten sehr klein. Dennoch haben wir die Kurve aufgezeichnet und an einigen Stützstellen mit den Programmresultaten verglichen.



**Abbildung 15** Performancekurve 1023-943 BCH  $p_f(p_b)$

Die durchgezogene Linie ist die mit Matlab berechnete Performancekurve. Für sechs Bitfehlerraten wurden zur Simulation je 5000 Durchläufe gemacht (d.h. 5000-512 Bit Daten mit Fehlern versehen und decodieren). Die Resultate sind in der Abbildung 15 mit Kreislein eingetragen. Man stellt fest, dass die Simulation ziemlich genau mit den Berechnungen übereinstimmt. Die tabellarische Gegenüberstellung soll noch mehr Aufschluss geben:



| Stützwerte<br>$p_b$ | gezählt<br>(Messwerte) | gerechnet<br>$5'000 \cdot p_f(p_b)$ | Differenz | $\sigma(p_f, n)$ |
|---------------------|------------------------|-------------------------------------|-----------|------------------|
| 0.005               | 56                     | 6                                   | 50        | 3                |
| 0.01                | 446                    | 376                                 | 70        | 19               |
| 0.015               | 1875                   | 1813                                | 62        | 34               |
| 0.02                | 3531                   | 3480                                | 51        | 33               |
| 0.025               | 4481                   | 4469                                | 12        | 22               |
| 0.03                | 4859                   | 4854                                | 5         | 12               |

Die Tabelle zeigt, dass die Resultate, vor allem im Bereich kleiner Bitfehlerraten, doch nicht so gut korrespondieren, wie einem dies die Graphik vermuten lässt.

Neben dem bereits erwähnten Berechnungsproblem, sind allerdings noch einige Bemerkungen hinzuzufügen.

- Aufgrund der grossen Codelänge und der zeitraubenden Algorithmen war es kaum möglich, mehr als 5000 Durchläufe pro Stützwert zu machen. Die Werte aus der Simulation sind deshalb nicht sehr aussagekräftig.
- Ebenfalls der grossen Codelänge wegen, besteht die Möglichkeit, dass die Periodizität des Zufallsgenerators die Werte beeinflusst.
- Bei kleinen Bitfehlerraten macht sich noch ein weiterer Effekt bemerkbar. Man betrachte dazu die zweite Variante des Bitfehlergenerators (Kapitel 6.2). Der ganzzahligen Variablen 'limit' wird eigentlich eine reelle Grösse zugewiesen. In C++ werden bei dieser Typenkonversion einfach die Nachkommastellen abgeschnitten. Da bei kleinen Bitfehlerraten nur wenige Zufallszahlen einen Bitfehler bewirken, fällt es ins Gewicht, ob das Limit um Eins höher liegt oder nicht.

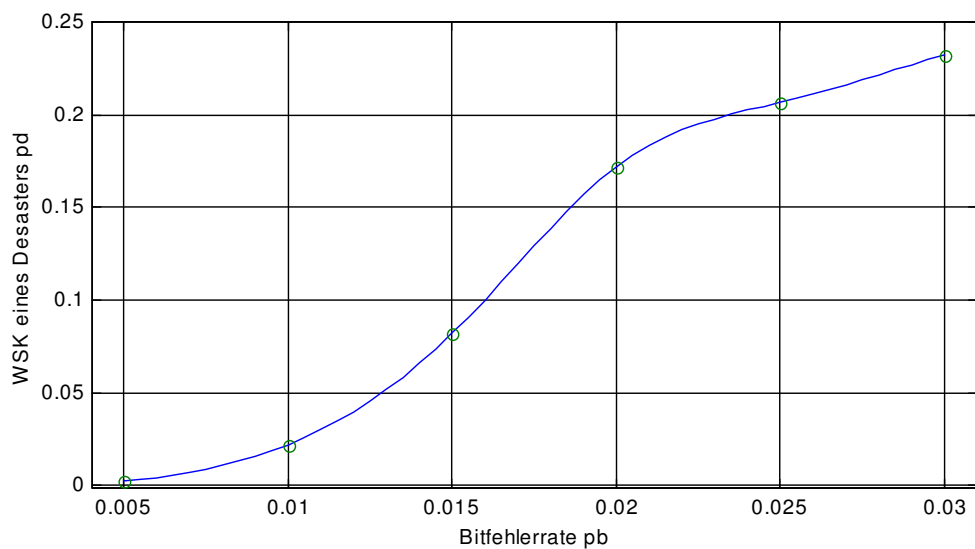
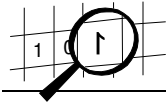
### 6.7.2 WSK, DASS EIN FEHLER FALSCH KORRIGIERT WIRD (DESASTER)

Die WSK  $p_d$  eines Desasters ist nun nicht mehr rechnerisch zu bestimmen. Man müsste dazu –analog zu Tabelle 2- alle möglichen Bitfehlermuster durchgehen und die Desasterfälle zählen um in der Rechnung die Terme zu gewichten (siehe Kapitel 6.6.2). Es kann also keine Vorhersage getroffen werden.

Um trotzdem eine Performancekurve zu erhalten, haben wir zu den bekannten sechs Stützstellen die Desasterfälle gezählt. Auch für diesen Versuch haben wir zu jeder der sechs Bitfehlerraten 5'000 Durchgänge gemacht.

| Stützwerte<br>$p_b$ | gezählte<br>Desasterfälle<br>(von 5'000) | WSK eines<br>Desasters<br>$p_d$ |
|---------------------|--|---------------------------------|
| 0.005               | 14                                       | 0.0028                          |
| 0.01                | 111                                      | 0.0222                          |
| 0.015               | 410                                      | 0.082                           |
| 0.02                | 860                                      | 0.172                           |
| 0.025               | 1034                                     | 0.2068                          |
| 0.03                | 1162                                     | 0.2324                          |

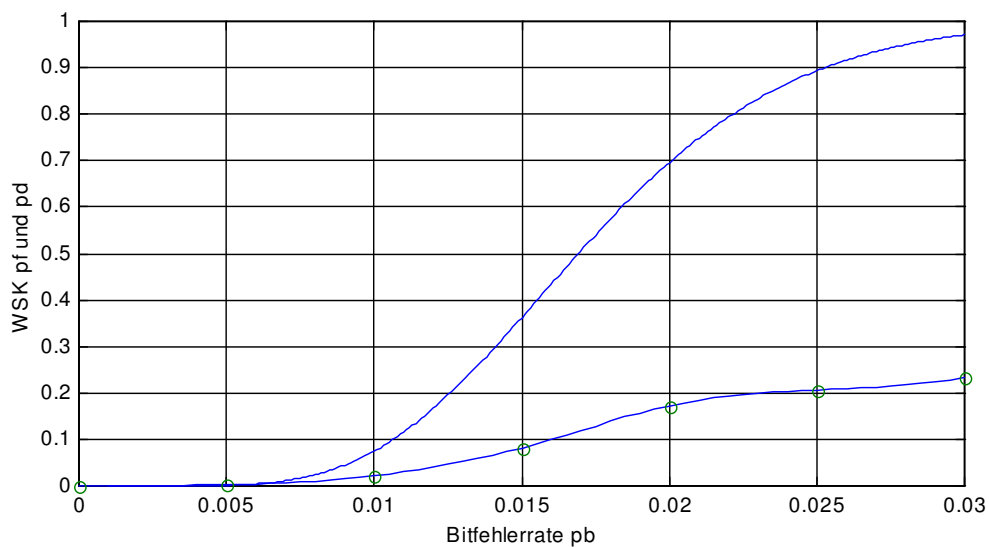
Hieraus können natürlich keine quantitative Aussagen gemacht werden. Dafür sind es viel zu wenige Werte. Um jedoch eine Grössenordnung anzugeben, reichen diese Daten vollends.



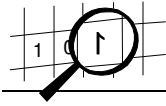
**Abbildung 16** Performancekurve 1023-943 BCH  $p_d(p_b)$

Auch die Kurve ist in dieser Form nur zu qualitativen Aussagen zu gebrauchen. Sie wurde lediglich aus den sechs Stützpunkten interpoliert.

Zum direkten Vergleich seien die beiden Kurven in der selben Graphik noch einmal dargestellt.



**Abbildung 17** Performancekurve 1023-943 BCH  $p_f(p_b)$  und  $p_d(p_b)$



## 7 SCHLUSSWORT

Das Know How, das wir aus der zweiten Studienarbeit mitbrachten, hat uns einen raschen Einstieg in diese Diplomarbeit ermöglicht. Ohne anderweitig schulische Verpflichtungen konnten wir uns voll auf die Arbeit konzentrieren. Dennoch gingen die sieben Wochen schnell vorbei.

Wir haben bei der Implementation der Codes die Vor- und Nachteile der verschiedenen Codes aus mehreren Sichten gesehen. Aus Sicht des Programmierers sind die Unterschiede im Programmieraufwand sehr gross. Das macht sich natürlich auch in der Rechenzeit der Programme bemerkbar, was wiederum aus Sicht des Anwenders festzustellen ist. Auch die Leistungsfähigkeit der Codes sind sehr verschieden, wobei man nicht sagen kann welcher der Beste ist. Die Stärken der Codes sind unterschiedlich verteilt.

Die Diplomarbeit war im Gegensatz zur letzten Studienarbeit eher praktisch ausgerichtet, worüber wir sehr froh waren. Dies war auch nur deshalb möglich, weil wir uns in der Theorie schon gut auskannten. Die beiden Arbeiten ergänzen sich sehr gut zu einem umfangreichen Projekt, welches einem zukünftigen Anwender eine gute Stütze sein kann.

Rückblickend sind wir mit dem Resultat zufrieden. Der Aufgabenstellung sind wir in allen Punkten gerecht geworden. Einige interessante Aspekte hätten wir zwar gerne noch länger untersucht. Um jedoch nicht zu viel Zeit zu verlieren, mussten diese Versuche teilweise verschoben oder abgebrochen werden.

Während den sieben Wochen stand uns unser Betreuer, Herr A.Schaub, für unsere Anliegen stets zur Verfügung. Ihm und den Herren A.Schüeli und P.Roffler möchten wir für ihre Unterstützung danken.

Rapperswil, den 4. Dezember 1998

Die Autoren:

Yves Gross

Tobias Läubli