Diplomarbeit

Christian Koch

Entwicklung und Erprobung einer Software zur Rahmensynchronisation und Qualitätsüberprüfung von CCSDS-kompatiblen Satellitendaten

Fakultät Technik und Informatik
Studiendepartment Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Christian Koch

Entwicklung und Erprobung einer Software zur Rahmensynchronisation und Qualitätsüberprüfung von CCSDS-kompatiblen Satellitendaten

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Informations- und Elektrotechnik
Studienrichtung Informationstechnik
am Studiendepartment Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Jürgen Reichardt
Zweitgutachter : Prof. Dr. Karl-Ragmar Riemschneider

Abgegeben am 27. Februar 2008

Christian Koch

Thema der Diplomarbeit

Entwicklung und Erprobung einer Software zur Rahmensynchronisation und Qualitätsüberprüfung von CCSDS-kompatiblen Satellitendaten

Stichworte

Reed-Solomon Codierung, Rahmensynchronisation, Bytesynchronisation, CCSDS Standard, Bitschlupf, Interleaving, C++

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung eines Rahmensynchronisators in Software. Dazu werden die Grundlagen zur Rahmensynchronisation, der CCSDS Standard und zusätzlich die Reed-Solomon Codierung, behandelt, um die gefundenen Rahmen auf Fehler zu überprüfen. Die erarbeiteten Informationen werden in Flussdiagrammen zur besseren Implementierung umgesetzt und es wird ein Dateimanagement entwickelt, das die Schnittstelle zwischen Festplatte und Rahmensynchronisator darstellt. Am Ende wird die Software verifiziert und ein Geschwindigkeitstest durchgeführt.

Christian Koch

Title of the paper

Development and test of a sofware that framesynchronizes and checks the quality of CCSDS-compatible satellitedata

Keywords

Reed-Solomon Coding, framesynchronization, bytesynchronization, CCSDS standard, bitslip, interleaving, C++

Abstract

This report describes the development of a framesynchronizer in software. So the basis about framesynchronisation, CCSDS standard and additional RS-Coding is described, to check the founded frames on errors. For better implementation into software this informations are turned into flow-charts and a datamanagement will be designed as an interface between the synchronizer and the hard disk. At the end the software will be tested.

Inhaltsverzeichnis

1 Einleitung

Das Thema der Diplomarbeit: „Entwicklung und Erprobung einer Software zur Rahmensynchronisation und Qualitätsüberprüfung von CCSDS-kompatiblen Satellitendaten“, ist in enger Zusammenarbeit mit dem Deutschen Zentrum für Luft- und Raumfahrt e.V. - Außenstelle Neustrelitz entstanden. An diesem Standort konzentrieren sich spezielle Forschungsvorhaben insbesondere auf den Empfang, die Verarbeitung und Archivierung von Satellitendaten, sowie auf die Entwicklung und den Betrieb von erforderlichen Bodensegmenten, die eine satellitengestützte Ortung und Navigation und ausgewählte Probleme der Fernerkundung der Erde und des erdnahen Raumes unter regionalen und globalen Aspekten unterstützen.

Das Ziel dieser Arbeit ist es ein Programm zu entwickeln, dass eine Telemetry-Datei synchronisiert und als Ergebnis die synchronisierte Datei sowie eine Tabelle mit Qualitätsparametern liefert. Eine Telemetry-Datei ist eine Datei, die die Messergebnisse der Sensoren eines Satelliten enthält.

Zusätzlich werden die folgenden Festlegungen gemacht, um die Arbeit einzugrenzen und genauer zu spezifizieren:
Die Software wird für den Einsatz auf einem digitalen Datenrekorder entwickelt. Jedoch soll die Software in Zukunft auch auf anderen Systemen zum Einsatz kommen. Die Inbetriebnahme des Datenrekorders ist nicht Thema dieser Diplomarbeit, sodass die zu synchronisierenden Telemetry-Daten bereits als Dateien auf der Festplatte zur Verfügung gestellt werden. Gleichzeitig wird damit festgelegt, dass keine Echtzeitanforderung vorgesehen ist. Diese Arbeit dient einerseits zum experimentellen Vergleich einer Softwarelösung mit der einer bestehenden in Echtzeit ablaufenden Hardwarelösung. Deshalb soll die Software die Daten mindestens mit einer durchschnittlichen Geschwindigkeit von 20 MB/s bearbeiten. Anderseits soll die Software dazu dienen, ausgewählte Datensätze zu untersuchen.

In Anlehnung an den kürzlich gestarteten Radarsatelliten Terra-Sar X, soll zur Qualitätsüberprüfung auch die in den Telemetry-Daten enthaltene Reed-Solomon Codierung in geeigneter Weise genutzt werden. Die nötigen Testdateien stammen von diesem Satelliten und orientieren sich dabei an dem im Thema festgelegten CCSDS-Rahmenformat.

Die Hinführung zum Thema und der damit verbundenen näheren Erläuterung der Thematik erfolgt mit den Grundlagen im zweiten Kapitel. Auf diese Grundlagen aufbauend wird im vierten Kapitel das Konzept erarbeitet. Dieses umfasst die Wahl der Programmiersprache, als auch die Entwicklung von Flussdiagrammen für die in Kapitel fünf vorgestellte Implementierung. In diesem Kapitel werden die Flussdiagramme in Software umgesetzt und in diesem Zusammenhang auf besondere Lösungen eingegangen. Im sechsten Kapitel wird die Software getestet. Dazu gehört ein Verifikationstest als auch ein Geschwindigkeitstest auf dem Zielsystem. Zum Schluss werden im siebten Kapitel ein Rückblick, ein Fazit sowie ein Ausblick gegeben.

2 Grundlagen

2.1 Die Übertragungsstrecke

Zunächst wird kurz der Weg vom Satelliten zum Datenrekorder beschrieben und danach mögliche Fehlerquellen auf dieser Strecke aufgezeigt. Für die Entwicklung eines Rahmensynchronisators muss bekannt sein, welche Arten von Fehlern im empfangenen Datenstrom auftreten können.

Die Übertragungsstrecke beginnt beim Satelliten, der mittels Sensoren aufgezeichnete Telemetry-Daten zur Bodenstation überträgt. Am Boden wird dieses digitale Signal durch eine Parabolantenne empfangen und über einen Lichtwellenleiter ins nahe gelegene Gebäude übermittelt. Hier erfolgt die Demodulation und Bitsynchronisation. Zum Schluss werden die Telemetry-Daten auf der Festplatte des Datenreckorders aufgezeichnet.

Die Satelliten, die Daten an die Bodenstation senden, befinden sich im Low Earth Orbit (LEO) in einer Entfernung von 600 – 800 km. Dabei unterscheiden sich die Satelliten in zwei Klassen: Satelliten mit Speicher an Bord und ohne Speicher an Bord. Die Satelliten mit Speicher an Bord beginnen die Übertragung der Daten erst, wenn diese in Reichweite der Bodenstation sind. Zusätzlich werden zu Beginn Idle-Rahmen übertragen. Diese Rahmen besitzen keinen Inhalt und dienen dazu die Geräte der Bodenstation auf das Signal zu synchronisieren, bevor die eigentlichen Rahmen mit Daten gesendet werden. Bei Satelliten ohne Bordspeicher werden die Daten kontinuierlich übertragen, auch wenn gerade keine Bodenstation in der Nähe ist.

Je nach Art des Satelliten kann dieser die verschiedensten Sensoren besitzen, z.B.:

- abbildendes Radar zur Kartierung der Erdoberfläche und der Ozeane
- Alimeter zur Bestimmung der Geländehöhe
- optische Sensoren
- Ozonsensoren
- Sensoren zur Messung der Temperatur

Die von den Sensoren stammenden Telemetry-Daten werden gesammelt und Blockweise zur Bodenstation übertragen.

Die verwendete Modulation ist ein Phase Shift Keying (PSK) oder auch Phasenumtastung genannt. Bei dieser Art der Modulation werden die Informationen durch eine Phasenverschiebung moduliert. Je nach Anzahl der unterschiedlichen Phasenwinkel und daraus resultierenden Bits, die übertragen werden, unterscheidet man die folgenden Signale:

- BPSK – Binäry PSK (1 Bit)
- QPSK – Quadrature PSK (2 Bit)
- 8PSK – (3 Bit)
- DPSK – Differential PSK

Beim Binäry PSK oder auch 2PSK genannt, wird die Phasenlage bei der Modulierung zwischen den beiden binären Werten „0“ und „1“ geändert. Das entspricht einer Änderung der Phasenlage des Trägersignals zwischen zwei Phasen, die jeweils um 180° zueinander

verschoben sind. Beim Quadrature PSK oder auch 4PSK genannt, wird zwischen vier unterschiedlichen Phasen unterschieden, die jeweils um 90° verschoben sind. Mit diesen vier Zuständen können 2 Bits übertragen werden. Beim 8PSK sind das jeweils acht Zustände und den daraus resultierenden 3 Bits. Beim Differential PSK wird der binäre Wert „0“ durch eine gleich bleibende Phase und der binäre Wert „1“ durch eine wechselnde Phase moduliert.

Die Aufgabe des Demodulators besteht darin, das phasenmodulierte Signal in ein digitales NRZ-Signal (Non-Return-to-Zero) umzuformen. Bei diesem Signal wird zwischen zwei Pegeln unterschieden. Dieses gibt es als NRZ-L Signal und NRZ-M Signal. Bei der NRZ-L (Level) Codierung repräsentiert ein Pegel die binäre „1“ und der andere Pegel die binäre „0“. Bei der NRZ-M (Mark) Codierung wird eine binäre „1“ durch einen wechselnden Pegel und eine binäre „0“ durch einen gleich bleibenden Pegel repräsentiert.

Bei der seriellen Datenübertragung durch einen Satelliten wird kein Taktsignal mit gesendet. Daher muss am Boden eine Taktrückgewinnung mittels einer Phase-Locked-Loop (PLL) erfolgen, die im Bitsynchronisator enthalten ist. Die PLL ist ein phasengekoppelter Regelkreis, der auf die Referenzfrequenz, d.h. auf die Übertragungsrate, eingestellt wird. Anhand der Flanken des NRZ-Signals regelt sich der Bitsynchronisator automatisch nach, um den idealen Abtastzeitpunkt zu gewährleisten. Daher ist es von Wichtigkeit, dass das Signal viele Flanken enthält.

Zuletzt werden die Daten auf der Festplatte für die spätere Weiterverarbeitung abgespeichert. Je nach Dauer des Überfluges und der Übertragungsgeschwindigkeit können die Dateien um die 20 GByte groß werden.

Auf dem gesamten Übertragungsweg kann es zu Störungen kommen, die sich in der Qualität der Telemetry-Daten widerspiegeln. Dabei ist das Signal-Rausch-Verhältnis entscheidend. Wird dieses Verhältnis zu klein, kann nicht mehr eindeutig zwischen Signal und Rauschen unterschieden werden. Die auf der gesamten Übertragungsstrecke auftretenden Einflüsse wie z.B. Dämpfung des Signals oder thermisches Rauschen als auch Parameter wie z.B. Sendeleistung oder Entfernung des Satelliten werden im „Link Budget“ berücksichtigt [1] S.535. Das „Link Budget“ ist eine Worst-Case Überschlagsrechnung, die auch eine bestimmte Bitfehlerrate (BER), z.B. 10^{-6}, mit der die Daten bei niedrigen Elevationen empfangen werden sollen, berücksichtigt. Bei einer niedrigen Elevation der Empfangsantenne ist der Satellit am weitesten entfernt und bei einem Winkel von 90° am dichtesten. Zusätzlich wird für den Empfang freie optische Sicht benötigt, was bei kleinen Elevationen durch Bäume oder Gebäuden erschwert wird.

Letztendlich wird zwischen drei Arten von Fehlern unterschieden:

- Bitfehler – kleines Signal-Rausch Verhältnis
- Burst-Fehler – Signal-Rausch Verhältnis so klein, dass zeitweilig nicht mehr zwischen Signal und Rauschen unterschieden werden kann
- Bitschlupf (Bitslip) – Abtastung durch den Bitsynchronisator gerät aus dem Takt

Bei einem Bitfehler handelt es sich um einen einzelnen Bitfehler, der eher zufällig verteilt auftritt. Bei einem Burst-Fehler hingegen tritt eine blockweise Störung des Signals auf, die dazu führt, dass eine ganze Folge von Bits falsch („gekippt“) ist.

Ein Problem für die Rahmensynchronisation stellt der Bitschlupf bzw. Bitslip dar. Die Abbildung 1 zeigt den negativen Bitschlupf, in dessen Folge Bits ausgelassen werden und somit im empfangenen Datenstrom fehlen. Dies ist der Fall, wenn die Abtastung des Signals durch den Bitsynchronisator schneller erfolgt, als die Daten gesendet werden. Im schlimmsten Fall wird, wie in der Abbildung 1 dargestellt, ein Bit übersprungen.

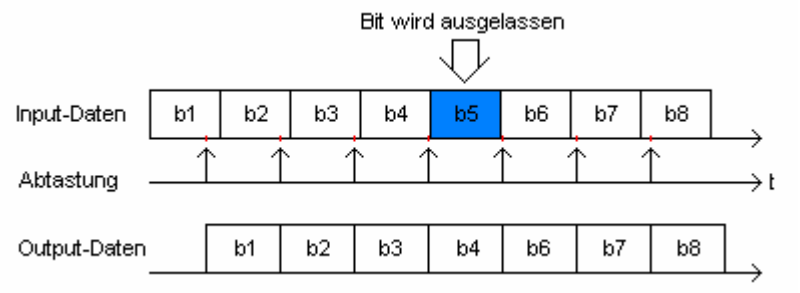

Abbildung 1: Negativer Bitschlupf

Zusätzlich können im empfangenen Datenstrom nicht nur einzelne Bits fehlen, sondern es können auch Bits mehrfach vorhanden sein. Die Abbildung 2 stellt den so genannten positiven Bitschlupf dar. Diese Art des Bitschlupfs tritt auf, wenn die Abtastperiode des Bitsynchronisators kleiner ist als die Bitperiode des empfangenen seriellen Signals. Ist nun zum Abtastzeitpunkt das abzutastende Bit breiter als die Abtastperiode, wird das entsprechende Bit doppelt abgetastet.

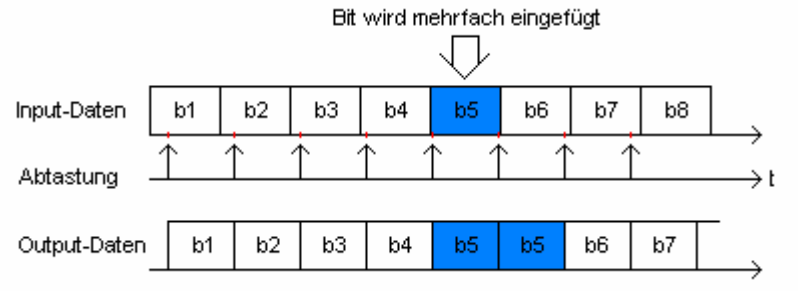

Abbildung 2: Positiver Bitschlupf

2.2 Der CCSDS-Standard

CCSDS ist das Consultative Commitee for Space Data Systems mit dem Sitz in Washington, welches 1982 gegründet wurde. Es ist eine internationale Organisation der führenden Weltraumorganisationen. Die Aufgabe besteht darin gemeinsame Methoden des Datenverkehrs auszuarbeiten und Standards festzulegen. Dies ermöglicht die gemeinsame Nutzung von Infrastrukturen der einzelnen Organisationen. Der für diese Diplomarbeit benötigte Standard ist der des „Telemetry-Channel-Coding“[5]. Er wird in diesem Kapitel näher erläutert.

In Anlehnung an moderne Datenübertragungssysteme werden 8 Bits zu einem Wort zusammengefasst. Um die empfangenen Bits in einem vorwärtsgerichtetem N-Bit langem Datenfeld eindeutig zuordnen zu können, gibt es eine weitere Festlegung, die mithilfe der Abbildung 3 näher erklärt werden soll. Anders als es sonst in der Digitaltechnik üblich ist, ist das erste empfangene Bit das „Bit 0“, das zweite empfangene Bit das „Bit 1“, bis hin zum

letzten empfangen Bit das „Bit N-1“. Um einen binären Wert darzustellen, ist das MSB immer links im Feld und das LSB rechts.

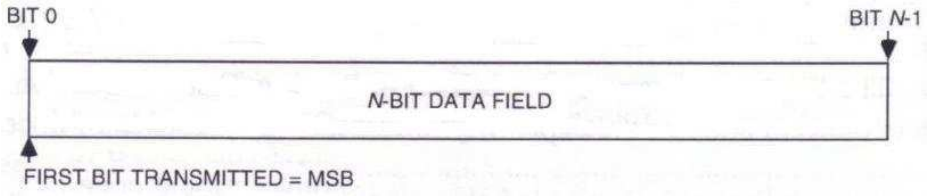

Abbildung 3: Bitnummerierung Quelle: [5] 1-2

Der Aufbau eines Rahmens ist in Abbildung 4 dargestellt. Er besteht aus einem Synchronwort, dem Attached Sync Marker (ASM), dem Informationsteil und einem Reed-Solomon Codeblock.

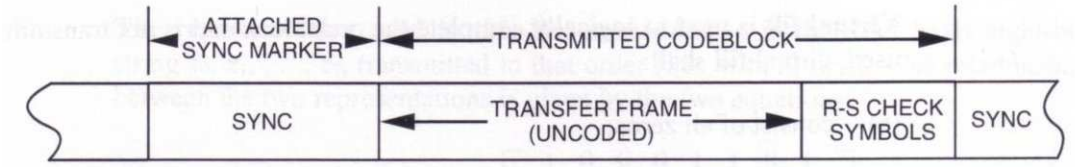

Abbildung 4: Aufbau eines Rahmens Quelle: [5] 3-5

Der Attached Sync Marker

Das Synchronwort dient dazu eine Rahmen- und Symbolsynchronisation durchzuführen. Diese ist notwendig für die Reed-Solomon Decodierung bzw. das Derandomizing und um überhaupt eine Prozessierung der Daten durchzuführen. In einem Datenstrom existieren feste Rahmenlängen, die mit einem ASM beginnen, der nicht codiert ist. Nach jedem Rahmen folgt ein weiterer Rahmen mit ASM ohne Lücke dazwischen, um so die Synchronisation aufrecht zu erhalten. Das verwendete Bitmuster ist in Abbildung 5 dargestellt.

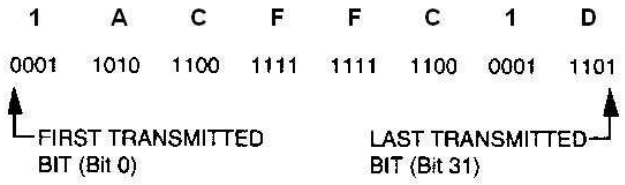

Abbildung 5: Synchronwortbitmuster Quelle: [5] 5-2

Der Pseudo-Randomizer

Um die Bitsynchronisation zu unterstützen, ist es von Vorteil, wenn das digitale Signal viele Flankenwechsel aufweißt. Daher wird jeder Rahmen mit einer statischen Zufallsbitfolge (Pseudo-Random Sequenz) Exclusiv-Oder verknüpft, um auch bei Idle-Rahmen (Rahmen ohne Inhalt) Flankenwechsel zu garantieren. Die Zufallsbitfolge entspricht dem folgenden Polynom:

$$h(x) = x^8 + x^7 + x^5 + x^3 + 1 \qquad \text{Gl. 2.1}$$

Die durch das Polynom erzeugten ersten 40 Bits entsprechen der Bitfolge:

1111 1111 0100 1000 0000 1110 1100 0000 1001 1010 ….

Es ist zu beachten, dass der Generator am Anfang mit „all ones“ initialisiert wird, d.h. jedes Register hat am Anfang den Zustand „1“[5].

Eigenschaften der Reed-Solomon Codierung und Interleaving

Der verwendete Reed-Solomon Code weist sehr gute Eigenschaften im Bereich der Korrektur von Burst-Fehlern auf. Laut Standard gibt es zwei mögliche Varianten der Reed-Solomon Codierung. Einmal die Variante mit der Korrektur von 16 Symbolfehlern und die Variante mit der Korrektur von 8 Symbolfehlern. Die beiden Varianten dürfen nicht gemischt werden.
Des Weiteren besitzt der Reed-Solomon Code folgende Parameter:

- Die Symbollänge J beträgt 8 Bits.
- Die maximale Anzahl an Symbolfehlern, die korrigiert werden können, beträgt E = 16 oder E = 8.
- Die Codewortlänge beträgt $n = 2^J - 1 = 255$ Symbole.
- Die Anzahl an Parity-Check Symbolen im Codewort beträgt 2*E, d.h. 32 Symbole für E = 16 oder 16 Symbole für E = 8.
- Es ergeben sich demnach k = n – 2*E Symbole, die Informationen enthalten.
- Das Polynom zur Erzeugung des Galois-Feldes (GF) lautet:

$$F(x) = x^8 + x^7 + x^2 + x + 1 \qquad \text{Gl. 2.2}$$

  irreduzibles Polynom über GF(2).

- Das entsprechende Code-Generator Polynom lautet:

$$g(x) = \prod_{j=128-E}^{127+E} (x - \alpha^{11j})$$

  über $GF(2^8)$ mit $F(\alpha) = 0$.

- Der verwendete Code ist ein vorwärtsgerichteter.

Codierverfahren arbeiten leistungsfähiger, je homogener die Fehler verteilt sind. Es ist nämlich schwieriger eine große Anzahl an Fehlern, die unmittelbar aufeinander folgend auftreten, zu korrigieren. Daher werden mehrere Reed-Solomon Codewörter zu einem Rahmen zusammengefasst und gleichzeitig Byteweise verschachtelt. Die Anzahl an Codewörtern, die in einem Rahmen enthalten ist, wird durch den Interleaving-Faktor (I) angegeben. Dieser kann die Werte I = 1, 2, 3, 4, 5 und 8 annehmen. Dadurch ergibt sich eine Rahmenlänge ohne ASM von:

$$L_{max} = n * I = (2^J - 1) * I = 255 * I$$

I	1	2	3	4	5	8
$\mathbf{L_{max}}$	255	510	765	1020	1275	2040

Tabelle 1: Rahmenlänge im Verhältnis zum Interleaving-Faktor

Die Tabelle 1 zeigt die entsprechende Rahmenlänge resultierend aus dem Interleaving-Faktor, während die Abbildung 6 das Interleaving verdeutlichen soll.

$$d_1^1 \ldots d_1^5 \; d_2^1 \ldots d_2^5 \ldots d_k^1 \ldots d_k^5 \; p_1^1 \ldots p_1^5 \ldots p_{2E}^1 \ldots p_{2E}^5$$

d: Datenbyte k: Informationsblocklänge p: Paritätssymbole

Abbildung 6: Interleaving mit einem Interleaving-Faktor = 5

Die Abbildung 6 zeigt einen Rahmen, der fünf Reed-Solomon Codewörter enthält. Die oberste Ziffer repräsentiert den jeweiligen RS Block und die unterste das entsprechende Byte, sei es vom Informationsblock d oder den Paritätsstellen p. Zuerst werden alle ersten Bytes der Blöcke zusammengefügt. Danach die zweiten bis hin zu den k-ten Bytes des Informationsteils jedes Blockes. Die Paritätssymbole werden ebenfalls in der Reihenfolge erstes Symbol des ersten Blockes bis hin zum ersten Symbol des fünftes Blockes angefügt, bis alle 2*E Symbole der Blöcke angefügt wurden.

Dual-Basis Darstellung der Reed-Solomon Codierung

Jedes 8 Bit breite Reed-Solomon Symbol ist ein Element aus dem endlichen Feld GF(256). Ebenfalls kann angenommen werden, dass das Galois-Feld(256) ein Feld der Größe acht über das binäre Feld GF(2) ist. Die aktuelle 8 Bit Darstellung eines Symbols entspricht einer Funktion der gewählten Basis. Eine Basis für GF(256) über GF(2) ist das Paar $(1, \alpha^1, \alpha^2, \ldots, \alpha^7)$. Das heißt, dass jedes Element aus GF(256) der folgenden Darstellungsform entspricht:

$$u_7\alpha^7 + u_6\alpha^6 + \ldots + u_1\alpha^1 + u_0\alpha^0,$$

wobei u_i eine eins oder eine null ist.

Eine andere Basis über GF(2) bildet das Paar $(1, \beta^1, \beta^2, \ldots, \beta^7)$, bei dem $\beta = \alpha^{11}$ entspricht bzw. mit dem Polynom aus Gl 2.3 erzeugt werden kann[2]S.14.

$$F(x) = x^8 + x^6 + x^4 + x^3 + x^2 + x + 1 \qquad \text{Gl. 2.3}$$

Zu dieser Basis existiert eine so genante „Dual-Basis“ $(l_0, l_1, \ldots, l_7)$, in die die Elemente des Reed-Solomon Codeblocks transformiert sind. Diese „Dual-Basis“ wird auch als Berlekamp Darstellung bezeichnet. Eine Transformation von der konventionellen Darstellung in die Dual-Basis erfolgt nach der folgenden Formel:

$$z = z_0l_0 + z_1l_1 + \ldots + z_6l_6 + z_7l_7$$

bzw.

$$[z_0, z_1, \ldots, z_6, z_7] = [u_7, u_6, \ldots, u_1, u_0]\ T_{\beta l}$$

und der Matrix

$$T_{\beta l} = \begin{bmatrix} 0&0&1&1&0&1&1&1 \\ 0&1&0&1&1&1&1&1 \\ 1&0&0&0&0&1&1&1 \\ 0&0&0&0&1&0&0&1 \\ 0&0&1&1&1&1&1&1 \\ 0&0&1&0&1&0&1&1 \\ 0&1&1&1&1&0&0&1 \\ 0&1&1&1&1&0&1&1 \end{bmatrix}$$

und die Rücktransformation entsprechend:

$$[z_0, z_1, \ldots, z_6, z_7]\ T_{\beta l}^{-1} = [u_7, u_6, \ldots, u_1, u_0]$$

mit der inversen Matrix

$$T_{\beta l}^{-1} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Kurz formuliert wird ein Element β mit der Matrix $T_{\beta l}$ multipliziert. Die Reihe 1, Reihe 2, …, und Reihe 8 der Matrix $T_{\beta l}$ entsprechen den in die Dual-Basis transformierten Werten von β^7, β^6,…, und β^0.

2.3 Theorie der Rahmensynchronisation

Die Begriffe Rahmensynchronisation bzw. Blocksynchronisation und Bytesynchronisation haben eine ähnliche Bedeutung. Die Bytesynchronisation beschreibt das Auffinden der Synchronworte in einem seriellen Bitstrom. Anhand dieser Bitmuster kann die genaue Bytelage festgestellt werden. Die Rahmensynchronisation beschreibt das wiederholte Auffinden dieser Synchronwörter im Blockabstand. Wird von Rahmensynchronisation gesprochen, schließt das in der Regel den Begriff der Bytesynchronisation ein. Alle Informationen dieses Kapitels wurden den Quellen [3], [4] entnommen.

In der digitalen Informationstechnik werden die Daten parallel gespeichert, wobei acht Bits ein Byte bilden. Nach der seriellen Übertragung muss eine eindeutige Interpretation der Daten gewährleistet sein. Daher müssen die acht Bits, die auf der Sendeseite ein Byte bilden, auch auf der Empfängerseite entsprechend als Byte erkannt und abgespeichert werden. Dazu bedarf es einer Bytesynchronisation, bei der das entsprechende Synchronwort des ersten Rahmens gesucht wird. Dieser Zustand wird als „Search“ bezeichnet. Hat man das Synchronwort gefunden, kann eine eindeutige Zuordnung erfolgen. Da wie bereits beschrieben Bitschlupf auftreten kann, muss die Bytesynchronisation regelmäßig überprüft werden. In der Abbildung 7 ist ein byteunsynchronisierter Zustand dargestellt. Die seriellen Daten werden im Schieberegister zu jeweils acht Bits zusammengefasst und abgespeichert. Die Farben im Bild sollen hervorheben, welche Bits auf der Sendeseite zusammengefasst werden. Um diese Daten richtig lesen zu können, müssten alle Bits um drei Bits nach links verschoben werden.

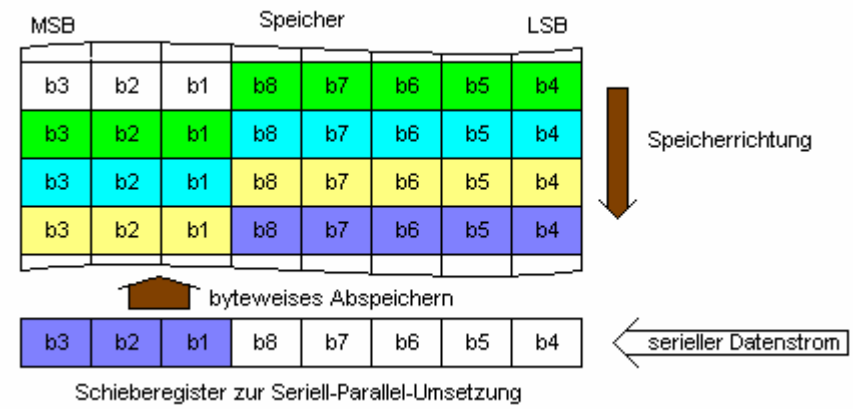

Abbildung 7: Beispiel für byteunsynchronisiertes Abspeichern von seriellen Daten

Des Weiteren muss eine Rahmensynchronisation erfolgen, um zu bestimmen, ob die Informationen eines Rahmens noch bytesynchronisiert sind. Bei einer Rahmensynchronisation wird das jeweils nächste Synchronwort am Ende eines Rahmens überprüft. Dabei bildet das Synchronwort das Ende des vorhergehenden und den Beginn des folgenden Rahmens. Die Abbildung 8 verdeutlicht diesen Zusammenhang nochmals. Gleichzeitig ist der rahmensynchronisierte Zustand dargestellt, der nur dann vorliegt, wenn das Synchronwort mehrfach jeweils im Rahmenabstand erkannt wird. Dieser Zustand wird auch als „Locked“ bezeichnet.

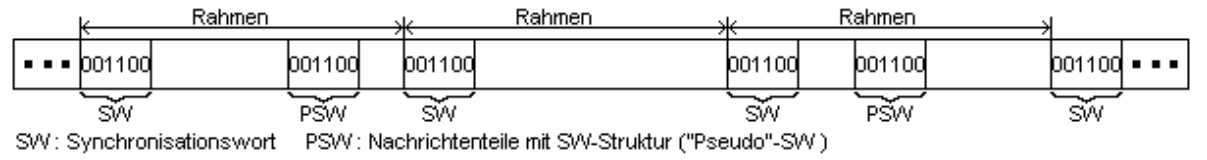

Abbildung 8: Beispiel eines rahmensynchronisierten Zustandes

Es können auch Bitmuster im Datenteil auftreten, die dem Synchronwort ähnlich sind. Das spielt aber im synchronisierten Zustand keine Rolle, da an den jeweiligen Stellen nicht auf das Vorhandensein eines Synchronwortes geprüft wird.

Probleme bei der Rahmensynchronisation

Besitzt das Synchronwort einen oder mehrere Bitfehler, so wird dies durch die Überprüfung festgestellt. Denn selbst wenn das Synchronwort aufgrund von Bitfehlern nicht erkannt wird, wird von dessen Vorhandensein ausgegangen, wenn das nächste Synchronwort an der richtigen Stelle ist. Dadurch besteht die Möglichkeit, zusätzlich eine Qualitätsüberwachung mit einzubauen. Somit kann eine Bitfehlerrate über alle Synchronworte bestimmt werden.

Anhand der Abbildung 9 soll verdeutlicht werden, was beim Auftreten von Bitschlupf geschieht. Durch die Verschiebung wird das Synchronwort immer an der falschen Stelle gesucht. Das hat zur Folge, dass alle folgenden Synchronwörter als falsch erkannt werden, da das Bitmuster nicht mit dem Bitmuster des Synchronwortes übereinstimmt.

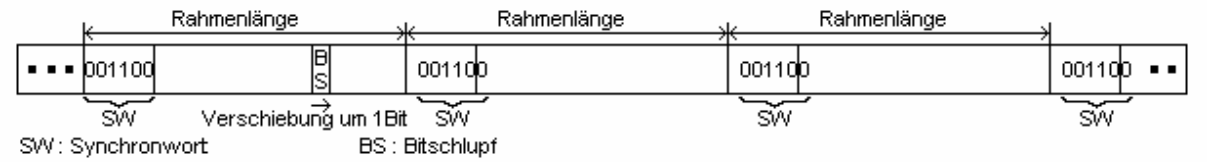

Abbildung 9: Verlust der Synchronisation durch Bitschlupf

Ebenfalls kann es vorkommen, dass mehrere Synchronwörter in Folge Bitfehler enthalten. Daher ist es angebracht, erst ab einer gewissen Anzahl von falsch erkannten Synchronwörtern eine erneute Bytesynchronisation durchzuführen, da eine Bytesynchronisation immer mit einem erhöhten Aufwand verbunden ist. Aus diesem Grund wird nach jedem falsch erkannten Bitmuster, dieses zusätzlich noch innerhalb eines Bitschlupffensters gesucht. Dabei werden alle Bitmuster innerhalb des Fensters untersucht. Wird auf diese Weise das Vorhandensein von Bitschlupf festgestellt, muss keine Resynchronisation durchgeführt werden. Nicht erkannte Synchronwörter und damit Rahmen werden auch als „Flywheel-Frames“ bezeichnet.

Um die Synchronisation wiederherzustellen, wird ab dem letzten nicht erkannten Synchronwort eine Bytesynchronisation durchgeführt. Die Abbildung 10 zeigt den Fall einer erfolglosen Resynchronisation. Da es sich bei dem gefundenen Synchronwort um ein Pseudo-

Synchronwort handeln kann, wird erst von einem synchronisierten Zustand ausgegangen, wenn die Prüfung auf das nächste und übernächste Synchronwort positiv ausfällt.

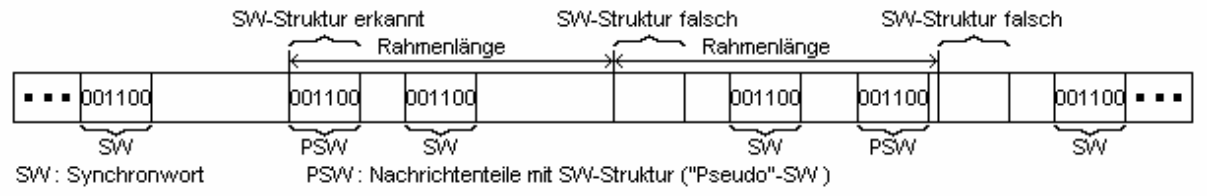

Abbildung 10: Erfolgloser Resynchronisationsversuch

Das liegt daran, dass es sehr unwahrscheinlich ist, dass an der gleichen Stelle im nächsten Rahmen als Nutzsignal wieder eine Synchronwort-Bitfolge gefunden wird. Selbst wenn dieser Fall auftreten sollte, wird er nicht dauerhaft sein, sodass selbst bei einer scheinbar erfolgreichen Synchronisation auf das Pseudo-Synchronwort nach kurzer Zeit ein erneuter Synchronausfall und Synchronsuchlauf erfolgt. Wird ein echtes Synchronwort gefunden, ist die Prüfung auf das nächste Synchronwort positiv, wenn keine Bitfehler vorliegen.

2.4 Reed-Solomon Codierung

In diesem Kapitel werden nur die systematischen Reed-Solomon Codes behandelt. Bei systematischen Codes bleibt der Informationsteil transparent und muss daher nicht dekodiert werden, um diesen lesen zu können. Die Codierung erfolgt in diesem Fall, indem redundante Symbole an die Nachricht angefügt werden. Weiterhin gehören sie zu der Gruppe der zyklischen Blockcodes (zyklische Verschiebung eines Codewortes ergibt wiederum ein Codewort). Auf Beweise der einzelnen Behauptungen wird in diesem Kapitel verzichtet. Diesbezügliche Informationen können der Quelle [6] entnommen werden.

Um mit der Reed-Solomon Codierung zu arbeiten, muss der Begriff der endlichen Körper bzw. Galois-Felder(GF) erklärt werden. Zitat gemäß Quelle [6] Seite 27: „Sei n Element N. Es gibt genau dann einen Körper mit n Elementen, wenn $n = p^J$, wobei p eine Primzahl ist und J Element N. Ist $n = p^J$ so gibt es bis auf Isomorphie genau einen Körper mit p^J Elementen“. Auf die Elemente des Körpers gelten die Regeln der Addition und der Multiplikation. Ebenfalls gilt das Distributivgesetz:

$$c * (a + b) = c * a + c * b$$

In der Codierungstheorie arbeitet man immer mit solchen endlichen Körpern bzw. Alphabeten. Endlich heißt, dass der Körper endlich viele Elemente besitzt. Ebenfalls sollten arithmetische Operationen auf diesen Körper definiert sein, was wie oben erwähnt der Fall ist. Isomorphie heißt, dass die Elemente des Körpers anders benannt werden können, jedoch die Multiplikation und Addition nach Anpassung an die neuen Elemente gleich bleibt. Für die Erzeugung eines solchen Körpers wird ein irreduzibles Polynom über den endlichen Körper GF(p) mit dem Grad k benötigt. Die Feldelemente werden als α^x bezeichnet mit $x = 0$ für das erste Element und $x = 2^J-1$ für das letzte Element im Feld. Da Rechneroperationen auf einem Computer üblicherweise im Dualsystem ausgeführt werden, basieren die meisten Codes auf der Primzahl $p = 2$ und somit auf $n = 2^J$ Elementen. Im Verlauf dieses Kapitels wird nur noch diese Sorte von Codes behandelt. Die Minimaldistanz ergibt sich aus der Anzahl zu korrigierender Fehler t: $n-k = d-1 = 2t$, $d = d_{min}$.

Ein t-Fehler korrigierender RS-Code hat die folgenden Parameter:

Blocklänge:	$n = 2^J - 1$
Anzahl an Paritätsprüfstellen:	$n - k = 2t$
Minimaldistanz:	$d_{min} = 2t + 1$

Wird der Parameter J festgelegt, ergibt sich eine feste Blocklänge für die Codierung. Das Generatorpolynom eines t-Fehler korrigierenden RS-Codes berechnet sich wie folgt:

$$g(x) = \prod_{i=0}^{2t-1} (x - \alpha^{i+\Delta}), \text{ mit } 1 \leq \Delta \leq 2^J - 2t$$

Der Parameter Δ ist der Basisexponent. Dieser kann als eine Art Offset aufgefasst werden, ab dem das Polynom im $GF(2^J)$ gebildet wird. Die Wurzeln des Polynoms g(x) sind demnach durch die Feldelemente α^{Δ} bis $\alpha^{\Delta+2t-1}$ gegeben und werden auch als Stützstellen bezeichnet. Der Informationsteil der Länge k kann in Polynomschreibweise folgendermaßen dargestellt werden:

$$i(x) = a_0 + a_1*x + a_2*x^2 + \ldots + a_{k-2}*x^{k-2} + a_{k-1}*x^{k-1}$$

Die Symbole a_x werden als Elemente des $GF(2^J)$ aufgefasst. Dieses Polynom wird durch die Codierung um 2t Prüfstellen erweitert, sodass ein Codeblock der Länge n entsteht:

$$c(x) = a_0 + a_1*x + a_2*x^2 + \ldots + a_{k-2}*x^{k-2} + a_{n-1}*x^{n-1}$$

Dies erfolgt mithilfe des Generatorpolynoms g(x), sodass dessen Stützstellen gleichzeitig auch Wurzeln des nun erweiterten Polynoms sind. Die 2t Paritätssymbole ergeben sich folgendermaßen:

$$p(x) = i(x) * x^{n-k} \bmod g(x)$$

Um eine Überprüfung des RS-Codeblocks auf Fehler hin durchzuführen, reicht es aus die 2t Syndrome zu berechnen. Das Syndrom ist vergleichbar mit dem Paritätscheck bei der RS-232 Schnittstelle. Zur Berechnung aller Syndrome, werden alle 2t Stützstellen des Generatorpolynoms nacheinander in das Codeblockpolynom eingesetzt. Ist eines der Syndrome ungleich Null, ist ein Fehler im RS-Codeblock aufgetreten[7]. Ist dies der Fall, kann eine Fehlerkorrektur durchgeführt werden. Dafür müssen zuerst die Positionen der fehlerhaften Symbole und danach die richtigen Symbole an diesen Stellen ermittelt werden. Dies erweist sich als relativ aufwendig, da es mit dem Lösen von Gleichungssystemen mit t Unbekannten verbunden ist.

2.5 Der digitale Datenrekorder

Der digitale Datenrekorder (DDR) der Firma „Reach Technologies Inc.“ ist in der Lage serielle Telemetry-Daten (z.B. NRZ-L Signal) auf Festplatte zu speichern und auch wiederzugeben. Dieser besitzt die folgenden Systemparameter:

- Betriebssystem Windows XP
- Pentium 4 mit 3,4 GHz
- 512 MB Arbeitsspeicher
- 2 SATA Festplatten mit 80 GB und 750 GB
- Maus und Tastatur

Für die Steuerung aller Aktivitäten, besitzt der Rekorder ein Benutzerinterface im Standard Windows Design um die Arbeit mit diesem zu erleichtern. Die geplanten Operationen können entweder manuell durchgeführt werden oder über einen „Schedule Manager“, der automatisch alle Operationen bis zu sieben Tagen in der Zukunft liegend bearbeitet.

Es gibt drei Arten von Applikationen[8]:

- Transport-Applikationen, die die Aufnahme auf das Medium steuern (z.B. Festplatte).
- Interface-Applikationen, die die Ein und Ausgabe steuern (z.B. Telemetry-Interface).
- Prozessierungs-Applikationen, welche die Telemetry-Daten manipulieren oder analysieren.

Die Aufzeichnung der Daten erfolgt in einem strukturierten Format. Die Abbildung 11 zeigt dieses Format.

Beginning of Data Set — End of Data Set

Data Set Header		Data					
Data Set Header	Aux Data File	Time Stamp	Data Record 0	Time Stamp	Data Record 1	Time Stamp	Data Record 2
512 bytes		32 bytes		32 bytes		32 bytes	
64k bytes		64k / 1M / 4M bytes		64k / 1M / 4M bytes		64k / 1M / 4M bytes	
BoDS		MoDS					EoDS

Abbildung 11: Datenformat des Rekorders Quelle: [8]

Am Anfang eines jeden Datensatzes wird ein 64 KB großer Data-Set Header angefügt, der diverse Informationen enthält. Die Telemetry-Daten werden je nach Einstellung zu Blöcken von 64 kB/ 1 MB oder 4 MB zusammengefasst, wobei die ersten 32 Bytes wiederum zusätzlich angefügte Informationen enthalten. Der Datensatz besteht aus einem Feld von 4 Byte Blöcken. Die Anordnung der Bytes innerhalb dieser Blöcke erfolgt nach der Darstellung „Little Endian“. Bei dieser Darstellung werden die höherwertigen Bytes auf der rechten Seite und die niederwertigen Bytes auf der linken Seite im Block angeordnet. (Abbildung 12).

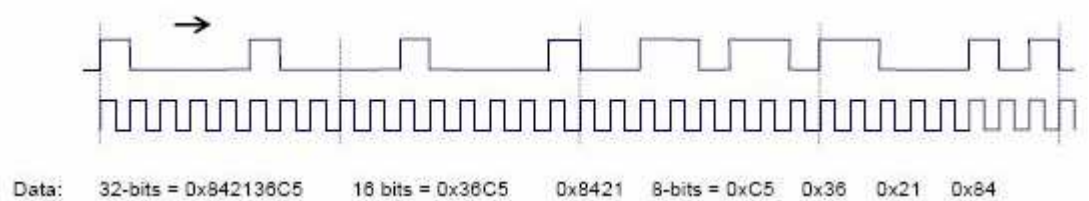

Abbildung 12: Anordnung der Bytes durch den Rekorder Quelle: [8]

3 Konzept

3.1 Wahl der Programmiersprache

In diesem Kapitel wird eine Aussage darüber getroffen, welche Programmiersprache für die Entwicklung der Software geeignet ist und welche Entwicklungsumgebung dafür zur Verfügung steht.

Ein Kriterium für die Programmiersprache ist, dass sie Bitoperationen des Prozessors unterstützt. Denn die im Rahmensynchronisator enthaltene Bytesynchronisation kann ohne Bitoperationen nicht ausgeführt werden. Unter dieser Vorraussetzung bleiben die beiden Sprachen C und C++ übrig. Damit ist die Wahl deutlich eingegrenzt worden und es muss als nächstes zwischen den beiden Programmiersprachen gewählt werden.

C++ ist eine objektorientierte Sprache die von kleinen Abweichungen abgesehen eine Obermenge von C ist. Des Weiteren hat C++ die folgenden Neuerungen gegenüber C, die für eine Verwendung sprechen:

- elegante Ein- und Ausgabe mit dem überladenen Operatoren << und >>
- Referenzen auf Variablen, Überladen von Funktionen und Operatoren
- einfache dynamische Speicherverwaltung mit `new` und `delete`
- Klassenkonzept zur Realisierung Abstrakter Datentypen (ADT)
- Vererbung von Methoden und Attributen einer Klasse

Zusätzlich hat C++ einfache Erweiterungen, die es in vielen Fällen ermöglichen, Zusammenhänge klarer und besser lesbar zu formulieren.

Alle Vorteile einbezogen, liegt hierfür die Wahl bei C++. Ein weiterer Punkt der für C++ spricht, ist das Schlüsselwort `inline`. Mit diesem Schlüsselwort wird der Quelltext der entsprechenden Methoden durch den Compiler direkt an der Stelle des Methodenaufrufes eingefügt. Dadurch wird die Datei größer, aber bei kleinen Methoden wird so ein Performancegewinn erzielt.

Für die Entwicklung der Software steht ein PC mit dem Betriebssystem Windows XP und der Entwicklungsumgebung Microsoft Visual C++ 2005 Express Edition zur Verfügung. Der PC ist von der Marke Fujitsu Siemens und ist mit einem Intel Core 2 Duo Prozessor (1,86 GHz), 2048 MB Arbeitspeicher und einer 100 GB SATA Festplatte ausgestattet.

3.2 Qualitätsüberwachung

In diesem Kapitel werden die Qualitätskriterien definiert. Infolgedessen muss geklärt werden, welche Informationen bezüglich der Qualitätsüberwachung während der Rahmensynchronisation gesammelt werden können, um die Qualität der auf der Festplatte aufgezeichneten Telemetry-Daten darzustellen. Es muss ebenfalls die Entscheidung darüber getroffen werden, inwieweit die Reed-Solomon Codierung dafür genutzt wird.

Die Qualitätsparameter werden dem Kap. 2.3 entnommen. Demnach können die folgenden Parameter während der Synchronisation gesammelt werden, um die Qualität der aufgezeichneten Telemetry-Daten darzustellen:

- aktueller Zustand (Search, Locked)
- Bitfehler der Synchronwörter
- Anzahl an Flywheel-Frames
- Bitschlupf

Der Zustand gibt Auskunft darüber, wann der Rahmensynchronisator in welchem Zustand war. Ideal wäre wenn der Rahmensynchronisator sich vom Anfang bis zum Ende der Datei im Zustand „Locked“ befinden würde. Kommt es zwischendurch zu Synchronisationsausfällen, d.h. es wird öfter zwischen den beiden Zuständen „Locked“ und „Search“ gewechselt oder die Synchronisation wird nicht gleich zu Beginn der Datei hergestellt, so ist daraus schon ersichtlich, dass die Telemetry-Datei fehlerbehaftet ist. Der nächste Parameter zählt die Anzahl der Bitfehler in den Synchronwörtern bzw. Attached Sync Markern (ASM). Das ist möglich, weil das Synchronwortbitmuster bekannt ist. Anhand der Bitfehler ist es auch möglich eine Bitfehlerrate über eine festgelegte Anzahl von Synchronwörtern zu berechnen. Dazu wird die folgende Formel verwendet:

$$BER_{ASM} = \text{Anzahl Bitfehler in ASM} / (\text{Anzahl ASM} * 32)$$

Die Gesamtanzahl an Bits ergibt sich durch die Multiplikation der gefundenen Synchronwörter mit der Bitlänge des Synchronwortes. Diese Bitfehlerrate ist aber nicht mit der herkömmlichen Bitfehlerrate zu vergleichen, da von jedem Rahmen nur ein kleiner Bereich (ASM) in die Berechnung einfließt. Jedoch kann davon ausgegangen werden, dass bei einer hohen Bitfehlerrate, z.B. 10^{-4}, Bitfehler in diesem Maße aufgrund der statistischen Gleichverteilung im Rest des Rahmens vorhanden sind. Wie bereits in Kap. 2.3 erwähnt, wird im Zustand „Locked“ das Synchronwort an den vermuteten Stellen überprüft. Ist die Abweichung des Bitmusters aufgrund von vielen Bitunterschieden zu groß, gilt das entsprechende Synchronwort und damit der ganze Rahmen als nicht erkannt (Flywheel-Frame). Mit diesem Parameter kann eine Aussage darüber getroffen werden, wie viele Synchronwörter über einem bestimmten Maß an Bitfehler enthalten. Der letzte Parameter ist die Anzahl an Bitschlüpfe, die über die Datei hinweg aufgetreten ist.

Da eine komplette Reed-Solomon Decodierung sehr aufwendig ist und dementsprechend viel Zeit und Performance beansprucht, werden nur die Rahmen auf Fehler hin überprüft. Dies ist, wie in Kap. 2.4 erwähnt, relativ einfach zu realisieren und somit kommt noch ein Qualitätsparameter hinzu, der die Anzahl an fehlerhaften Rahmen angibt.

Um eine genaue Zuordnung treffen zu können, werden diese Information gebündelt in Dateibereiche gesammelt. Dadurch wird die Verteilung der Fehler in der Datei dargestellt.

3.3 Schnittstellendefinition

3.3.1 Eingabeparameter und Parameterdatei

Die Parameterübergabe wird über die Kommandozeile in MS-Dos erfolgen. Es werden drei Parameter festgelegt: Pfadangabe zu einer Textdatei, die die Synchronisationsparameter enthält, Pfadangabe der zu synchronisierenden Datei und die Pfadangabe unter der die synchronisierte Datei abgespeichert wird. Um Probleme bei der Angabe der Dateipfade zu vermeiden, sollten diese in Anführungszeichen („“) angegeben werden.

Die Synchronisationsparameter haben die folgende Reihenfolge:

- Der erste Parameter ist das Bitmuster des Synchronwortes. Die Übergabe kann sowohl in hexadezimaler Schreibweise mit vorangestelltem „0x“ oder in dezimaler Schreibweise erfolgen. Die Länge ist auf vier Bytes festgelegt.
- Die Rahmenlänge einschließlich Synchronwort wird mit dem zweiten Parameter festgelegt. Wird dieser Parameter falsch angegeben, werden die Synchronwörter im Zustand „Search“ gefunden, jedoch wird der Zustand „Locked“ nicht erreicht, da aufgrund der falschen Rahmenlänge an der falschen Stelle geprüft wird.
- Der dritte Parameter legt die Anzahl an Bitstellen fest, die im Vergleich mit dem Bitmuster unterschiedlich sein dürfen, um noch als Synchronwortbitmuster erkannt zu werden. Die zulässigen Werte sind 0 bis einschließlich 32.
- Mit dem vierten Parameter wird die Anzahl an hintereinander erkannten Synchronwörtern festgelegt, um aus dem Zustand „Search“ in den Zustand „Locked“ zu gelangen. Werte kleiner gleich Null sind unzulässig.
- Die Anzahl an Flywheel-Frames, um aus dem Zustand „Locked“ in den Zustand „Search“ überzugehen, wird durch den fünften Parameter festgelegt. Auch hier sind nur Werte größer gleich Eins zulässig.
- Die Breite des Bitschlupffensters wird mit dem sechsten Parameter festgelegt. Der Wertebereich ist von 1 bis 8.
- Ob die gefundenen Rahmen zusätzlich durch die Reed-Solomon Codierung überprüft werden sollen, wird mit dem Parameter sieben festgelegt. Bei einer „0“ wird kein und bei einer „1“ wird ein Reed-Solomon Check durchgeführt.
- Der letzte Parameter ist der achte und legt fest, nach wie vielen gefundenen Rahmen eine Ausgabe erfolgen soll.

Die Parameter werden alle in der oben genannten Reihenfolge untereinander in die Textdatei geschrieben. Der Inhalt der Textdatei muss wie folgt aussehen:

```
AttachedSyncMarker=0x1ACFFC1D
FrameLength=1279
AllowedASMErrors=2
SearchFrames=3
FlywheelFrames=4
Bitslip=1
RS-Check=0
OutputBlock=200000
```

Die Zuordnung erfolgt durch „Parametername=Wert“ ohne Leerzeichen dazwischen. Der Parametername darf nicht falsch geschrieben werden, sonst erfolgt eine Fehlermeldung.

3.3.2 Ausgabe

Die Ausgabe erfolgt sowohl auf dem Bildschirm als auch in eine Textdatei. Das Aussehen ist jeweils gleich und wird im Verlauf dieses Kapitels am Beispiel der Ausgabe auf dem Bildschirm erläutert. Der Name und Dateipfad für die Textdatei entspricht dem der Zieldatei und wird daher von dieser abgeleitet.

Nach dem Start des Programms erfolgt eine Ausgabe der gewählten Eingabeparameter, um sich zu vergewissern, dass diese richtig eingegeben wurden. Das Aussehen entspricht der folgenden Graphik:

```
|            Parameter               |    Wert   |
|------------------------------------|-----------|
| Attached Sync Marker               |  1acffc1d |
| Rahmenlaenge                       |      1279 |
| Zulaessige Bitfehler im ASM        |         2 |
| Bakannte ASM in Folge bis Locked   |         2 |
| Unbekannte ASM in Folge bis Search |         2 |
| Bitschlupffensterbreite            |         1 |
| Ausgabeblockgroesse                |    200000 |
| Reed-Solomon-Check                 |      Ja   |
|------------------------------------|-----------|
 Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\1.RS-Test\8errors_test.chr
 Ziel  : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr
```

Die Ausgabe der in Kap. 3.2 aufgezählten Qualitätsdaten, erfolgt blockweise in einer Tabelle auf dem Bildschirm. Findet ein Zustandswechsel statt, so wird dieser sofort mit Byteposition angegeben. Wechselt der Zustand von „Locked“ nach „Search“, werden zusätzlich die gesammelten Qualitätsdaten ausgegeben, auch wenn die durch den achten Eingabeparameter festgelegte Rahmenanzahl noch nicht erreicht wurde. Würde dies nicht gemacht, erfolgt die Ausgabe der Qualitätsdaten unter Umständen erst zu einem späteren Zeitpunkt. Dadurch würden diese einen falschen Bereich auf der Telemetry-Datei widerspiegeln. Am Ende wird die Gesamtanzahl an gefundenen Rahmen, fehlerhaften Rahmen, Flywheel-Rahmen, Bitfehlern mit berechneter BER_{ASM} und Bitschlüpfen angegeben. Zum Schluss werden die benötigte Zeit und die durchschnittliche Bearbeitungsgeschwindigkeit ausgegeben:

```
|      Zustand     | Byteposition |  Rahmen  | R/S-Check | FW-Rahmen |  ASM  | BER-ASM | Bitslip |
|------------------|--------------|----------|-----------|-----------|-------|---------|---------|
|      Search      |            0 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
| Search -> Locked |       223124 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
| Locked -> Search |      1351202 |      882 |         0 |         1 |     4 | 1.4e-004|     0   |
| Search -> Locked |      1352481 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
| Locked -> Search |      6240819 |     3822 |         0 |         1 |     3 | 2.5e-005|     0   |
| Search -> Locked |      6242098 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
| Locked -> Search |     10913006 |     3652 |         0 |         1 |    10 | 8.6e-005|     0   |
| Search -> Locked |     10915564 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
| Locked -> Search |     12436295 |     1189 |         0 |         1 |    13 | 3.4e-004|     0   |
| Search -> Locked |     28991671 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
| Locked -> Search |     29004461 |       10 |         0 |         1 |     9 | 2.8e-002|     0   |
| Search -> Locked |     30425430 |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |
|      Locked      |    286225430 |   200000 |         0 |         0 |     0 | 0.0e+000|     0   |
|      Locked      |    542025430 |   200000 |         0 |         0 |     0 | 0.0e+000|     0   |
|      Locked      |    797825430 |   200000 |         0 |         0 |     0 | 0.0e+000|     0   |
|     Dateiende    |    999998239 |   158071 |         0 |         0 |     0 | 0.0e+000|     0   |
|       Gesamt     |    999998239 |   767626 |         0 |         5 |    39 | 1.6e-006|     0   |
|------------------|--------------|----------|-----------|-----------|-------|---------|---------|

Zeit in Millisekunden: 21922
Zeit in Sekunden: 21
Bearbeitungsgeschwindigkeit: 45.62 MB/s
```

3.4 Allgemeine Struktur des Rahmensynchronisators

Nachfolgend wird das theoretische Grundgerüst des Rahmensynchronisators für die spätere Implementierung erstellt. Dazu dienen die Grundlagen der Kap. 2.2, 2.3 und 2.4. Dabei wird zunächst auf eine von der Programmierung unabhängige Entwicklung geachtet, um den systematischen Entwurfsablauf zu erleichtern. Eine Vertiefung zu den einzelnen Abläufen des Rahmensynchronisators wird in den folgenden Kapiteln durchgeführt.

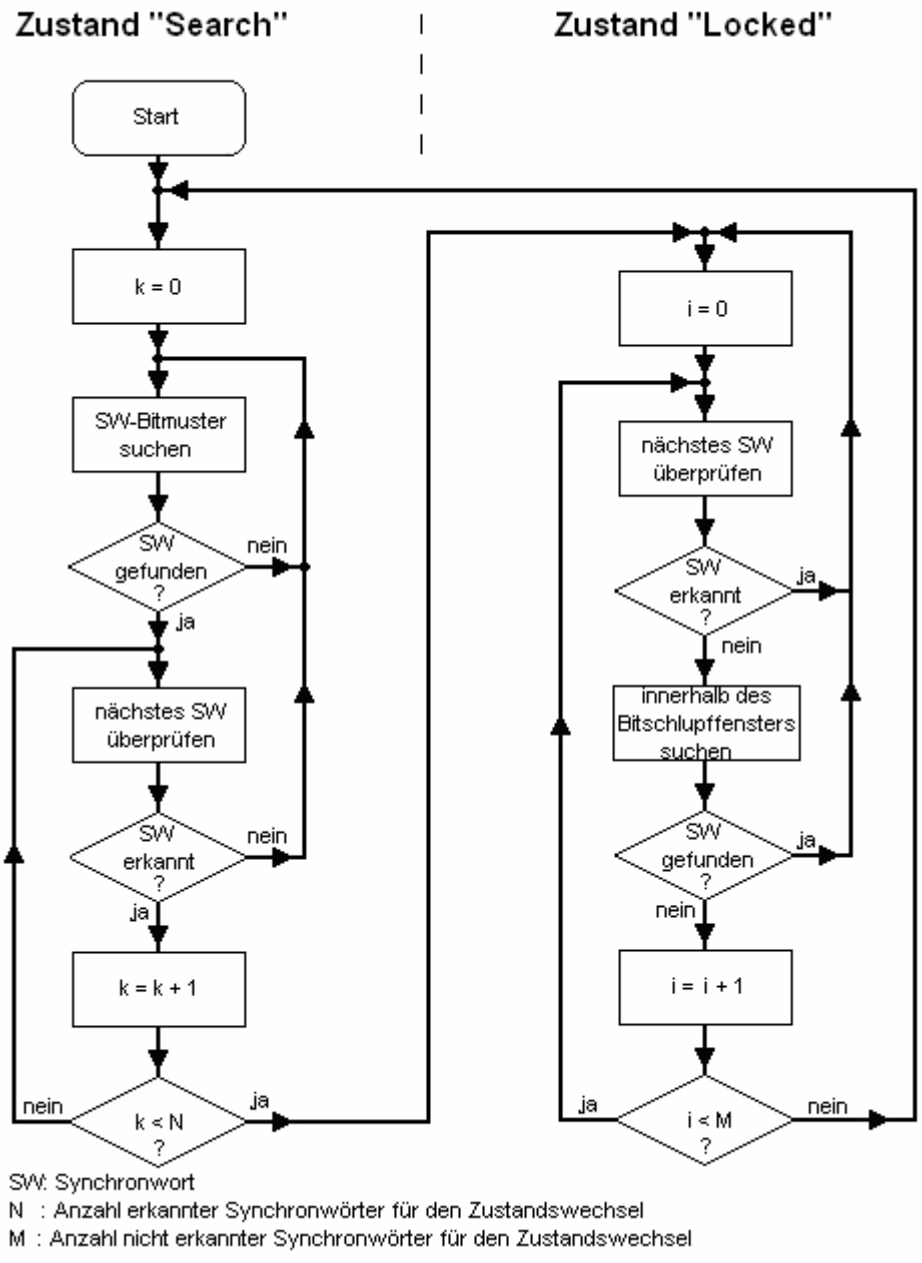

Abbildung 13: Flussdiagramm des Rahmensynchronisators

Um die Struktur zu erstellen, muss zunächst aufgezählt werden, wie der genaue Ablauf im späteren Programm sein muss. Zu Beginn erfolgt der Start des Programms und eine Ausgabe der eingegebenen Parameter. Danach befindet sich dass Programm im Zustand „Search“. In diesem Zustand wird vom Anfang der Datei beginnend, das Synchronwortbitmuster im gesamten Datenstrom gesucht. Wird ein solches Bitmuster gefunden, muss zunächst um eine Rahmenlänge im Datenstrom weiter gesprungen werden, um das nächste Synchronwort zu

untersuchen. Gilt dieses als erkannt, wird solange das nächste Synchronwort untersucht bis N Synchronwörter in Folge erfolgreich erkannt wurden. Erst jetzt wird in den Zustand „Locked“ übergegangen. Wird hingegen ein Bitmuster nicht erkannt, so wird wieder mit der Suche im gesamten Datenstrom ab dem letzten erkannten Synchronwort begonnen. Im Zustand „Locked“ wird kontinuierlich das Synchronwort im Abstand einer Rahmenlänge überprüft. Wird eines nicht erkannt, erfolgt die Suche innerhalb des Bitschlupffensters. Wird innerhalb dieses Bitschlupffensters das richtige Bitmuster gefunden, wird der kontinuierliche Ablauf fortgesetzt. Ist dies nicht der Fall, wird auch fortgesetzt, jedoch läuft ein Zähler mit, der die Anzahl an nicht erkannten Bitmustern zählt. Jedes Mal wenn ein Bitmuster an der erwarteten Stelle nicht erkannt wird, erfolgt die Suche innerhalb des Bitschlupffensters. Werden M Bitmuster infolge nicht erkannt, wird in den Zustand „Search“ zurückgesprungen. Wird hingegen vorher ein Bitmuster erkannt, wird der Zähler gelöscht und die Synchronisation bleibt erhalten. Beim Rücksprung in den Zustand „Search“ erfolgt der Ablauf wie oben beschrieben. Jedoch beginnt die Suche des Bitmusters nun ab der Position des letzten nicht erkannten Synchronwortes (Abbildung 13).

Der Abbildung 13 kann zusätzlich entnommen werden, dass ein Dateimanagement benötigt wird. Dieses muss den Datentransfer von der Festplatte zum Arbeitsspeicher und zurück organisieren und kontrollieren. Dazu gehört auch das automatische bytesynchronisierte Abspeichern der gefundenen Rahmen. Das Sammeln der Qualitätsinformationen wurde hier noch nicht weiter erwähnt. Dies erfolgt nachfolgend im Kapitel 4.5.

3.5 Bitmustererkennung und Bitmustersuche

Dieses Kapitel befasst sich mit der Synchronwortsuche im Datenstrom. Dazu muss zuerst eine geeignete Methode gefunden werden zwei Bitmuster zu vergleichen, um dieses in einem zweiten Schritt auf einem Datenstrom anzuwenden. Zuletzt wird ein Flussdiagramm erstellt, um die entwickelte Struktur implementieren zu können.

Das Referenzbitmuster stellt eine Schablone dar. Um diese Schablone mit einem zweiten Bitmuster zu vergleichen, werden all die Bitstellen zusammengezählt, an denen sich das Bitmuster von dem der Schablone unterscheidet (Abbildung 14). Je kleiner der Wert ist, desto ähnlicher sind sich die beiden Bitmuster. Der maximale Wert ergibt sich durch die Bitlänge der Schablone.

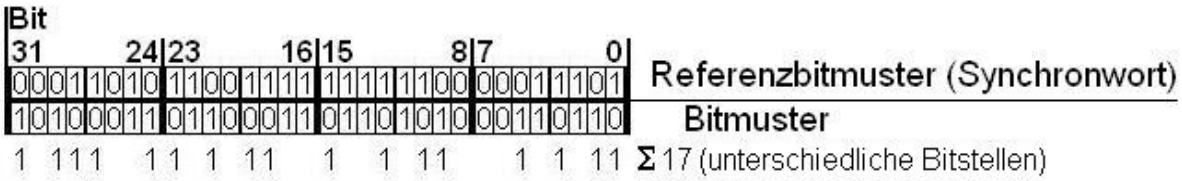

Abbildung 14: Beispiel eines Bitmustervergleichs

Um ein bestimmtes Bitmuster in einem seriellen Datenstrom zu suchen, wird die Schablone nach dem Fensterprinzip Bit für Bit über den Datenstrom geschoben und nach jedem Mal Schieben verglichen. Als nächstes wird eine Bitunterschiedsgrenze festgelegt, die angibt ob ein Bitmuster als erkannt gilt oder nicht. Die Schablone wird nun so lange über den seriellen Datenstrom geschoben, bis ein Muster als erkannt bestätigt wird oder die Datei zu Ende ist. Laut CCSDS-Standard hat das Synchronwort eine Länge von 4 Bytes und daher wird der Datenstrom in jeweils 4 Byte große Blöcke zerlegt. Die in den 4 Bytes enthaltenen 32 Bit ergeben 32 mögliche Bitmuster. Um diese zu untersuchen, werden die nächsten 4 Bytes des Datensatzes benötigt (Abbildung 15).

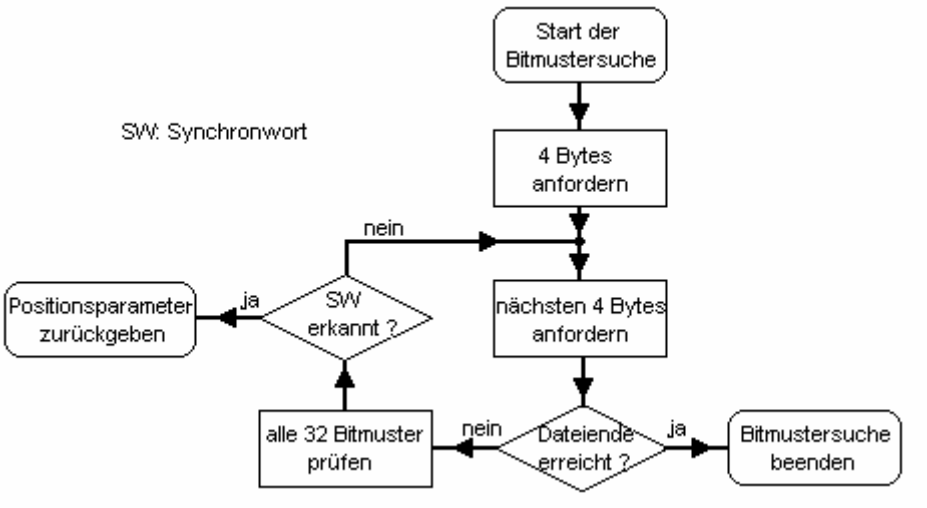

Abbildung 15: Flussdiagramm der Bitmustersuche

3.6 Das Bitschlupffenster

Das Bitschlupffenster wird aufgerufen, wenn das als Synchronwort erwartete Bitmuster abweicht. Um eine Fehlerkennung durch Bitfehler auszuschließen, wird überprüft ob das Synchronwortbitmuster bitverschoben in einem definierten Bereich vorhanden ist.

Der Ablauf orientiert sich dabei an dem in Abbildung 16 dargestellten Flussdiagramm. Zuerst wird davon ausgegangen, dass das Synchronwortbitmuster um ein Bit nach rechts verschoben ist. Das entsprechende Bitmuster wird daraufhin untersucht. Handelt es sich bei dem geprüften Bitmuster nicht um das gesuchte, wird entsprechend von einer Verschiebung von einem Bit nach links ausgegangen usw. bis entweder das richtige Bitmuster gefunden oder die definierte Fenstergrenze erreicht wird. Je nach Erfolg oder Misserfolg wird der Rahmensynchronisationsblock darüber in Kenntnis gesetzt. Bei Erfolg wird zusätzlich der Dateimanager benachrichtigt, damit dieser die Bytesynchronisation der Daten an die neue Bitverschiebung anpassen kann.

Da der Bitmustervergleich aus Kap. 3.5 insgesamt 8 Byte benötigt, sind die für die Suche von der aktuellen Bitverschiebung aus nach rechts benötigten Bits darin enthalten. Zusätzlich wurde die Fenstergröße in Kap. 3.3.1 auf maximal 8 Bit zu jeder Seite begrenzt. Für die Bitmustersuche innerhalb des Fensters nach links, muss der Dateimanager dafür sorgen, dass die entsprechenden Daten zur Verfügung stehen. Auch dann wenn gerade eingelesen wurde.

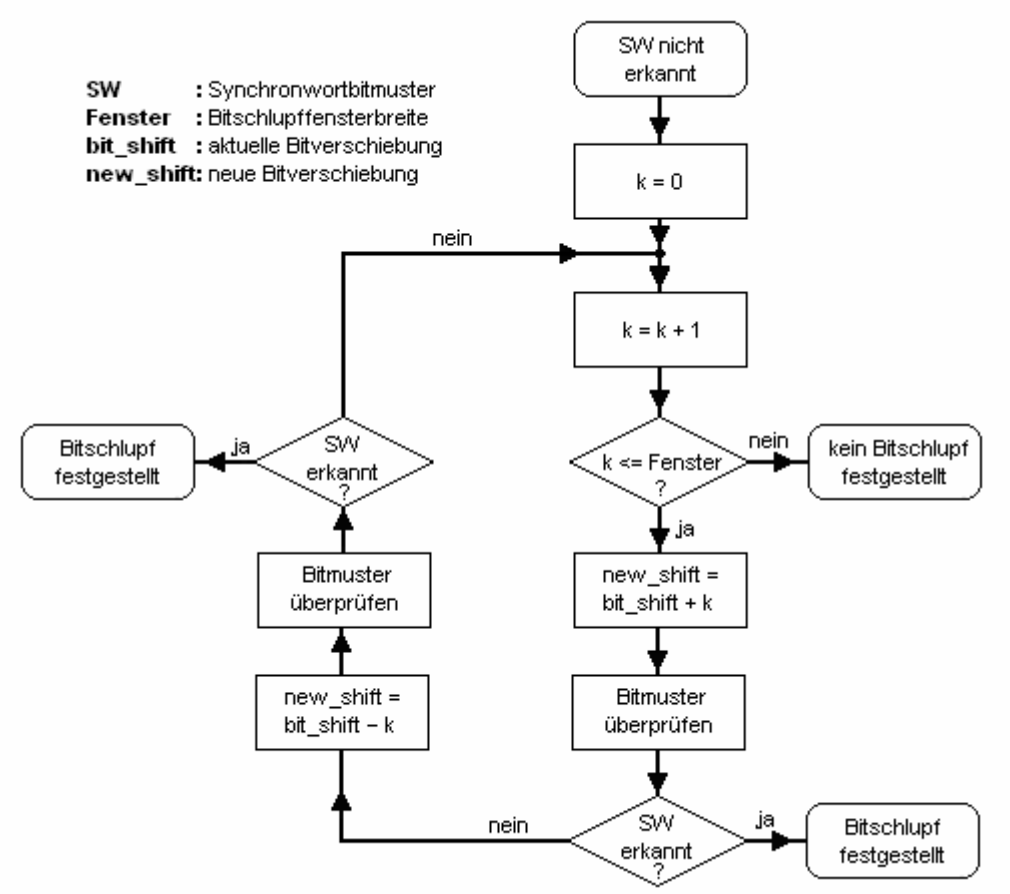

Abbildung 16: Flussdiagramm des Bitschlupffensters

3.7 Der Dateimanager

Wie bereits im Kapitel 3.4 kurz angesprochen, wird es nötig eine Dateiverwaltung zu entwickeln. Das liegt hauptsächlich daran, dass die zu bearbeitenden Dateien mehrere Gigabyte groß sind und der Arbeitspeicher nur begrenzte Kapazität besitzt.

Bevor die notwendigen Anforderungen und Problemstellungen des Dateimanagements erarbeitet werden, muss zuerst eine Wahl zwischen den folgenden beiden Möglichkeiten getroffen werden:

1. Möglichkeit:
Es wird ohne Nutzung des Arbeitsspeichers direkt mit der Festplatte gearbeitet. Die Daten werden lediglich in Variablen abgelegt. Die Vorteile hierbei sind, dass wenig Arbeitsspeicher benötigt wird und keine komplizierte Logik verwendet werden muss. Der Nachteil ist, dass sich die häufigen Festplattenzugriffe negativ auf die Performance auswirken.

2. Möglichkeit:
Es wird nur so oft wie nötig mit der Festplatte gearbeitet. Dazu sind zwei größere Blöcke im Arbeitsspeicher notwendig. Je größer diese beiden Blöcke sind, umso seltener muss auf die Festplatte zugegriffen werden. Der erste Block dient als Speicher für die eingelesenen Daten. Die Rahmen werden bytesynchronisiert aus dem ersten in den zweiten Block kopiert. Die Abspeicherung der synchronisierten Daten erfolgt erst, wenn der zweite Block vollständig gefüllt ist (Abbildung 17). Der Vorteil liegt im Performancegewinn gegenüber der ersten Möglichkeit. Probleme treten auf, wenn ein neuer Block eingelesen wurde und z.B. programmtechnisch bedingt die vorher geladenen Daten benötigt werden. Damit dies ohne Probleme abläuft, ist bei dieser Möglichkeit der Entwicklungsaufwand höher.

Abbildung 17: Modell des Dateimanagements

Ich entscheide mich für die zweite Möglichkeit. Diese ist mit einem höheren Aufwand verbunden, der aber unter dem Gesichtspunkt des Performancegewinns in Kauf genommen werden kann.

Grundanforderungen an den Dateimanager:

- Daten automatisch einlesen
- Daten automatisch bytesynchronisieren und abspeichern
- Schnittstelle zur Initialisierung und für den Zugriff auf die Daten

Nach Abbildung 13 benötigt die Rahmensynchronisation die Daten von der Festplatte, die durch den Dateimanager zur Verfügung gestellt werden. Daher müssen zuerst die Schnittstellen zwischen den beiden Blöcken näher definiert werden:
Für die Bitmustersuche aus Kap. 3.5 müssen die Daten in 4 Byte Blöcken zur Verfügung gestellt werden. Der Dateimanager muss sich die aktuelle Position im Lesespeicher merken und bei jeder Blockübergabe diesen Positionswert inkrementieren. Befindet sich der nächste

Block außerhalb des Lesespeichers, wird neu eingelesen, der Positionswert auf Null gesetzt und die ersten 4 Byte aus dem Lesespeicher übergeben. Diese Schnittstelle benötigt somit keinerlei Parameter. Mit jedem Aufruf dieser Schnittstelle wird linear durch den Datenstrom geschritten.

Zur Überprüfung des nächsten Synchronwortes, muss der Dateimanager unter Angabe der Rahmenlänge in Byte den richtigen Datenblock liefern. Beim Aufruf dieser Schnittstelle wird ab der aktuellen Position im Lesespeicher um die durch die Rahmenlänge definierte Byteanzahl gesprungen und die Bytes an dieser Stelle übergeben. Dies ist gleichzeitig die neue Position für den Dateimanager. Jeder Aufruf dieser Schnittstelle entspricht einem Rahmen, womit gleichzeitig die Möglichkeit für den Dateimanager gegeben ist, die Rahmenanzahl zu zählen. Neu eingelesen wird immer dann, wenn die neue Position über das Ende des reservierten Arbeitsspeichers hinausragen würde. Der Positionswert wird nun neu festgelegt, indem beginnend am Anfang des Speicherblocks weitergezählt wird. Gleichzeitig muss jedes Mal nach dem Einlesen auf das Dateiende hin geprüft werden.

Damit die Bytesynchronisation automatisch ablaufen kann, wird die bytegenaue Position des ersten Synchronwortes benötigt. Daher wird der Dateimanager nach dem Auffinden des Synchronwortes initialisiert. D.h. er erhält den Grad der Bitverschiebung und anhand des Zeitpunktes der Initialisierung die Position des Bitmusters. Ab dieser Position werden die Daten später synchronisiert, bevor neu eingelesen wird, die Synchronisation verloren geht oder die Rahmenanzahl ausreicht, womit nach der Bytesynchronisation der Schreibspeicher vollständig gefüllt ist und abgespeichert werden kann. Hier ist die optimale Stelle, um den Reed-Solomon Check durchzuführen. Bevor die bytesynchronisierten Rahmen abgespeichert werden, wird der Reed-Solomon Check auf diese angewandt. Vorraussetzung dafür ist, dass der Schreibspeicher ein Vielfaches der Rahmenlänge groß ist, damit zum Zeitpunkt des Checks kein halber Rahmen enthalten ist.

Die Parameter für die Schnittstelle zur Initialisierung ist die Bitverschiebung und zur Positionsberechnung die Bytelage im 4 Byte Block. Die Bytelage ist notwendig, um die genaue Byteposition des Synchronwortanfangs im Lesespeicher zu bestimmen. Zur Bitmustersuche dienen 4 Byte große Blöcke und deshalb kann der Dateimanager die Position des Synchronwortes auf 4 Byte genau bestimmen. Um zusätzlich die genaue Lage innerhalb dieses Blockes zu bestimmen, benötigt der Dateimanager die Bytelage innerhalb dieses Blockes.
Nach der Initialisierung zeigt die aktuelle Byteposition auf das Byte im Arbeitsspeicher, in dem der Anfang des Synchronwortes liegt. Durch den Rahmensynchronisationsblock wird nun das nächste Synchronwort überprüft. Der Dateimanager verschiebt die aktuelle Byteposition um eine Rahmenlänge und zeigt nun auf das Byte, in dem das Synchronwort vermutet wird und liefert die entsprechenden Bytes zur Überprüfung an den Rahmensynchronisationsblock zurück. Die aktuelle Byteposition bleibt von diesem Vorgang unberührt. Diese wird erst bei der nächsten Verschiebung um eine Rahmenlänge neu berechnet, sodass die neue Byteposition wieder auf den Anfang des Bitmusters zeigt.
Dabei ergibt sich ein Problem. Zeigt die Byteposition auf das Ende des Lesespeichers, sodass neu eingelesen wird, bleibt die Position davon unberührt. D.h. es wird immer noch die Position am Ende des Lesespeichers angegeben. Beim nächsten Sprung darf somit nicht noch einmal eingelesen werden, obwohl die neue Position, die berechnet wird, weit außerhalb des Lesespeichers liegt und wie beschrieben, dies die Bedingung zum neuen Einlesen ist. Somit muss eine zweite Bedingung zum Einlesen erfüllt sein. Kommt es zu dem besagten Umstand, wird ein Flag gesetzt. Nur wenn dieses Flag nicht gesetzt ist, darf neu eingelesen werden. Ist dieses Flag gesetzt, darf es nach Abarbeitung der Funktion gelöscht werden.

Um die Funktion des Bitschlupffensters zu gewährleisten, müssen vor jedem neuen Einlesen die letzten 4 Bytes im Lesespeicher zwischengespeichert werden. Zusätzlich werden nach jedem Sprung im Lesespeicher die 4 Bytes vor der neuen Position gespeichert. Beträgt die Bitverschiebung 0 Bit und es tritt negativer Bitschlupf auf, kann das Bitschlupffenster auf diese zwischengespeicherten 4 Bytes zurückgreifen. Wird durch das Bitschlupffenster das Vorhandensein von Bitschlupf festgestellt, wird die entsprechende Korrektur durch den Dateimanager durchgeführt.
Damit definiert sich eine weitere Schnittstelle. Die Parameter bestehen aus der neuen Bitverschiebung und einem weiteren Wert, der angibt ob sich der Anfang des Synchronwortes von der aktuellen Byteposition aus in das nächste oder das vorherige Byte verschoben hat. Zuerst erfolgt eine Bytesynchronisation mit der alten Bitverschiebung bis zur Position des verschobenen Synchronwortes. Dabei wird beim letzten Rahmen auf eine Rahmenlänge angepasst, sodass zu viele Bits abgeschnitten bzw. zu wenige Bits aufgefüllt werden. Zum Schluss wird die Bitverschiebung und wenn notwendig die Byteposition aktualisiert.

3.8 Der Reed-Solomon Check

Wie in Kap. 3.7 kurz erwähnt, erfolgt die Überprüfung der Reed-Solomon Codierung im Schreibspeicher kurz bevor die Rahmen abgespeichert werden. Der Dateimanager muss die Anzahl an Rahmen angeben. Diese Anzahl wird dann Rahmen für Rahmen überprüft und als Ergebnis wird die Anzahl an fehlerbehafteten Rahmen ermittelt. Die Daten im Lesespeicher dürfen nicht verändert werden, d.h. diese müssen für die Bearbeitung kopiert werden.

Beginnend mit dem ersten Rahmen wird die folgende Prozedur durchgeführt:

1. Derandomizing
2. Deinterleaving
3. RS-Check

Zuerst wird die Pseudo-Random Sequenz entfernt. Dazu wird diese mit dem Rahmen Exclusive-Oder verknüpft. Dabei ist zu beachten, dass das Synchronwort ausgelassen wird. Beim nächsten Rahmen wird die Bitfolge wieder beginnend nach dem Synchronwort Exclusive-Oder verknüpft. Zur Erzeugung dient das Polynom aus Gl. 2.1. Das logische Diagramm ist in Abbildung 18 dargestellt.

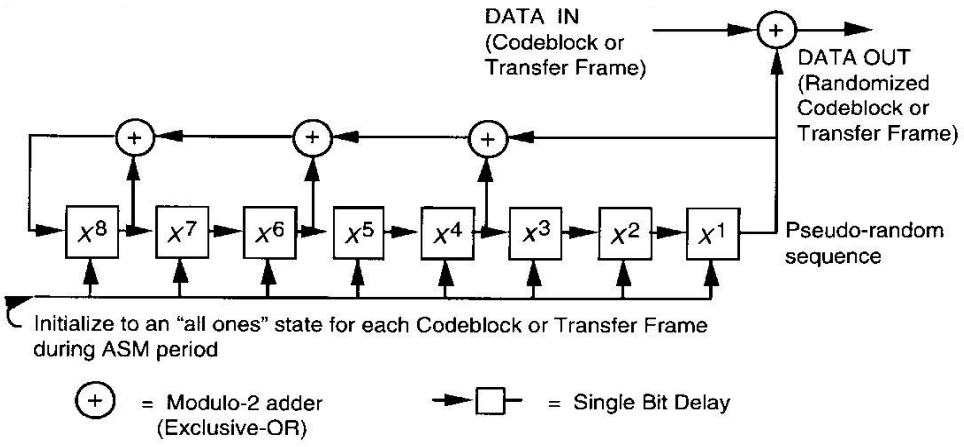

Abbildung 18: Logisches Diagramm zur Erzeugung der Pseudo-Random Bitfolge Quelle: [5] 6-3

Um die Bitfolge zu generieren, wird zu Beginn der Generator mit Einsen initialisiert. Danach wird acht Zyklen gewartet, das Bitmuster ausgelesen und abgespeichert. So wird Byte für Byte die Bitfolge erzeugt, bis diese die Länge eines Rahmens ohne Synchronwort hat. Die Pseudo-Random Bitfolge wird auf diese Weise im Arbeitsspeicher abgelegt, um diese Byte für Byte mit jedem Rahmen zu verknüpfen.

Als nächstes muss ein Deinterleaving durchgeführt werden, um den Rahmen ohne Synchronwort in 255 Byte große Blöcke zu zerlegen. Dafür werden X Blöcke der Größe 255 Byte im Arbeitsspeicher benötigt. X entspricht dem Interleaving-Faktor. Um das Deinterleaving durchzuführen, werden alle Bytes nach dem Synchronwort auf die Blöcke aufgeteilt. Das erste Byte wird am Anfang des ersten Blockes abgelegt, das Zweite in den Zweiten und nach dem letzten Block wird wieder mit dem ersten Block begonnen. Am Ende ist der Rahmen auf die X Blöcke aufgeteilt.

Zum Schluss wird jeder der X Blöcke auf Fehler untersucht. Dazu werden die Syndrome eines jeden Blockes berechnet. Sind ein oder mehrere Syndrome ungleich Null, sind in dem Block ein oder mehrere Fehler aufgetreten. Sowie ein Block fehlerhaft ist, wird der gesamte Rahmen als fehlerbehaftet angegeben. Für die Berechnung der Syndrome müssen Multiplikation und Addition auf die Elemente des Galois-Feldes angewandt werden. Die Addition entspricht einer Exclusive-Oder Verknüpfung von zwei Elementen[7]:

Beispiel:

```
  10010110
xor 00101101
  10111011
```

Bei der Multiplikation werden die Indizes der Elemente addiert[7]:

Beispiel: $\alpha^{67} * \alpha^{30} = \alpha^{67+30} = \alpha^{107}$

Das Beispielergebnis der Multiplikation ist das Element mit dem Index 107 im Galois-Feld. Ist der Index größer 254, muss der Wert zusätzlich modulo 255 gerechnet werden, um das richtige Element innerhalb des Feldes zu ermitteln. Die Umsetzung des in Gl. 2.3 angegebenen Polynoms zur Erzeugung des Galois-Feldes ist in Abbildung 19 dargestellt. Anders als bei der Pseudo-Random Bitfolge wird die Schaltung zum Anfang mit einer „1“ am Eingang initialisiert. Mit jeder Periode (Verschiebung) ergibt sich ein neues Codewort bzw. Element des Galois-Feldes. Nach 255 Perioden beginnt die Folge von vorn. Das 256te Element ist somit das erste und das 257te das zweite usw. Die Folge wäre somit scheinbar unendlich aber mit einer endlichen Anzahl an gleichen Elementen. Somit reicht es aus, die ersten 255 Elemente zu erzeugen und in einem Feld mit den Indizes 0 bis 254 abzulegen. Die Indizes verhalten sich dabei modulo 255, weil das 256 Element (Index 255) wieder das erste Element (255 modulo 255 = Index 0) ist.

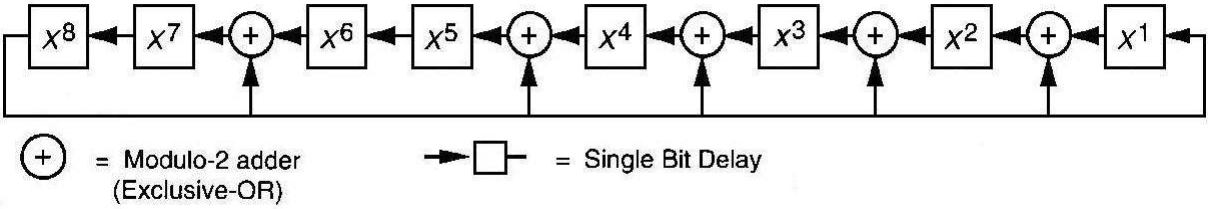

Abbildung 19: Logisches Diagramm des Polynoms zur Erzeugung des Galois-Feldes

Das erstellte Galois-Feld entspricht der konventionellen Darstellung und muss noch wie in Kap. 2.2 beschrieben in die Berlekamp Darstellung (Dual-Basis) transformiert werden. Eine andere Möglichkeit besteht darin, die Syndrome mittels des Feldes in der konventionellen Darstellung zu ermitteln. Dazu müssen jedoch die 255 Codeblockelemente eines jeden der X Blöcke transformiert werden. Die Transformation müsste hier aus der Berlekamp in die konventionelle Darstellung erfolgen. Der Nachteil bei dieser Möglichkeit besteht darin, dass mit jedem Rahmen und den darin enthaltenen Blöcken diese Transformation durchgeführt werden muss, wohingegen bei der Transformation des Galois-Feldes diese nur einmal durchgeführt wird.

4 Implementierung

4.1 Namenskonvention und weitere Regeln

Um den Quelltext besser zu verstehen und um zusätzliche Probleme zu vermeiden, wird die folgende Namenskonvention festgelegt[9]:

- Englische Namen verwenden
- Aussagekräftige Namen verwenden (ausgenommen Zählvariablen für Schleifen)
- Methoden-, Attribut- und Klassennamen bestehen aus zusammengesetzten Namen
- Ein Klassenname beginnt mit „ct_“
- Das Objekt einer Klasse beginnt mit „co_“
- Methodennamen werden klein geschrieben
- Attribute beginnen mit einem Großbuchstaben
- Zeiger werden mit einem vorangestelltem „p_“ gekennzeichnet

Der Quelltext besteht aus mindestens drei Dateien. Dem Hauptprogramm (main), einer Headerdatei mit den Klassendefinitionen und einer Datei mit den implementierten Methoden.

Des Weiteren gelten folgende Regeln:
Häufig verwendete Methoden und solche mit wenig Quellcode werden mit dem Schlüsselwort `inline` deklariert, um so die Performance zu steigern. Die Bedingung bei `inline`-Methoden ist, dass die Deklaration der Methode in derselben Datei erfolgen muss, in der auch die Definition erfolgt. Daher wird die Ausnahme getroffen, dass die `inline`-Methoden auch in der Headerdatei deklariert werden dürfen. Bei viel verwendeten Variablen werden diese mit dem Schlüsselwort `register` versehen, damit diese möglichst im Register des Prozessors gehalten werden. Dies ist aber keine Garantie, dass dies auch wirklich gemacht wird. Um dies zu unterstützen gibt es zwei weitere Möglichkeiten: Es werden möglichst wenig Variablen in jeder Methode verwendet, sodass die Optimierung des Compilers dafür sorgt, dass diese wenigen Variablen in den Registern des Prozessors gehalten werden. Die zweite Möglichkeit befasst sich mit globalen Variablen bzw. Attributen einer Klasse. Diese werden nicht im Register gehalten, da der Compiler nicht weiß, ob eine Änderung des Wertes im Speicher durchgeführt wurde und liest daher den Wert immer aus dem Speicher. Dies kann umgangen werden, indem der Wert der globalen Variable einer lokalen Variable zugewiesen wird, die im Register gehalten werden kann. Eine weitere Performancesteigerung kann erzielt werden, indem Schleifen mit konstanter Anzahl an Durchläufen auskommentiert werden, sofern diese Anzahl nicht zu groß ist. Dadurch wird der Schleifen-Overhead gespart. Eine letzte Festlegung besteht darin, magische Zahlen im Quellcode zu vermeiden und dafür symbolische Konstanten zu definieren.

4.2 Hochauflösende Zeitmessungen

In diesem Kapitel wird beschrieben, wie eine genaue Zeitmessung möglich ist, mit der selbst die Zeit kleinster Codesegmente bestimmt werden kann. Um diese Genauigkeit zu erreichen, muss der Time-Stamp-Counter des Prozessors ausgelesen werden. Der Time-Stamp-Counter ist ein 64 Bit Register und wird mit jedem Prozessortakt inkrementiert.

Dadurch kann ermittelt werden, wie viele Takte der Prozessor für eine bestimmte Codesequenz benötigt. So kann zwischen mehreren Implementierungsvarianten verglichen werden.

Der folgende Code zeigt, wie der Time-Stamp-Counter ausgelesen wird und wie damit die Zeitdauer eines Codesegmentes bestimmt werden kann.

```
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__rdtsc)

int main()
{
      unsigned __int64 begin, end;

      begin = __rdtsc();
      /*************************/
      // Codesegment
      /*************************/
      end = __rdtsc();

      printf("%I64d Ticks\n", end – begin);
                              //end – begin - 12
 return 0;
}
```

Zuerst muss ein Abgleich durchgeführt werden. Dazu wird der obige Code ohne eingefügtes Codesegment ausgeführt. Das Ergebnis steht für die Zeit, die die Funktion `__rdtsc()` benötigt und muss am Ende der Division mit abgezogen werden. Bei dem verwendeten System betrug dies 12 Ticks. Wird der obige Quellcode mit diesem ermittelten Wert ausgeführt, sollten 0 Ticks als Ergebnis herauskommen. Um die Zeit so genau wie möglich zu bestimmen, muss die Messung mehrmals durchgeführt werden, da parallel ablaufende Prozesse des Betriebssystems den Wert verfälschen können.

4.3 Die Klasse ct_CompareBitpattern

Es werden zwei Möglichkeiten beschrieben den Bitvergleich aus Kap. 3.5 zu implementieren. Diese beiden Möglichkeiten werden verglichen und eine ausgewählt. Anhand dieser Methode wird am Schluss die Struktur der Klasse festgelegt.

Um einen Bitvergleich mit dem 4 Byte breiten Synchronwort durchzuführen, eignen sich die folgenden zwei Möglichkeiten:

1. Möglichkeit:
Der entwickelte Code ist der folgende:

```
bits = bit_pattern ^ Mask;

for(k=0;k<32;k++)
{
  value += (bits & 0x01);
  bits = bits >> 0x01;
}
```

Die Variable `bit_pattern` enthält das zu untersuchende Bitmuster und wird mit dem Bitmuster des Synchronwortes (`Mask`) Exclusiv-Oder verknüpft. Dabei ergeben gleiche Bits eine logische „0“ und unterschiedliche Bits eine logische „1“ an der entsprechenden Bitstelle im Ergebnis `bits`. In der `for`-Schleife wird die Anzahl an unterschiedlichen Bits bestimmt. Dazu wird das LSB auf die Variable `value` aufaddiert und danach die Variable `bits`, bitweise um ein Bit nach rechts verschoben. Das Ergebnis steht fest, wenn alle 32 Bits nach rechts verschoben wurden. Die Variable `value` enthält nun die Anzahl an unterschiedlichen Bits.

2. Möglichkeit:
Bei dieser Möglichkeit werden zwei Look-Up-Tabellen (LUT) verwendet:

```
value =
p_LookUpTable1[(bit_pattern>>BIT_16)] + p_LookUpTable2[bit_pattern & 0xFFFF];
```

Bei dieser Variante wird das Bitmuster (`bit_pattern`) in die oberen 16 Bit und die unteren 16 Bit zerlegt und als Indizes für die beiden LUTs verwendet. Die LUTs liefern die Anzahl an falschen Bits in dem entsprechenden Teilmuster. Um das Ergebnis zu erhalten, müssen diese beiden Werte addiert werden. Für die Erstellung der Tabellen werden die beiden Zeiger `p_LookUpTable1` und `p_LookUpTable2` verwendet. Das Referenzbitmuster des Synchronwortes dient als Voralge um die beiden LUTs zu erstellen:

```
pattern1 = (short)(Mask >> BIT_16);
pattern2 = (short) Mask;
for(k=0;k<=USHRT_MAX;k++)
{
      value = 0;
      bits = k^pattern1;
      for(l=0;l<16;l++)
      {
            value += (bits & 1);
            bits = bits >> 1;
      }
      p_LookUpTable1[k] = value;
```

```
            value = 0;
            bits = k^pattern2;
            for(l=0;l<16;l++)
            {
                  value += (bits & 1);
                  bits = bits >> 1;
            }
            p_LookUpTable2[k] = value;
      }
```

Der Codeausschnitt zeigt wie die Tabellen erstellt werden. Zuerst wird das Referenzbitmuster in zwei Hälften aufgespaltet. Danach wird jeder Wert von k = 0 bis 2^{16} mit jeder der beiden Referenzbitmusterhälften verglichen und die Anzahl an unterschiedlichen Bits wird im Feld `p_LookUpTablex[k]` an der Stelle k eingetragen. Ideal wäre eine Tabelle, die als Index das zu untersuchende Bitmuster übergeben bekommt. Da jedoch das Synchronwort 4 Byte breit ist, müsste die Tabelle rund 4 GByte groß sein. Bei der Verwendung von zwei Tabellen, werden hingegen nur $2*2^{16}$ = 131072 Byte = 128 KB benötigt.

Die zweite Möglichkeit ist die elegantere Variante und ist auch von der Zeitmessung her die schnellere. So betrug die Zeit bei der ersten Möglichkeit 112 Ticks und bei der zweiten 27. Daher wird die Klasse auf Basis der zweiten Möglichkeit entwickelt:
Das erste Attribut ist `Mask` vom Typ `unsigned long` und enthält das Referenzbitmuster. Die beiden Zeiger `p_LookUpTable1` und `p_LookUpTable2` sind vom Typ `unsigned char` und enthalten die entsprechenden Tabellen.
Die Methode `void initLook_up_tables(void)` wurde als `privat` definiert und erstellt die beiden Tabellen nach der oben genannten Methode. Dazu wird zuvor der benötigte Speicherplatz für die beiden Tabellen reserviert. Der Vergleich mit einem Bitmster erfolgt mit der Methode `unsigned int compare(unsigned long bit_pattern)`. Der Rückgabewert enthält die Anzahl an unterschiedlichen Bits des zu untersuchenden Bitmusters `bit_pattern`. Der Konstruktor der Klasse bekommt das Referenzbitmuster übergeben und legt dieses im Attribut `Mask` ab. Danach wird die Methode `void initLook_up_tables(void)` aufgerufen um die Tabellen zu erstellen. Der Destruktor gibt den durch die Methode `void initLook_up_tables(void)` reservierten Speicherplatz frei. Die beiden Methoden `void setMask(unsigned long pattern)` und `unsigned long getMask(void)` wurden entwickelt um das Referenzbitmuster zu verändern bzw. auzulesen. Die Methode `void setMask(unsigned long pattern)` ruft nach der Änderung des Refernzbitmusters die Methode `void initLook_up_tables(void)` auf, um die Tabellen auf das neue Bitmuster hin zu ändern.

4.4 Die Klasse ct_MemoryAndFileManagement

4.4.1 Dateioperationen für binäre Dateien größer 4 GB

Für die Ein- und Ausgabe von binären Dateien, gibt es die folgenden Funktionen. Diese sind in der standardisierten Header-Datei `stdio.h` enthalten.

```
FILE * fopen (const char * filename, const char * mode);
```

Die Funktion `fopen()`öffnet eine durch `filename` definierte Datei mit dem in `mode` angegebenen Modus. Der Parameter `filename` ist ein String, der den Namen der Datei und optional auch die Pfad- und Laufwerksangabe enthält. Der Parameter `mode` kann einen der folgenden Werte annehmen:

Wert	Beschreibung
r	Öffnet eine Datei ausschließlich zum Lesen.
w	Datei zum Schreiben erzeugen. Existiert die Datei bereits, so wird sie überschrieben.
a	Öffnet eine Datei zum Anfügen. Existiert die Datei bereits, so werden neue Daten an das Dateiende angefügt, ansonsten wird die Datei neu erzeugt.
r+	Öffnen einer Datei zum Schreiben und Lesen. Die Datei muss bereits existieren.
w+	Datei zum Schreiben und Lesen erzeugen. Existiert die Datei bereits, so wird sie überschrieben.
a+	Öffnen einer Datei zum Lesen und Anfügen. Existiert die Datei bereits, so werden neue Daten an das Ende angehängt, ansonsten wird die Datei neu erzeugt.

Tabelle 2: Der Parameter mode der Funktion fopen()

Zum Öffnen einer Datei für binäre Ein-/Ausgabe muss zusätzlich der Buchstabe `b` bei `mode` angegeben werden wie z.B. `rb`. Die Funktion `fopen()` liefert bei fehlerfreier Ausführung einen Pointer auf `FILE` zurück. Im Fehlerfall wird `NULL` zurückgegeben.

```
int fclose (FILE * stream);
```

Die Funktion `fclose()` sorgt dafür, dass die Daten, die auf die Festplatte sollen und noch im Dateipuffer stehen, in die Datei geschrieben werden und die durch `stream` definierte Datei geschlossen wird. Die Funktion `fclose()` liefert bei fehlerfreier Abarbeitung den Wert `0` zurück. Im Fehlerfall wird `EOF` zurückgegeben.

```
size_t fwrite (const void * ptr, size_t size, size_t nmemb, FILE * stream);
```

Die Funktion `fwrite()`schreibt `nmemb` Objekte der Größe `size` aus dem Array, auf das der Pointer `ptr` zeigt, in die durch `stream` definierte Datei. Insgesamt werden `nmemb * size` Bytes geschrieben. Der Datentyp `size_t` ist in `stddef.h` als `unsigned int` definiert. Die Funktion `fwrite()` liefert die Anzahl der erfolgreich geschriebenen Objekte zurück. Im Fehlerfall ist die Anzahl der geschriebenen Objekte kleiner als `nmemb`.

```
size_t fread (void * ptr, size_t size, size_t nmemb, FILE * stream);
```

Die Funktion `fread()` liest `nmemb` Objekte der Größe `size` aus der durch `stream` definierten Datei aus und schreibt diese in das Array, auf das der Pointer `ptr` zeigt. Insgesamt werden `nmemb * size` Bytes gelesen. Die Funktion `fread()` liefert die Anzahl der erfolgreich

gelesenen Objekte zurück. Im Fehlerfall oder bei Erreichen des Dateiendes ist diese Anzahl kleiner als `nmemb`.

```
int fseek (FILE * stream, long offset, int whence);
int _fseeki64 (FILE * stream, __int64 offset, int whence);
```

Die Funktion `fseek()` bzw. `_fseeki64()` setzt den Dateipositions-Zeiger der durch `stream` definierten Datei auf die Position, die `offset` Bytes von `whence` entfernt ist. Mit der Funktion `fseek()`, ist aufgrund des Datentyps `long` für den `offset` eine maximal zulässige Dateigröße von rund 4 GB gegeben. Für größere Dateien ist die Funktion `_fseeki64()` gegeben, welche als `offset` eine 64 Bit Variablen des Typs `__int64` erwartet. Die Funktion `_fseeki64()` und der Variablentyp `__int64` sind kein ANSI Standard und laufen nur unter dem Betriebssystem Windows. Für den Parameter `whence` sind in `stdio.h` drei Konstanten definiert:

`SEEK_SET`	`offset` ist relativ zum Dateianfang
`SEEK_CUR`	`offset` ist relativ zur aktuellen Position
`SEEK_END`	`offset` ist relativ zum Dateiende

Ein Aufruf von `fseek()` bzw. `_fseeki64()` mit `whence` gleich `SEEK_END` muss nicht unbedingt unterstützt werden. Ein erfolgreicher Aufruf von `fseek()` bzw. `_fseeki64()` löscht das Dateiende-Flag des Streams. Ist die Datei zum Lesen und Schreiben geöffnet, so kann nach dem Aufruf von `fseek()` bzw. `_fseeki64()`unabhängig davon, was die letzte Ein-/Ausgabeoperation war, gelesen oder geschrieben werden. Die Funktion `fseek()` bzw. `_fseeki64()` liefert bei fehlerfreier Ausführung den Wert `0` zurück. Im Fehlerfall wird ein Wert ungleich `0` zurückgegeben.

```
long ftell (FILE * stream);
__int64 _ftelli64 (FILE * stream);
```

Die Funktion `ftell()` bzw. `_ftelli64()` liefert die Position des Dateipositions-Zeigers der durch `stream` definierten Datei zurück. Dabei wird die Position relativ zum Dateianfang in Bytes gemessen. Mit der Funktion `ftell()`, ist aufgrund der Rückgabe des Datentyps `long` eine maximal zulässige Dateigröße von rund 4 GB möglich. Für größere Dateien ist die Funktion `_ftelli64()` mit der Rückgabe einer 64 Bit Variable des Typs `__int64` gegeben. Die Funktion `_ftelli64()` und der Variablentyp `__int64` sind kein ANSI Standard und laufen nur unter dem Betriebssystem Windows. Die Funktion `ftell()` bzw. `_ftelli64()` liefert bei fehlerfreier Ausführung die aktuelle Position des Dateipositions-Zeigers zurück. Im Fehlerfall wird -1L zurückgegeben und ein implementierungsabhängiger positiver Wert in der Fehlervariablen `errno` gespeichert.

4.4.2 Auslesen des Speichers

Die Telemetry-Daten des Satelliten sind im Byte-Format zusammengefasst und auf der Festplatte abgespeichert. Beim Auslesen der Daten im Format `char` (Byte) treten keine Probleme auf. Diese treten erst auf, wenn auf die Daten mit einem größeren Datentyp wie `short` oder `long` zugegriffen wird. Auch hier liegt dies an der „Little Endian“ Darstellung. So sind beim Datentyp `short` jeweils die beiden Bytes vertauscht, sodass z.B. der Wert `0x1A CF` nach dem Auslesen in der Reihenfolge `0xCF 1A` in der Variablen abgelegt ist. Ebenso verhält

es sich mit dem Datentyp `long`. Bei diesem Datentyp wird aus der Reihenfolge `0x1A CF FC 1D` der entsprechend gespiegelte Wert `0x1D FC CF 1A`.
Damit für die Bitmustersuche, die Bitmusterüberprüfung und das Bitschlupffenster die Werte die im Datentyp `long` übergeben werden, auch richtig interpretiert werden können, muss vorher eine Umsortierung der Bytes erfolgen. Dazu eignet sich der folgende Algorithmus:

```
value  = (unsigned long)p_ReadMemory[index_char] << BIT_24;
value |= (unsigned long)p_ReadMemory[index_char+ONEBYTE] << BIT_16;
value |= (unsigned long)p_ReadMemory[index_char+TWOBYTE] << EIGHTBIT;
value |= p_ReadMemory[index_char+THREEBYTE];
```

Der Zeiger `p_ReadMemory` zeigt auf den Lesespeicher und ist vom Datentyp `unsigned char`. Es werden von der aktuellen Position `index_char` aus vier Bytes eingelesen. Dabei werden die Bytes in der richtigen Reihenfolge in die Variable value vom Datentyp `unsigend long` abgelegt.
Alternativ kann auch der folgende Algoritmus verwendet werden:

```
value_char[ZEROBYTE]  = p_ReadMemory [index_char+THREEBYTE];
value_char[ONEBYTE]   = p_ReadMemory [index_char+TWOBYTE];
value_char[TWOBYTE]   = p_ReadMemory [index_char+ONEBYTE];
value_char[THREEBYTE] = p_ReadMemory [index_char];
p_value_long = (unsigned long*)value_char;
value_long = *p_value_long;
```

Bei dieser Variante, werden die Bytes einzeln in ein 4 Byte großes Feld `value_char` abgelegt. Danach wird die Addresse des Feldes in die Zeigervariablen `p_value_long` vom Datentyp `unsigned long` kopiert. Zuletzt wird der Wert der sich hinter der Addresse befindet in der Zielvariablen `value_long` abgelegt.

Im Gegensatz zur ersten Variante, benötigt die zweite mehr Variablen. Unter dem Gesichtspunkt möglichst wenig Variablen in jeder Methode zu verwenden, wird die erste Variante für die Implementierung gewählt.

Die beiden Varianten gehen von einem Lesespeicher vom Typ `unsigned char` aus. Genausogut kann auch ein Zeiger vom Typ `unsigned long` verwendet werden. Damit die oberen Algorithmen unter dieser Bedingung noch gelten, müsste vorher zusätzlich noch ein Zeigercast von `unsigned long` auf `unsigned char` erfolgen. Für die Klasse wird jedoch als Attribut der Zeiger `p_ReadMemory` vom Datentyp `unsigned char` festgelegt.

4.4.3 Aufbau der Klasse

Die Klasse besteht aus 25 Attributen und 31 Methoden. Zu Beginn werden alle Attribute und deren Bedeutung aufgezählt. Danach werden die Methoden kurz erläutert. Ein detaillierter Einblick in die Implementierung erfolgt im Kap. 4.5.4.

Die verwendeten Attribute und deren Bedeutung werden in der nachfolgenden Tabelle 3 aufgezählt:

Name des Attributes	Datentyp	Bedeutung
`p_ReadPath`	`char *`	Zeiger auf den Dateipfad zum Einlesen
`p_WritePath`	`char *`	Zeiger auf den Dateipfad zum Schreiben

EnableReedSolomonCheck	bool	Flag, das angibt ob ein RS-Check erfolgt
p_FileToRead	FILE *	Dateipointer der Quelldatei
p_FileToWrite	FILE *	Dateipointer der Zieldatei
MemoryWidth	unsigned long	gibt die Speichergröße in Byte an
FrameNumber	unsigned long	maximale Anzahl an Rahmen, die im Schreibspeicher abgelegt werden können
FreeSpaceOnWriteMemory	unsigned long	zählt die Anzahl an Rahmen im Schreibspeicher; beginnt mit dem Wert FrameNumber und zählt runter bis null; bei null ist der Schreibspeicher voll
p_ReadMemory	unsigned char *	Zeiger auf den Lesespeicher
p_WriteMemory	unsigned char *	Zeiger auf den Schreibspeicher
BufferWriteMemory	unsigned char	Zwischenspeicher für das bytesynchronisierte Kopieren der Rahmen
BufferReadMemory	unsigned long	Zwischenspeicher für den Lesespeicher
IndexWriteMemory	unsigned long	aktuelle Byteposition im Schreibspeicher
IndexReadMemoryChar	unsigned long	aktuelle Byteposition im Lesespeicher
IndexReadMemoryLong	unsigned long	gibt die Position im Lesespeicher an, wenn auf diesen ein Zeiger vom Typ unsigned long zeigt
MaxIndexChar	unsigned long	maximaler Index des Lesespeichers
MaxIndexCharOld	unsigned long	alter maximaler Index des Lesespeichers
MaxIndexLong	unsigned long	maximaler Index des Lesespeichers, wenn ein Zeiger vom Typ unsigned long verwendet wird
FramesWithErrors	unsigned long	zählt die fehlerhaften Rahmen und wird zyklisch ausgelesen und gelöscht
AllFramesWithErrors	unsigned long	zählt alle fehlerhaften Rahmen
BitShift	unsigned int	aktuelle Bitverschiebung
BeginToCopy	unsigned long	Index, ab dem das bytesynchronisierte Kopieren begonnen bzw. fortgesetzt wird
FilePosition	__int64	enthält die aktuelle Anzahl, der von der Festplatte gelesenen Bytes
ReadStatus	bool	Flag, das angibt ob gerade neu eingelesen wurde, während IndexReadMemoryChar noch auf das Ende des Lesespeichers zeigt (verhindert doppeltes Einlesen)
FrameAtFileEnd	bool	Flag, das angibt ob ein kompletter Rahmen am Dateiende existiert

Tabelle 3: Attribute der Klasse ct_MemoryAndFileManagement

Die Klasse besitzt einen eigenen Konstruktor. Dieser fordert den benötigten Speicherplatz für den Lese- und Schreibspeicher an, sowie den Platz für die beiden char-Arrays zum Ablegen des Quell- und Zieldateipfades in den zugehörigen Attributen. Weiterhin erfolgt eine erste Initialisierung der Attribute. Der Destruktor gibt den reservierten Speicher wieder frei und schließt die beiden durch p_FileToWrite und p_FileToRead definierten Dateien.

Die definierten Methoden sind in der nachfolgenden Tabelle 4 aufgelistet und erläutert:

Methodenname	Rückgabewert	Parameter	Funktion
`read_frames_from_file`	`void`	`void`	liest einen Block der Größe `MemoryWidth` aus der Datei und legt diesen im Lesespeicher ab
`write_synchronized_bytes_to_file`	`void`	`void`	speichert den gesamten Schreibspeicher ab
`write_synchronized_bytes_to_file`	`void`	`unsigned long number_of_bytes`	speichert `number_of_bytes` Bytes des Schreibspeichers ab
`decreaseFreeSpaceOnWriteMemory`	`void`	`void`	zählt die Rahmen und gegebenenfalls werden diese bytesynchronisiert kopiert und abgespeichert
`aktualize_FilePosition`	`void`	`void`	erhöht `FilePosition` um die Anzahl neu eingelesener Bytes
`IndexReadMemoryChar_plus_one`	`void`	`void`	inkrementiert `IndexReadMemoryChar`
`IndexReadMemoryChar_minus_one`	`void`	`void`	dekrementiert `IndexReadMemoryChar`
`synchronize_bytes`	`void`	`void`	führt eine Bytesynchronisation bis zum Ende des Lesespeichers durch
`synchronize_bytes`	`void`	`unsigned long number_of_bytes`	führt eine Bytesynchronisation der Anzahl number_of_bytes Bytes durch
`open_file_to_write`	`void`	`void`	öffnet bzw. legt die Zieldatei an
`getp_ReadPath`	`char *`	`void`	liefert `p_ReadPath`
`getp_WritePath`	`char *`	`void`	liefert `p_WritePath`
`getIndexReadMemoryLong`	`unsigned long`	`void`	liefert `IndexReadMemoryLong`
`getIndexReadMemoryChar`	`unsigned long`	`void`	liefert `IndexReadMemoryChar`
`getBufferReadMemory`	`unsigned long`	`void`	liefert `BufferReadMemory`
`getEnableReedSolomonCheck`	`bool`	`void`	liefert `EnableReedSolomonCheck`
`getBitShift`	`unsigned int`	`void`	liefert `BitShift`
`get_and_reset_FramesWithErrors`	`unsigned long`	`void`	liefert `FramesWithErrors`, erhöht `AllFramesWithErrors` um diesen Wert und löscht diesen
`getAllFramesWithErrors`	`unsigned long`	`void`	liefert `AllFramesWithErrors`
`getFrameAtFileEnd`	`bool`	`void`	liefert `FrameAtFileEnd`
`getReadStatus`	`bool`	`void`	liefert `ReadStatus`
`getFilePosition`	`__int64`	`void`	liefert `FilePosition`
`getBytePosition`	`__int64`	`void`	berechnet und liefert die aktuelle Byteposition in der Datei
`get_next_long_value_from_read_memory`	`bool` (Dateiende = `true`)	`unsigned long &value`	liefert die jeweils nächsten 4 Bytes
`get_long_value_from_read_memory`	`unsigned long`	`unsigned long index_char`	liefert die folgenden 4 Bytes an der Stelle `index_char` im Lesespeicher
`get_char_value_from_read_memory`	`bool` (Dateiende = `true`)	`unsigned int byte_offset, unsigned long &value1, unsigned long &value2`	liefert die in Rahmenlänge (`byte_offset`) entfernten 8 Bytes in den Variablen `value1` und `value2`
`setIndexReadMemoryLong`	`void`	`unsigned long index`	`IndexReadMemoryLong` bekommt den Wert der Variablen `index`
`setBufferReadMemory`	`void`	`unsigned long value`	`BufferReadMemory` bekommt den Wert der Variablen `value`
`set_index_char_to_int`	`void`	`void`	`IndexReadMemoryLong` wird neu initialisiert
`set_begining`	`void`	`unsigned int byte, unsigned int bit_shift`	dient zur Initialisierung von `BufferWriteMemory`, `BeginToCopy`, `BitShift` und `IndexReadMemoryChar`

`reset_write_memory_options`	`void`	`void`	löscht `IndexWriteMemory` und setzt `FreeSpaceOnWriteMemory` gleich `FrameNumber`
`open_file_to_read`	`void`	`void`	öffnet die Quelldatei
`rescue_data_after_sync_lose_or_file_end`	`void`	`unsigned int frame_length, bool end_of_file`	synchronisiert und speichert sofort alle Rahmen aus dem Lesespeicher ab
`synchronize_bit_slip`	`void`	`unsigned int mode, unsigned long value1, unsigned long value2, unsinged int new_bit_shift`	synchronisiert beim Auftreten von Bitschlupf und passt `BitShift`, `IndexReadMemoryChar`, `BeginToCopy` und `BufferWriteMemory` an die neuen Verhältnisse an

Tabelle 4: Methoden der Klasse ct_MemoryAndFileManagement

4.4.4 Implementierung der Methoden

Im Kapitel zuvor wurden die Attribute und Methoden vorgestellt. Nun wird auf die Implementierung von einigen Methoden eingegangen.

- Die Methode `void read_frames_from_file(void)`liest mittels der Funktion `fread()` Daten von der Festplatte in den Lesespeicher. Der Rückgabewert dieser Funktion wird in `MaxIndexChar` kopiert. Zuvor wird jedoch der alte Wert `MaxIndexChar` in dem Attribut `MaxIndexCharOld` abgelegt. Der Wert für `MaxIndexLong` ergibt sich aus der Rechnung: `MaxIndexChar/FOURBYTE`. Zuletzt wird `FilePosition` durch den Aufruf der Methode `aktualize_FilePosition()` aktualisiert.

- Die Methode `bool get_next_long_value_from_read_memory(register unsigned long &value)` liefert die jeweils nächsten 4 Byte des Lesespeichers. Dazu wird zu Beginn das Attribut `IndexReadMemoryLong` inkrementiert, damit dieser Index auf die nächsten 4 Bytes weist. Zunächst muss der Index überprüft werden, ob dieser außerhalb des Lesespeichers liegt. Dies wird durch einen Vergleich mit dem Attribut `MaxIndexLong` erzielt. Ist der Index größer oder gleich dem maximalen Index, muss neu eingelesen werden. Zuerst werden die letzten 4 Bytes des Lesespeichers in dem Attribut `BufferReadMemory` abgelegt:

```
BufferReadMemory = get_long_value_from_read_memory(MaxIndexChar - FOURBYTE);
```

  Danach wird `IndexReadMemoryLong` auf `0` gesetzt und mit dem Aufruf der Methode `read_frames_from_file()` neu eingelesen. Das Dateiende ist erreicht, wenn `MaxIndexChar` kleiner 4 d.h. weniger als 4 Bytes im Lesespeicher sind. Die Methode wird dann mit dem Parameter `true` beendet. Ist hingegen das Dateiende noch nicht erreicht oder musste eventuell nicht eingelesen werden, so wird entsprechend dem Kap. 4.5.2 ausgelesen und die Methode mit dem Parameter `false` beendet.

- Mit der Methode `void set_begining(unsigned int byte, unsigned int bit_shift)` wird diese Klasse nach erfolgreich gefundenem Synchronwort durch die Klasse `ct_FrameSynchronizer` initialisiert. Zuerst wird geprüft, ob der Index `IndexReadMemoryLong` kleiner `1` ist. Ist dies der Fall, so wurde neu eingelesen und die aktuelle Position des Synchronwortes lag am Ende des Lesespeichers. Daher muss zuerst das Flag `ReadStatus = true` gesetzt werden und die Daten müssen manuell in den Schreibspeicher synchronisiert werden:

```
value = (MaxIndexCharOld - FOURBYTE) + byte;
byte_value[0] = (unsigned char) (BufferReadMemory >> BIT_24);
byte_value[1] = (unsigned char) (BufferReadMemory >> BIT_16);
byte_value[2] = (unsigned char) (BufferReadMemory >> EIGHTBIT);
byte_value[3] = (unsigned char)  BufferReadMemory;
for(k=0;k<(3-byte);k++)
{
      p_WriteMemory[k] = (byte_value[k + byte] << bit_shift) |
     (byte_value[k + byte + 1] >> (EIGHTBYTE - bit_shift));
}
```

Die Variable `value` gibt die Byteposition des Synchronwortes vor dem Einlesen an und wird berechnet, indem zuerst 4 Bytes (4 Byte Block) von der alten Lesespeichergröße abgezogen werden. Dies wäre die Position des Synchronwortes auf 4 Bytes geanau. Als nächstes muss auf diesen Wert noch die Variable `byte` addiert werden. Diese Variable gibt an, um wie viele Bytes (maximal drei) der Anfang des Synchronwortes von dieser Position entfernt ist. Die Variable `bit_shift` gibt die genaue Verschiebung des Synchronwortanfangs in diesem Byte an. Als Nächstes werden die Bytes des Zwischenspeichers `BufferReadMemory` in dem Feld `byte_value` abgelegt und das Synchronwort das in diesem Zwischenspeicher enthalten ist byteweise in den Lesespeicher synchronisiert. Danach werden die zugehörigen Attribute initialisert:

```
IndexWriteMemory = 3 - byte;
BufferWriteMemory = byte_value[3];
BeginToCopy = 0;
```

Das Attribut `IndexWriteMemory` muss um die Anzahl an in den Schreibspeicher geschriebenen Bytes erhöht werden und das Attribut `BufferWriteMemory` erhält das letzte Byte des Zwischenspeichers, da dies für die spätere Vortführung des bytesynchronisierten Kopierens benötigt wird. Dies erfolgt dann ab dem Anfang des Lesespeichers und deshalb wird `BeginToCopy` auf `0` gesetzt. Ist hingegen das Attribut `IndexReadMemoryLong` größer `0`, wurde noch nicht neu eingelesen und somit ist das Synchronwort im aktuellen Lesespeicher enthalten. Die Variable `value` berechnet sich nun folgendermaßen:

```
value = FOURBYTE * (IndexReadMemoryLong - 1) + byte;
```

`IndexReadMemoryLong` zeigt immer auf den jeweils nächsten 4 Byte Block. Daher wird dieses Attribut um eins erniedrigt und mit der Blockgröße multipliziert um die Byteposition des Synchronwortes wiederum auf 4 Byte genau zu bestimmen. Wird nun die Variable `byte` auf diesen Wert addiert, erhält man wiederum die genaue Byteposition. Das Attribtut `BufferWriteMemory` erhält das Byte an dieser Stelle und `BeginToCopy` zeigt ein Byte weiter. Zum Schluss wird die Bitverschiebung `bit_shift` in dem Attribut `BitShift` kopiert und der Index `IndexReadMemoryChar` erhält den Wert der Variablen `value` und zeigt nun auf das aktuelle Synchronwort im Lesespeicher.

- Als nächstes wird die Methode `bool get_char_value_from_read_memory(register unsigned int byte_offset, register unsigned long &value1, register unsigned long &value2)` näher beschrieben. Die Methode signalisiert mit dem Rückgabewert `true`, dass das Dateiende erreicht wurde. Die Methode liefert die von der Position `IndexReadMemoryChar` um `byte_offset` Byte entfernt liegenden Daten in den Referenzvariablen `value1` und `value2`. Zuerst wird die neue Position `position` berechnet, indem `byte_offset` auf das Attribut `IndexReadMemoryChar` aufaddiert wird. Ab dieser

Position werden die ersten 4 Byte in `value1` und die nächsten 4 Byte in `value2` kopiert. Daher wird zuerst geprüft ob `position` und die 8 Byte innerhalb des Feldes liegen:

```
if((position + EIGHTBYTE) <= MaxIndexChar)
```

Trifft die Bedingung zu, werden die Bytes ausgelesen, `IndexReadMemoryChar` aktualisiert d.h. es nimmt den Wert von `position` an und die Methode `decreaseFreeSpaceOnWriteMemory()` aufgerufen, um diesen Rahmen zu zählen.
Trifft die Bedingung nicht zu, so ist entweder `position` außerhalb des Feldes oder nur ein Teil der 8 Byte sind im Speicher und der andere Teil muss erst noch von der Datei eingelesen werden. Daher wird zuerst berechnet um wie viele Bytes `position + EIGHTBYTE` über den Speicherbereich hinausragt und somit nach dem Einlesen am Speicheranfang benötigt wird:

```
carry_over = (position + EIGHTBYTE) – MaxIndexChar;
```

Nun ergeben sich acht Möglichkeiten die auftreten können: Von den acht Bytes kann sowohl ein Byte bis hin zu sieben Bytes außerhalb des Lesespeichers liegen oder der Standort des Synchronwortes `position` liegt komplett außerhalb. Diese acht Möglichkeiten werden innerhalb einer `switch-case` Anweisung bearbeitet. Mit den Werten `switch (carry_over)` gleich eins bis sieben, werden die Fälle, dass ein Byte bis sieben Bytes nach neuem Einlesen aus dem Lesespeicher noch eingelesen werden müssen, behandelt. Die Abarbeitung ist hierbei immer gleich: Zuerst wird `IndexReadMemoryChar` mit dem Wert von `position` aktualisiert und danach wird die Anzahl an Bytes, die noch am Ende des Speichers vorhanden ist, eingelesen. Als nächstes werden vor dem Einlesen eines neuen Datenblocks mit der Methode `read_frames_from_file()` die 4 Bytes vor der Position `position` in das Attribut `BufferReadMemory` kopiert, danach die Methode `decreaseFreeSpaceOnWriteMemory()` zum Zählen des Rahmens aufgerufen und alle Daten des Lesespeichers durch die Methode `synchronize_bytes()` bytesynchronisiert in den Schreibspeicher kopiert. Nachdem neu eingelesen wurde, wird das Flag `ReadStatus = true` gesetzt und überprüft ob das Dateiende erreicht wurde. Ist dies der Fall wird `FrameAtFileEnd` auf `true` gesetzt und die Methode mit dem Rücksprungwert `true` beendet. Ist das Dateiende nicht erreicht, werden die restlichen Bytes in die beiden Variablen `value1` und `value2` kopiert und die Methode mit dem Rücksprungwert `false` beendet. Der letzte Fall wird mit dem `switch-case` Zweig `default` bearbeitet. Hier wird zuerst die neue Position `position` berechnet, an der nach dem Einlesen das Synchronwort liegt:

```
position = carry_over – EIGHTBYTE;
IndexReadMemoryChar = position;
```

Nun muss das Flag `ReadStatus` geprüft werden, ob überhaupt eingelesen werden muss. Ist der Status `false`, werden erst die letzten 4 Byte des Lesespeichers in das Attribut `BufferReadMemory` kopiert, danch mit der Methode `synchronize_bytes()` eine synchronisation der Daten durchgeführt und danach neu eingelesen. Nun wird auf das Dateiende hin geprüft und wenn dieses erreicht ist, ob noch ein kompletter Rahmen existiert:

```
if(MaxIndexChar < carry_over)
{     if(MaxIndexChar >= position)
      {
            FrameAtFileEnd = true;
            synchronize_bytes(position + ONEBYTE);
            decreaseFreeSpaceOnWriteMemory();
      }
      return true;
}
```

Ist der Status des Flags hingengen `true` muss nicht eingelesen werden und das Flag wird zurück auf `false` gesetzt. Am Ende der `default` Anweisung werden die 8 Byte ab `position` in die Variablen `value1` und `value2` kopiert und die Methode `decreaseFreeSpaceOnWriteMemory()` aufgerufen.

- Die Methode `void decreaseFreeSpaceOnWriteMemory(void)` zählt jeweils einen Rahmen und dekrementiert dazu das Attribut `FreeSpaceOnWriteMemory`. Erreicht der Wert des Attributes `0`, so sind genug Rahmen vorhanden um den Schreibspeicher vollständig zu füllen und zu speichern. Dazu wird zuerst die Anzahl an Bytes bestimmt, die noch aus dem Lesespeicher in den Spreibspeicher kopiert werden müssen:

```
value = MemoryWidth - IndexWriteMemory;
```

Diese Anzahl wird bytesynchronisiert kopiert und danach der gesamte Inhalt des Schreibspeichers abgespeichert:

```
synchronize_bytes(value);
write_synchronized_bytes_to_file();
```

Am Ende wird die Methode `reset_write_memory_options()` aufgerufen um `IndexWriteMemory` auf den Anfang des Schreibspeichers zu setzten (`0`) und `FreeSpaceOnWriteMemory` mit der maxiamalen Rahmenanzahl `FrameNumber` neu zu initialisieren.

- Als nächstes werden die beiden Methoden `void synchronize_bytes(void)` und `void synchronize_bytes(register unsigned long number_of_bytes)` näher beschrieben. Beide Methoden haben den gleichen Quellcode bis auf zwei Unterschiede: Die Methode ohne Übergabeparameter synchronisiert bis zum Ende des Lesespeichers und berechnet die Anzahl der zu synchronisierenden Bytes nach folgender Formel:

```
number_of_bytes = MaxIndexChar - BeginToCopy;
```

Während die andere Methode diese Anzahl als Parameter übergeben bekommt. Der zweite Unterschied besteht darin, dass das Attribut `BeginToCopy` nach der Synchronisation der Daten durch die Methode ohne Parameter auf `0` gesetzt wird weil die Synchronisation später vom Anfang des Lesespeichers weitergeführt wird. Hingegen bei der Methode mit Parameterübergabe wird `number_of_bytes` auf `BeginToCopy` hinzuaddiert. Zum Anfang wird geprüft ob `number_of_bytes` gleich `0` ist, da dieser Wert auftreten kann und in diesem Fall keine Daten synchronisiert werden müssen und die Methode beendet wird. Ist die Bitverschiebung `BitShift` gleich `0` so werden die Bytes direkt aus dem Lesespeicher in den Schreibspeicher kopiert:

```
p_write_memory_char[0] = BufferWriteMemory;

for(i=1;i<number_of_bytes;i++)
{
      p_write_memory_char[i] = p_read_memory_char[i-1];
}
```

Die Variable `p_write_memory_char` ist ein Zeiger auf die Position `IndexWriteMemory` im Schreibspeicher und die Variable `p_read_memory_char` ist ein Zeiger auf die Position

`BeginToCopy` im Lesespeicher. Bei einer Bitverschiebung ungleich `0` werden die Bytes folgendermaßen bytesynchrinisiert kopiert:

```
p_write_memory_char[0] = (BufferWriteMemory << bit_shift) |
       (p_read_memory_char[0] >> (EIGHTBIT - bit_shift));

for(i=1;i<number_of_bytes;i++)
{

p_write_memory_char[i] = (p_read_memory_char[i-1] << bit_shift) |
       (p_read_memory_char[i] >> (EIGHTBIT - bit_shift));
}
```

Die Synchronisation erfolgt indem ein Byte um den Wert `bit_shift` nach links geschoben wird und das nächste Byte um die Anzahl an fehlendenden Bits (`EIGHTBIT - bit_shift`) entsprechend nach rechts geschoben wird und die beiden Ergebnisse am Ende bitweise Oder-Verknüft und im Schreibspeicher an der entsprechenden Stelle abgelegt wird. Zum Schluss wird `BufferWriteMemory` neu festgelegt und erhält das Byte an der Stelle `p_read_memory_char[number_of_bytes-ONEBYTE]` im Lesespeicher und `IndexWriteMemory` wird um die Anzahl an Bytes `number_of_bytes` erhöht.

- Die letzte Methode, die näher erläutert wird, ist die Methode `void synchronize_bit_slip (unsigned int mode, unsigned long value1, unsigned long value2,unsigned int new_bit_shift)`. Diese Methode wird durch die Methode `search_in_bitslip_window` der Klasse `ct_FrameSynchronizer` aufgerufen um aufgetretenen Bitschlupf zu korrigieren. Zuerst muss das Flag `ReadStatus` überprüft werden. Hat dieses den Wert `false`, existieren noch byteunsynchonisierte Daten im Lesespeicher, die mit der alten Bitverschiebung `BitShift` bis zu der Stelle des Synchronwortes (`IndexReadMemoryChar`) bytesynchronisiert werden müssen.
  Ist hingegen der Wert des Flags gleich `true`, wurde erst neu eingelesen und zuvor eine Bytesynchronisation der Daten durchgeführt. In diesem Fall wurde das durch Bitschlupf verschobene Synchronwort mit der falschen Bitverschiebung synchronisiert und muss nun manuell mit der neuen Bitverschiebung `new_bit_shift` im Schreibspeicher korrigiert werden. Dazu muss zu Beginn die Anzahl an Bytes bestimmt werden, die entweder bytesynchronisiert oder im Schreibspeicher korrigiert werden sollen.
  Danach wird über eine `switch-case` Anweisung, je nachedem welchen Wert der Parameter `mode` hat, eine von drei Möglichkeiten gewählt: `CURRENTBYTE` = 0 bedeutet das der Anfang des Synchronwortes noch an der Position `IndexReadMemoryChar` im Lesespeicher zu finden ist. Hingegen mit den symbolischen Konstanten `NEXTBYTE` = 1 und `LASTBYTE` = 2 wird signalisiert, dass der Anfang des Synchronwortes ein Byte später bzw. ein Byte früher von der Position `IndexReadMemoryChar` im Lesespeicher zu finden ist. Je nachdem welche der drei Möglichkeiten aufgetreten ist, wird in Abhängigkeit des Flags `ReadStatus` entweder der Schreibspeicher korrigiert oder alle Bytes bis zu dem Synchronwort synchronisiert. Danach wird `BitShift` mit dem Wert der neuen Bitverschiebung `new_bit_shift` aktualisiert und das Attribut `IndexReadMemoryChar` wird an die neue Position des Synchronwortes angepasst d.h. es wird nicht verändert oder um eins erhöht oder erniedrigt.

4.5 Die Klasse ct_FrameSynchronizer

Zu Beginn wird kurz der Aufbau der Klasse beschrieben und danach detaillierter auf die Implementierung der Methoden in Bezug zu den in Kap. 3 erarbeiteten Konzept eingegangen.

Bei dieser Klasse handelt es sich um die Basisklasse des Programms. Die Schnittstelle nach außen ist durch die drei Methoden `bool locked_mode(void)`, `bool search_mode(void)` und `void output_on_screen_and_file(unsigned int mode)` gegeben. Die beiden zuerst genannten Methoden entsprechen vom Ablauf her dem in Kap. 3.4 entwickelten Flussdiagramm. Die Klasse besitzt die folgenden 15 Attribute (Tabelle 5):

Name des Attributes	Datentyp	Bedeutung
`FrameLength`	`unsigned int`	beinhaltet die Rahmenlänge
`BitErrorMax`	`unsigned int`	maximal zulässige Anzahl an Bitfehlern im Synchronwort
`BitSlipWindowSize`	`unsigned int`	Größe des Bitschlupffensters
`NumberOfKnownPatternsToLock`	`unsigned int`	Anzahl an erkannten Synchronwörtern, um in den Zustand „Locked“ überzugehen
`NumberOfUnknownPatternsToSearch`	`unsigned int`	Anzahl an nicht erkannten Synchronwörtern, um in den Zustand „Search“ überzugehen
`OutputBlockSize`	`unsigned long`	ab dieser Anzahl an Rahmen erfolgt jeweils eine Ausgabe der Qualitätsparameter
`OutputFile`	`ofstream` Klasse	Objekt der Ausgabedatei(Text)
`co_CompareBitpattern`	`ct_CompareBit pattern` (Klasse)	Objekt für den Bitmustervergleich
`BitErrorCounter`	`unsigned long`	zählt die gesamte Anzahl an Bit-fehlern in den Synchronwörtern
`BlockBitErrorCounter`	`unsigned long`	zählt die Bitfehler pro Block
`FrameCounter`	`unsigned long`	gesamte Anzahl an Rahmen
`BlockFrameCounter`	`unsigned long`	zählt die Rahmen des Blockes
`BitSlipCounter`	`unsigned long`	gesamte Anzahl an Bitslips
`BlockBitSlipCounter`	`unsigned long`	zählt die Bitslips pro Block
`FlyWheelFrames`	`unsigned long`	gesamte Anzahl an nicht erkannten Rahmen
`BlockFlyWheelFrames`	`unsigned long`	zählt die nicht erkannten Rahmen pro Block

Tabelle 5: Attribute der Klasse ct_FrameSynchronizer

Die definierten Methoden werden in der Tabelle 6 aufgezählt:

Methodenname	Rückgabewert	Parameter	Funktion
`search_bitpattern`	`bool` (Dateiende = true)	`void`	sucht das Synchronwort im Datenstrom
`check_next_pattern`	`undigned int` (Dateiende = 1,Synchron = 0,Nicht Synchron = 2)	`void`	überprüft die folgenden Synchronwörter im Zustand „Search“
`search_in_bitslip_window`	`unsigned int` (Misserfolg = oxFF, sonst Bitfehler im Synchronwort)	`unsigned long value1, unsigned long value2`	sucht das Bitmuster innerhalb eines vorgegebenen Bereiches im Datenstrom

`open_output_file`	`void`	`char * writepath`	öffnet bzw. erzeugt die durch den Dateipfad `writepath` definierte Ausgabedatei
`getFrameCounter`	`unsigned long`	`void`	liefert `FrameCounter`
`getBitErrorCounter`	`unsigned long`	`void`	liefert `BitErrorCounter`
`getBitSlipCounter`	`unsigned long`	`void`	liefert `BitSlipCounter`
`getFlyWheelFrames`	`unsigned long`	`void`	liefert `FlyWheelFrames`
`getFrameLength`	`unsigned int`	`void`	liefert `FrameLength`
`getNumberOfUnknownPatternsToLock`	`unsigned int`	`void`	liefert das Attribut
`output_on_screen_and_file`	`void`	`unsigned int mode`	zuständig für die Ausgabe in die Datei und auf dem Bildschirm
`locked_mode`	`bool` (Dateiende)	`void`	Zustand „Locked“, sammelt auch die Qualitätsparameter
`search_mode`	`bool` (Dateiende)	`void`	Zustand „Search“

Tabelle 6: Methoden der Klasse ct_FrameSynchronizer

- Der Konstruktor der Klasse bekommt die in Kap. 3.3.1 genannten Parameter übergeben. Des Weiteren werden die Konstruktoren der Klasse `ct_CompareBitpattern` und `ct_MemoryAndFileManagment` mit den benötigten Parametern und die Methode `void open_output_file(char *writepath)` zum Öffnen der Ausgabedatei aufgerufen.

- Die Methode `bool search_mode (void)` ist repräsentativ für den Zustand „Search“ (Abbildung 14). Der Rückagebwert ist vom Typ `bool` und hat den Wert `true` wenn das Dateiende erreicht bzw. `false` wenn das Dateiende noch nicht erreicht wurde. Die Basis bildet eine `do-while` Schleife die erst verlassen wird, wenn die Methode `search_bitpattern` ein Synchronwort gefunden hat und die Methode `check_next_pattern` dieses durch mehrmaliges Auffinden der folgenden Synchronwörter bestätigt. Ist dies der Fall, wird die Methode mit dem Rückgabeparameter `false` beendet.

- Wie schon erwähnt ist die Methode `bool search_bitpattern(void)` für die Bitmustersuche im Datenstrom zuständig und wurde nach dem Flussdiagramm in Abbildung 16 entwickelt. Der Rückgabewert gibt ebenfalls das Dateiende an. Zuerst muss mit dem Aufruf der Methode `reset_write_memory_options()` der `ct_MemoryAndFileManagment` Klasse diese darüber informiert werden, dass die bytesynchronisierten Rahmen im Schreibspeicher verworfen werden können. Daher ist es wichtig, dass sofern vorher die Methode `locked_mode` ausgeführt wurde, diese der Klasse `ct_MemoryAndFileManagment` mitteilt, die gefunden bytesynchronisierten Rahmen abzuspeichern, damit diese nicht verloren gehen. Wurde vorher die Methode `check_next_pattern` ausgeführt, muss dies nicht erfolgen, da sich die gefundenen Rahmen unter diesem Umstand als falsch erwiesen haben. Um das Bitmuster zu suchen, dienen die beiden lokalen Variablen `zmemory` und `zpattern`. Zunächst wird der erste Datenblock angefordert und in der Variablen `zpattern` abgelegt:

```
end_of_file = get_next_long_value_from_read_memory(zpattern);
if(end_of_file) //Dateiende
      return true;
```

Ist das Dateiende erreicht, wird die Methode mit dem Rückgabewert `true` beendet. Wenn nicht, werden in einer `while(1)`–Endlosschleife alle weiteren Datenblöcke in der Variablen `zmemory` abgelegt. Wird das richtige Bitmuster gefunden oder das Dateiende erreicht, erfolgt ein Rücksprung mit dem entsprechenden Rücksprungparameter. Die Variable `zpattern` ist das Bitmuster, das mit dem des Synchronwortes verglichen wird:

```
value = co_CompareBitpattern.compare(zpattern);
```

Liegt es innerhalb der Bitfehlergrenze, wird die Klasse `ct_MemoryAndFileManagment` initialisiert und die Funktion verlassen:

```
if(value <= BitErrorMax)
{
      set_begining(ZEROBYTE, ZEROBIT);
      return false;
}
```

Mit der Übergabe von symbolischen Konstanten wie hier z.B. `ZEROBYTE` und `ZEROBIT` wird angegeben, wo im 4 Byte Block der Anfang des Bitmuster gefunden wurde. Handelt es sich nicht um das gesuchte Bitmuster, werden die Bits der Variablen `zpattern` um ein Bit nach links verschoben und danach das MSB der Variablen `zmemory` an die Stelle des LSB in der Variablen `zpattern` eingefügt. Implementiert sieht das folgendermaßen aus:

```
zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 31);
```

Führt auch dieses Bitmuster nicht zum Erfolg, wird das nächste Bit aus der Variablen `zmemory` an der Stelle des LSB in der Variablen `zpattern` abgelegt und das neue Bitmster wieder überprüft. Das geht so lange, bis alle 32 Bit der Variablen `zmemory` in die Variable `zpattern` kopiert wurden. Danach wird zum Anfang der `while(1)`–Schleife gesprungen, ein neuer Block in der Variablen `zmemory` abgelegt und der Kreislauf beginnt von neuen.

- Die Methode `unsigned int check_next_pattern(void)` besitzt drei Rückgabewerte, die entsprechend drei Konstanten zugeordnet sind: `SYNC` = 0 bei erfolgreich hergestellter Synchronisation, `ENDOFFILE` = 1 bei Erreichen des Dateiendes und `NOSYNC` = 2 wenn nicht die erforderliche Anzahl an Synchronwörtern in Folge gefunden wurde. In einer `do-while` Schleife wird jeweils das nächste Synchronwort im Abstand der Rahmenlänge untersucht. Die Variable `approach` zählt die Anzahl an gefundenen Synchronwörtern und wird daher vor jedem neuen Schleifendurchlauf inkrementiert. Entspricht die Anzahl dem Attribut `NumberOfKnownPatternsToLock`, wird die Schleife beendet und die Anzahl an gefundenen Rahmen (`approach`) und Bitfehlern in den entsprechenden Attributen abgelegt. In diesem Fall wird die Methode regulär mit der Konstante `SYNC` beendet. Um in der Schleife das nächste Synchronwort zu untersuchen, liefert die Methode `get_char_value_from_read_memory` der `ct_MemoryAndFileManagment` Klasse die entsprechenden beiden 4 Byte Blöcke im Rahmenabstand. Ist der Rückgabewert true, wird die Methode mit dem Rückagabewert `ENDOFFILE` beendet. Die Untersuchung des Bitmusters erfolgt folgendermaßen:

```
if(bit_shift != ZEROBIT)
{
      memory_one = (memory_one << bit_shift) |
      (memory_two >> (BIT_32 - bit_shift));
}

count = co_CompareBitpattern.compare(memory_one);
bit_error += count;
```

Die Variable `bit_shift` enthält den Grand der Bitverschiebung, der als Attribut in der Klasse `ct_MemoryAndFileManagment` abgelegt ist und nur über die entsprechende Methode `getBitShift()` angefordert werden kann. Vor dem Vergleich wird das zu untersuchende Bitmuster in der Variablen `memory_one` vom Typ `unsigned long` abgelegt. Das erfolgt indem die Variable `memory_one` um `bit_shift` Bits nach links verschoben wird und um die

nun fehlende Anzahl an Bits aus der Variablen `memory_two` aufgefüllt wird. Nach dem Vergleich wird die Anzahl an unterschiedlichen Bits der Variablen `bit_error` aufaddiert. Danach erfolgt die Abfrage: `if(count > BitErrorMax)` um die Gültigkeit zu überprüfen. Trifft die Bedingung zu, wird die Methode mit dem Rücksprungparameter `NOSYNC` beendet. Zuvor muss das Attribut `IndexReadMemoryLong` der `ct_MemoryAndFileManagment` Klasse über die entsprechenden Methoden so geändert werden, dass die Suche des Synchronwortes ab dem letzten gefundenen erfolgt. Stellt sich jedoch das Bitmuster als Synchronwort heraus, wird das nächste Synchronwort untersucht bzw. die Schleife wieder durchlaufen.

- Die Struktur der Methode `bool locked_mode(void)` ähnelt der Methode zuvor. Die Unterschiede bestehen darin, dass zum Einen der Rückgabewert wieder angibt ob das Dateiende erreicht ist. Der Rücksprung mit diesem Parameter erfolgt wiederum anhand des Rückgabewertes der `get_char_value_from_read_memory` Methode der `ct_MemoryAndFileManagment` Klasse:

```
end_of_file = get_char_value_from_read_memory(FrameLength,
              memory_one, memory_two);
if(end_of_file) //Dateiende?
{
      if(getFrameAtFileEnd())
            BlockFrameCounter++;

      rescue_data_after_sync_lose_or_file_end(FrameLength,
                                              end_of_file);
      return end_of_file;
}
```

Ist das Flag `FrameAtFileEnd` gesetzt, muss dieser Rahmen mitgezählt werden und es wird der `ct_MemoryAndFileManagment` Klasse mitgeteilt, dass alle restlichen gefundenen Rahmen synchronisiert und abgespeichert werden müssen. Desweiteren wird bei Nichterkennen des Synchronwortes zuerst die Methode `search_in_bitslip_window` aufgerufen um das Bitmuster innerhalb des Fensters zu suchen. Führt dies nicht zum Erfolg, wird die Variable `approach` als Indikator für die Fehlversuche und das Attribut `BlockFlyWheelFrames` inkrementiert. Wird hingegen das Bitmuster innerhalb des Bitschlupffensters gefunden, wird `approach` auf `0` gesetzt und das Attribut `BlockBitSlipCounter` inkrementiert. Am Ende der `do-while` Schleife wird jedes Mal geprüft, ob eine Ausgabe der Qualitätdaten erfolgen muss:

```
BlockBitErrorCounter += bit_error;
BlockFrameCounter++;

if(BlockFrameCounter == OutputBlockSize)
      output_on_screen_and_file(BLOCK);
```

Dazu wird die Anzahl an Bitfehlern und Rahmen aktualisiert und danach geprüft ob genug Rahmen gezählt wurden um eine Ausgabe zu erzeugen. Die `do-while` Schleife wird erst verlassen, wenn die Anzahl an unbekannten Bitmustern (`approach`) den Wert des Attributes `NumberOfUnknownPatternsToSearch` annimmt. In diesem Fall wird mittels der Methode `rescue_data_after_sync_lose_or_file_end` der Klasse `ct_MemoryAndFileManagment` die gesamte Anzahl an gefundenen Rahmen bytesynchronisiert und abgespeichert. Danach wird mit der Methode `set_index_char_to_long` das Attribut `IndexReadMemoryLong` der Klasse `ct_MemoryAndFileManagment` neu festgelegt.

- Mit der Methode `void open_output_file(char *writepath)` wird die Ausgabedatei erzeugt und geöffnet. Der Dateipfad wird von der Quelldatei `writepath` abgeleitet. Dies erfolgt indem der Dateityp am Ende des Dateipfades der Quelldatei mit der Endung „txt“ getauscht wird.

- Für die Ausgabe in die Datei und auf den Bildschirm dient die Methode `void output_on_screen_and_file(unsigned int mode)`. Mit dem Parameter `mode` wird eine von fünf Möglichkeiten angegeben, bei der eine Ausgabe erfolgen muss. Die Abarbeitung erfolgt innerhalb einer `switch-case` Anweisung anhand der folgenden symbolischen Konstanten: `PROGRAMSTART` = 0 erzeugt nach dem Start des Programms eine Ausgabe der gewählten Synchronisationsparameter, `ENDOFFILE` = 1 gibt zuerst eine Zeile mit den gesammelten Qualitätsdaten des aktuellen Blockes aus und danach die letzte Zeile mit allen bisher gesammelten Qualitätsdaten. `LOCKED` = 2 und `SEARCH` = 3 geben jeweils einen Zustandswechsel nach „Locked“ bzw. „Search“ an. Zusätzlich werden bei `SEARCH` die bisher im Block gesammelten Parameter ausgegeben. Mit `BLOCK` = 4 wird die reguläre Ausgabe einer Zeile nach Erreichen einer bestimmten Rahmenanzahl durchgeführt. Bei jeder Ausgabe der Qualitätsparameter werden zuvor diese Blockwerte auf die zugehörigen Gesamtzähler (Attribute) aufaddiert und nach der Ausgabe gelöscht.

- Die Methode `unsigned int search_in_bitslip_window(unsigned long value1, unsigned long value2)` ist aus der Umsetzung des Bitschlupffensters aus Abbildung 17 entstanden. Die Parameter `value1` und `value2` enthalten die beiden 4 Byte Blöcke, die durch die Methode `get_char_value_from_read_memory` übergeben wurden. Es stehen mehrere Rückgabewerte zur Verfügung. Der Rückgabewert `FOUNDNOTHING = 0xFF` signalisiert das kein Synchronwortbitmuster innerhalb des Fensters gefunden wurde. Um zu signalisieren das das Synchronwortbitmuster gefunden wurde, wird die Anzahl an unterschiedlichen Bitstellen und somit ein von `FOUNDNOTHING` unterschiedlicher Wert zurückgegeben. In einer `for` Schleife werden von `k=1` bis `BitslipWindowSize` alle Bitverschiebungen durchgetestet. Dafür werden die folgenden beiden Variablen verwendet:

```
shift_plus = bit_shift + k;
shift_minus = bit_shift - k;
```

  Die Variable `shift_plus` wird für den Test auf positiven Bitschlupf verwendet und die Variable `shift_minus` für den auf negativen. Die Variable `bit_shift` beinhaltet die aktuelle Bitverschiebung und `shift_minus` kann negative Werte annehmen. Anhand der zu testenden Bitverschiebung, muss beachtet werden in welchem Byte (`IndexReadMemoryChar`) begonnen wird. Ist diese größer gleich acht, erfolgt die Suche ein Byte später oder bei kleiner null ein Byte früher. Der Test auf das Synchronwort erfolgt folgendermaßen:

```
z_memory = (value1 << shift_plus) | (value2 >> (BIT_32 –
           shift_plus));
bit_error = co_CompareBitpattern.compare(z_memory);
if(bit_error <= BitErrorMax)
{
      synchronize_bit_slip(CURRENTBYTE, value1, value2,
                           shift_plus);
      return bit_error;
}
```

  Zuerst wird mit der Testbitverschiebung das zu untersuchende Bitmuster `z_memory` erstellt und mit dem des Synchronwortes verglichen. Ist der Test erfolgreich, wird mit der Methode

synchronize_bit_slip() die Klasse ct_MemoryAndFileManagment informiert und die Methode mit der Anzahl an unterschiedlichen Bits beendet.

4.6 Die Klasse ct_ReedSolomonCheck

Die Klasse besteht aus sechs Attributen und vier Methoden von denen jeweils nur eine als public deklariert ist. Des Weiteren wurden sieben Konstanten definiert: Die Symbollänge von acht Bits SYMBOLWIDTH = 8, die Länge des Synchronwortes in Byte ASMLENGTH = 4, die Codewortlänge CODEWORDLENGTH = 255, die Größe des Informationsteils in Byte INFORMATIONLENGTH = 223, die Anzahl an Paritätsstellen PARITYLENGTH = 32, die Größe des Galois-Feldes ARRAYSIZE = 256 und der Offset zu den Stützstellen im Galois-Feld OFFSET = 112.

Das erste Attribut hat den Namen p_PrimitivePolynomial und ist ein Feld der Länge SYMBOLWIDTH+1 vom Typ unsigned char und enthält das irreduzible Polynom. Das nächste Attribut enthält das Galois-Feld nach dessen Generierung und trägt den Namen p_PowerOfB und ist auch vom Typ unsigned char. Das Attribut p_IndexOfB ist ein Feld vom Typ int der Länge ARRAYSIZE und erwartet als Index ein Symbol und liefert den Index unter dem das Symbol im Galois-Feld zu finden ist. Der Zeiger auf einen Zeiger p_Blocks ist vom Typ unsigned char und enthält nach dem Konstrukoraufruf die Anzahl an Feldern der Länge CODEWORDLENGTH, die durch den Interleaving-Faktor gegenben ist. Der Interleaving-Faktor wird von der Rahmenlänge abgeleitet und im Attribut InterleavingDepth des Typs unsigned char abgelegt. Das letzte Attribut p_PseudoRandomSequence ist ein Zeiger auf ein Feld der Länge eines Rahmens ohne Synchronwort und enthält die Pseudo-Random Sequenz nach dessen Generierung.

- Der Konstruktor der Klasse bekommt als Parameter die Rahmenlänge übergeben und berechnet daraus den Interleaving-Faktor nach folgender Formel:

```
InterleavingDepth = (frame_length - ASMLENGTH) / CODEWORDLENGTH;
```

Danach wird der Speicher für die p_Blocks-Blöcke und die Pseudo-Random Sequenz angelegt. Nun wird noch das irreduzieble Polynom in das zugehörige Attribut kopiert und mit dem Aufruf der Methoden generate_galois_field() und generate_prs() das Galois-Feld und die Pseudo-Random Sequenz generiert. Der Destruktor gibt den angeforderten Speicherplatz frei.

- Die Methode void generate_galois_field(void) erzeugt das Galois-Feld und ist nach der Abbildung 20 implementiert worden. Das Galois-Feld wird nach der Erzeugung aus der herkömmlichen Darstellung in die Dual-Basis transformiert.

- Mit der Methode void generate_prs(void) wird die Pseudo-Random Sequenz erzeugt. Der erste Wert (p_PseudoRandomSequence[0]) ergibt sich durch die „all ones“ Initialisierung des Generators und ist 0xFF. Die weiteren Stellen ergeben sich durch den Generator:

```
for(k=1;k<frame_length;k++)
{
      for(l=0;l<8;l++)
      {
```

```
                bit8 = (pr_register >> 7) & 1;
                bit5 = (pr_register >> 4) & 1;
                bit3 = (pr_register >> 2) & 1;
                bit1 = pr_register & 1;

                pr_register = pr_register << 0x01;
                pr_register |= (bit1 ^ bit3 ^ bit5 ^ bit8);
        }
        p_PseudoRandomSequence[k] = pr_register;
    }
```

Es werden `frame_length` Bytes erzeugt und abgespeichert. Dazu wird nach jeweils acht Zyklen der Wert des Registers `pr_register` ausgelesen und abgespeichert. Die Variable `pr_register` vom Typ `unsigned char` ist vergleichbar mit einem rückgekoppelten Schieberegister. Mit jedem Zyklus wird das Register um ein Bit nach links geschoben und am Eingang d.h. der Stelle des LSB werden die Inhalte von Bit 1, Bit 3, Bit 5 und Bit 8 zurückgekoppelt und Exclusiv-Oder verknüpft.

- Die Überprüfung eines Reed-Solomon Codeblocks erfolgt mit der Methode `unsigned int decode_reed_solomon(register unsigned char *array_to_check)`. Mit dem Rückgabewert `1` wird signalisiert, dass der Codeblock `array_to_check` fehlerbehaftet ist und bei dem Rückgabewert `0` ist dieser fehlerfrei. Zuerst werden alle Elemente des Codeblocks in Indexform gebracht, weil zuerst immer eine Multiplikation zweier Elemente durchgeführt wird:

```
    for(i=0;i<CODEWORDLENGTH;i++)
        transformed_array[i] = p_IndexOfB[array_to_check[i]];
```

Das Feld `transformed_array` wurde lokal in der Methode angelegt, um die Elemente in `array_to_check` nicht zu verändern. Als Nächstes werden alle 32 Stützstellen in das Polynom eingesetzt:

```
for(i=OFFSET;i<=143;i++)
{
    syndrome = 0;
    for(j=0;j<CODEWORDLENGTH;j++)
    {
        if(transformed_array[j] != -1) // Wert != 00000000 (binaer)
            syndrome ^= p_PowerOfB[(transformed_array[j] + i*j) %
                        CODEWORDLENGTH];
    }
    if(syndrome != 0) // Wenn das Syndrome ungleich 0 ist, ist ein Fehler
                      aufgetreten
        syn_error = 1;
}
```

Die erste Stützstelle ist an der Stelle `OFFSET` = 112 im Galois-Feld enthalten und wird zuerst in das Polynom eingesetzt:

$$c(\beta^{112}) = a_0*(\beta^{112})^0 + a_1*(\beta^{112})^1 + a_2*(\beta^{112})^2 + \ldots + a_{253}*(\beta^{112})^{253} + a_{254}*(\beta^{112})^{254}$$

Das Ergebnis $c(\beta^{112})$ ist nach der Rechnung in der Variablen `syndrome` enthalten. Es wird am Anfang begonnen und zuerst die Multiplikation $a_x*((\beta^{112})^x)$ durchgeführt. Dazu wurde a_x bereits in Indexform gebracht und nun auf den Index 112*X aufaddiert. Für den Fall das dieser neu berechnete Index außerhalb des Galois-Feldes liegt, wird dieser noch modulo

`CODEWORDLENGTH` gerechnet. Danach wird das Element an der Stelle des neu berechneten Indexes geholt (`p_PowerOfB[Index]`) und auf `syndrome` aufaddiert (Exclusiv-Oder verknüpft). Dies wird nun bis zum Ende des Polynoms durchgeführt und am Ende der Inhalt von `syndrome` ausgewertet. Ist dieses nämlich ungleich Null, ist ein Fehler enthalten und `syn_error` wird auf `1` gesetzt. Nun wird die Rechnung auch mit den restlichen 31 Stützstellen durchgeführt und am Ende die Methode mit dem Rückgabewert `syn_error` beendet.

- Mit der Methode `unsigned long check_on_errors(register unsigned char *frames, register unsigned long number)` dient dazu die in dem Feld `frames` enthaltene Anzahl an `number` Rahmen auf Fehler hin zu untersuchen. Der Rückgabewert enthält die Anzahl an fehlerbehafteten Rahmen. Der Ablauf orientiert sich an der im Kap. 3.8 erarbeiteten Prozedur. Zuerst wird mit dem ersten Rahmen begonnen:

```
frames = &frames[ASMLENGTH];// Synchronwort auslassen
i=0;
for(z=0;z<CODEWORDLENGTH;z++)
{
      for(l=0;l<interleaving_depth;l++)
      {
            // Deinterleaving              Derandomizing
            p_Blocks[l][z] = *frames ^ p_PseudoRandomSequence[i];
            frames++;
            i++;
      }
}
```

Um das Synchronwort auszulassen, zeigt der Zeiger nun auf das Byte hinter dem Synchronwort. Danach wird der Rahmen auf die Interleaving-Blöcke aufgeteilt und gleichzeitig die statische Zufallsbitfolge von jedem Byte entfernt. Danach werden die Blöcke mit der Methode `decode_reed_solomon` untersucht. Ist einer der in den Rahmen enthaltenen Blöcke fehlerhaft, so wird die lokale Variable `all_frames` inkrementiert. Der Zeiger `frames` zeigt nun auf den Anfang des zweiten Rahmens und wird auch nach der obigen Prozedur untersucht. Dies wird bis zum letzten Rahmen (`number`) fortgesetzt und zum Schluss wird die Methode mit dem Rückgabewert `all_frames`, der für die Anzahl aller fehlerbehafteten Rahmen steht beendet.

5 Verifikation und Abschlusstest

5.1 Verifikation

Um die Funktionalität der Software nachzuweisen, wurden die in den folgenden vier Abschnitten beschriebenen Tests durchgeführt. Als Grundlage dient ein fehlerfreier und synchronisierter Datensatz, der für die unterschiedlichen Tests bearbeitet wurde.

5.1.1 Test des Reed-Solomon Checks

Um nachzuweisen, dass der Reed-Solomon Check richtig implementiert wurde, werden drei Tests mit den folgenden Parametern durchgeführt:

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bekannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\1.RS-Test\8errors_test.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

1. Fehlerfreier Datensatz

Zuerst wird das Programm mit dem fehlerfreien Datensatz als Quelldatei gestartet:

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked	255800000	200000	0	0	0	0.0e+000	0
Locked	511600000	200000	0	0	0	0.0e+000	0
Locked	767400000	200000	0	0	0	0.0e+000	0
Dateiende	999998940	181860	0	0	0	0.0e+000	0
Gesamt	999998940	781860	0	0	0	0.0e+000	0

Der Datensatz wird vom Programm als fehlerfrei angegeben, da in der Zeile R/S-Check keine fehlerhaften Rahmen aufgezählt wurden.

2. Vollständig fehlerbehafteter Datensatz

Für diesen Test wird eine Datei mit 781860 Rahmen generiert, bei denen der Rahmeninhalt durch Zufallswerte der Funktion `rand()` erzeugt wird. Nur die Synchronwörter sind fehlerfrei und im Rahmenabstand enthalten. Das Programm erzeugt die folgende Ausgabe:

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked	255800000	200000	200000	0	0	0.0e+000	0
Locked	511600000	200000	200000	0	0	0.0e+000	0
Locked	767400000	200000	200000	0	0	0.0e+000	0
Dateiende	999998940	181860	181860	0	0	0.0e+000	0
Gesamt	999998940	781860	781860	0	0	0.0e+000	0

Bei diesem fehlerbehafteten Datensatz wurden alle Rahmen als fehlerhaft erkannt.

3. Datensatz mit acht fehlerbehafteten Rahmen

Bei diesem Test werden in acht Rahmen des fehlerfreien Datensatzes Bitfehler nach der Tabelle 7 eingefügt. In einigen Rahmen werden auch mehrere Bytes geändert.

Rahmen	Bytepositon	Vorher(Wert)	Nachher(Wert)	Bitfehler
1	6	E	F1	8
1	168	41	40	1
48239	61696760	4	34	2
199999	255798411	8B	3	2
563915	721246010	D0	68	4
563915	721246017	BC	C1	6
625710	800282871	34	C4	4
721815	923200631	AD	AA	3
721815	923200634	46	B9	8
781859	999997660	36	17	4
781860	999998069	2B	D5	6
781860	999998070	F3	3	4
781860	999998071	E0	1E	7
781860	999998072	A1	54	5
781860	999998073	F	E	1

Tabelle 7: Systematisch in einen fehlerfreien Datensatz eingefügte Bitfehler

Das Programm findet alle acht fehlerbehafteten Rahmen:

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked	255800000	200000	3	0	0	0.0e+000	0
Locked	511600000	200000	0	0	0	0.0e+000	0
Locked	767400000	200000	1	0	0	0.0e+000	0
Dateiende	999998940	181860	4	0	0	0.0e+000	0
Gesamt	999998940	781860	8	0	0	0.0e+000	0

Die Verteilung der fehlerhaften Rahmen lässt sich durch die Tabelle 7 bestätigen. So sind z.B. unter den ersten 200000 Rahmen genau drei fehlerhafte enhalten.

5.1.2 „Search“ Test

Bei diesem Test wird die Bitmustersuche des Programms verifiziert. Es wird ein kompletter Datensatz aus Zufallswerten der Funktion `rand()` erzeugt. Danach werden an bestimmten Stellen in dieser Datei, Synchronworte nach dem in Tabelle 8 angegebenen Muster eingefügt.

Bytepositon	Bitverschiebung	Bitfehler
6998	1	4
9275	0	1
25927	3	3
180604	1	4
273026	6	4
291515	7	2
291744	7	4
288220	2	0
288229	6	0
288729	3	0
309204	5	3

Tabelle 8: Position, Bitverschiebung und Bitfehler der Synchronworte

Da es bei diesem Datensatz zu keiner Synchronisation kommen kann, muss für diesen Test die Methode `set_begining` der `ct_MemoryAndFileManagment` Klasse modifiziert werden. Dazu wird am Ende der Methode der folgende Quellcode eingefügt:

```
cout << "        |     ASM gefunden     |   " << setw(10) << getBytePosition()
<< "  |    " << setw(2) << bit_shift << "    |" << endl;
```

Mit dieser zusätzlichen Zeile gibt das Programm nach jedem gefundenen Synchronwort dessen Positon und Bitverschiebung aus. Durch Änderung des Eingabeparameters Zulaessige Bitfehler im ASM, werden nun die Fälle 0, 1, 2, 3 und 4 Bitfehler im Synchronwort getestet. Die Ergebnisse dieser vier Programmdurchläufe sind dem Anhang Kap. 9.1.1 beigefügt. Es wurde jedes Synchronwort an der entsprechenden Stelle mit der jeweiligen Bitverschiebung unter Beachtung des Parameters für die zulässigen Bitfehler gefunden. Zusätzlich wurden bei 3 und 4 zulässigen Bitfehlern im Synchronwort auch Pseudo-Synchronwörter gefunden. Wird dieser Wert weiter erhöht, steigt die Anzahl an gefundenen Pseudo-Synchronwörtern drastisch an.

5.1.3 „Locked“ Test

Dieser Test dient dazu, den Einfluss der Parameter Bekannte ASM in Folge bis Locked und Unbekannte ASM in Folge bis Search nachzuweisen. Dazu wurden einige Synchronwörter des fehlerfreien Datensatzes verfälscht.

ASM bis „Locked“	ASM bis „Search“	Rahmen	FW-Rahmen	Bitfehler
1	1	767645	10	108
1	3	767665	26	361
1	5	767678	37	542
2	5	767671	33	484
3	1	767640	8	84
3	3	767654	19	264
5	1	767626	5	39

Tabelle 9: Ergebnisse des Programms bei unterschiedlich gewählten Parametern

Die Tabelle 9 ist eine Zusammenstellung der wichtigsten Ergebnisse der Programmdurchläufe bei unterschiedlich gewählten Parametern. Die vollständigen Programmausgaben sind im Anhang Kap. 9.1.2 enthalten. Zusätzlich ist zu erwähnen, dass der Parameter Zulaessige Bitfehler im ASM auf 2 gesetzt wurde.

Der Tabelle kann entnommen werden, dass die Anzahl der gefundenen Rahmen, Flywheel-Rahmen und Bitfehler mit steigendem Parameter Unbekannte ASM in Folge bis Search zunimmt. Dies hängt damit zusammen, dass mehr fehlerhafte Synchronwörter (Rahmen) und die darin enthaltenen Bitfehler mitgezählt werden, bevor eine Resynchronisation durchgeführt wird. Mit steigendem Parameter Bekannte ASM in Folge bis Locked, nimmt diese Anzahl hingegen wieder ab. Dies ist auf das Muster des Datensatzes zurückzuführen, weil es erst zum Zustand „Search“ kommt, wenn ausreichend fehlerfreie Synchronwörter in Folge gefunden wurden. Daher werden weniger Rahmen gefunden und mitgezählt.

5.1.4 Test des Bitschlupffensters

Bei diesem Test wird dem fehlerfreien Datensatz Bitschlupf beigefügt um die Funktionsweise des Bitschlupffensters zu Testen. Die Größenordnung der Bitverschiebungen und Richtung ist in Tabelle 10 dargestellt. Im bearbeiteten Datensatz sind 781860 Rahmen enthalten.

Anzahl Rahmen	Bitverschiebung	Bitschlupftyp
148071	0	-
52497	8	Negativ
124370	6	Positiv
174500	4	Positiv
38999	1	Negativ
100012	5	Positiv
82570	3	Positiv
60841	1	Negativ

Tabelle 10: Muster des eingefügten Bitschlupfs

Die Parameter Bekannte ASM in Folge bis Locked, Unbekannte ASM in Folge bis Search und Zulaessige Bitfehler im ASM werden jeweils auf 2 gesetzt. Das Ergebnis des Tests mit unterschiedlichen Bitschlupffensterbreiten ist in Tabelle 11 angegeben.

Fensterbreite	Rahmen	RS-Check	FW-Rahmen	Bitfehler	Bitslip
1	781856	12	10	168	2
3	781857	11	8	136	3
4	781857	10	6	102	4
5	781858	9	4	68	5
6	781859	8	2	36	6
8	781860	7	0	0	7

Tabelle 11: Ergebnis des Programms bei unterschiedlichen Bitschlupffensterbreiten

Bei einer Fensterbreite von 8 Bit werden alle Rahmen gefunden und es sind nur die durch Bitschlupf veränderten Rahmen fehlerhaft. Bei kleineren Fensterbreiten, wird die Anzahl an Bitschlüpfen festgestellt, bei denen die Verschiebung (Tabelle 10) kleiner gleich der Fensterbreite ist. Mit jedem nichterkannten Bitschlupf, kommen zwei Flywheel-Rahmen hinzu, bevor die Resynchronisation begonnen wird. Bei diesem Vorgang geht ein Rahmen verloren. Die Anzahl an fehlerhaften Rahmen ergibt sich aus der Anzahl an erkannten Bitschlüpfen und der Anzahl an Flywheel-Rahmen. Die Flywheel-Rahmen wurden mit der falschen Bitverschiebung synchronisiert und sind daher fehlerhaft.

5.2 Abschlusstest

5.2.1 Anpassung der Software an das Datenformat des Datenrekorders

Bevor der Abschlusstest auf dem Datenrekorder durchgeführt werden kann, muss die Software noch an das in Kap. 2.5 angegebene Format angepasst werden. Dazu werden die folgenden Änderungen durchgeführt:

1. Das Attribut `MemoryWidth` wird in die beiden Attribute `ReadMemoryWidth` und `WriteMemoryWidth` aufgespalten. Die größe des Lesespeichers wird auf die Blockgröße des Datenrekorders von 4 MB minus die 32 Byte des Headers festgelegt. Die Schreibspeichergröße wird auf 5000 Rahmen begrenzt.

2. Die Methode `void open_file_to_read(void)` der `ct_MemoryAndFileManagment` Klasse wird am Ende durch die Zeile `_fseeki64(p_FileToRead, 65536, SEEK_SET)` ergänzt, um zu Beginn den Dataset Header auzulassen.

3. Bevor in der Methode `void read_frames_from_file(void)` der `ct_MemoryAndFileManagment` Klasse ein Block eingelesen wird, wird zuerst der Header des Blockes ausgelesen:

```
fread(buffer, ONEBYTE, BYTE_32, p_FileToRead);
```

Nachdem der Block eingelesen wurde, müssen noch die Bytes getauscht werden. Zusätzlich kommt ein neuer Eingabeparameter hinzu, mit dem der Datensatz zusätzliche invertiert werden kann:

```
if(Inverting)
{
      for(k=0;k<ReadMemoryWidth;k=k+4)
      {
            byte1 = ~(p_ReadMemory[k]);
            byte2 = ~(p_ReadMemory[k+ONEBYTE]);
            p_ReadMemory[k] = ~(p_ReadMemory[k+THREEBYTE]);
            p_ReadMemory[k+ONEBYTE] = ~(p_ReadMemory[k+TWOBYTE]);
            p_ReadMemory[k+TWOBYTE] = byte2;
            p_ReadMemory[k+THREEBYTE] = byte1;
      }
}
else
{
      for(k=0;k<ReadMemoryWidth;k=k+4)
      {
            byte1 = p_ReadMemory[k];
            byte2 = p_ReadMemory[k+ONEBYTE];
            p_ReadMemory[k] = p_ReadMemory[k+THREEBYTE];
            p_ReadMemory[k+ONEBYTE] = p_ReadMemory[k+TWOBYTE];
            p_ReadMemory[k+TWOBYTE] = byte2;
            p_ReadMemory[k+THREEBYTE] = byte1;
      }
}
```

In einer `for`-Schleife wird in 4 Byte Schritten durch den Lesespeicher gegangen und jeweils das erste Byte mit dem vierten und das zweite Byte mit dem dritten getauscht.

5.2.2 Bearbeitungs- und Geschwindigkeitstest

Als Abschlusstest wurde ein kompletter Satellitenpass von rund 20 GB Dateigröße mit dem Programm synchronisiert und ausgewertet. Es wurde jeweils die durchschnittliche Bearbeitungsgeschwindigkeit mit und ohne aktivierten Reed-Solomon Check bestimmt. Die beiden erzeugten Ausgaben des Programms sind im Anhang Kap. 9.1.4 dargestellt. Die gewählten Parameter waren die folgenden:

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	3
Unbekannte ASM in Folge bis Search	4
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Datensatzinvertierung	Ja
Reed-Solomon-Check	Ja

Quelle: d:\DataSet20.dat
Ziel : c:\tsx2.dat

Um die Performance zu erhöhen, wurde jeweils von der einen Festplatte gelesen und auf die andere geschrieben und es wurde die Optimierung des Compilers auf „Maximum Speed“ gesetzt.

Der Datensatz wurde vom Anfang bis zum Ende ohne Unterbrechungen synchronisiert. Nur am Ende wurde in den Zustand „Search“ übergegangen. Die Ursache dafür liegt am Empfang der Daten. Die Aufnahme wird solange fortgesetzt, bis das Signal des Satelliten im Rauschen verschwindet. Daher enthält der letzte Abschnitt der Datei keine Rahmen mehr. Ebenfalls ist die Qualität der Daten bei niedrigen Elevationen der Antenne zu erkennen. So ist die Qualität am Anfang noch schlecht und wird im Verlauf immer besser bis hin zu fehlerfrei empfangenen Rahmen. Am Ende wird die Qualität wieder schlechter bis hin zum Synchronausfall.

Die Bearbeitungsgeschwindigkeit betrug mit aktiviertem Reed-Solomon Check 7,5 MB/s und mit deaktiviertem Reed-Solomon Check 17,5 MB/s. Dennoch wurden Geschwindigkeiten von 30 MB/s auf dem Entwicklungssystem erzielt, womit die geforderte Geschwindigkeit von 20 MB/s erreicht wurde. Der Geschwindigkeitunterschied zwischen Entwicklungssystem und Rekorder kann an den unterschiedlichen Systemparametern liegen. So besitzt das Entwicklungssystem mehr Speicher und einen besseren Prozessor. Der Vorteil dieser Softwarelösung gegenüber dem verwendeten Hardwaresystem beim DLR ist der, dass ein Datensatz auf der Festplatte des Datenrekorders gespeichert wird und danach beliebig oft mit den unterschiedlichsten Synchronisationsparametern synchronisiert und analysiert werden kann.

6 Zusammenfassung und Ausblick

Die Aufgabe dieser Diplomarbeit bestand darin, ein Programm zu entwickeln, das eine Rahmensynchronisation an CCSDS-kompatiblen Satellitendaten auf einem digitalen Datenrekorder durchführt. Gleichzeitig sollten alle erfassbaren Qualitätsparameter ausgegeben werden. Dafür sollte auch die Reed-Solomon Codierung verwendet werden, welche in den Kapiteln 2.2 und 2.4 erläutert und in den Kapiteln 3.8 und 4.6 implementiert wurde. Zunächst wurden im zweiten Kapitel alle Grundlagen bezüglich der Übertragungsstrecke, Rahmensychronisation, CCSDS Standard und des digitalen Datenrekorders behandelt, um danach im dritten Kapitel mit der Konzeption der Arbeit zu beginnen. Hier wurde C++ als Programmiersprache festgelegt und die Schnittstellen des Programms definiert. Des Weiteren wurden erste Flussdiagramme erstellt, um die Bitmustersuche im Datenstrom und die Suche eines Bitmusters innerhalb eines Bitschlupffensters besser implementieren zu können. Es wurde auch ersichtlich, dass ein Dateimanagement notwendig wurde, welches als Schnittstelle zwischen Rahmensynchronisator und Festplatte dient. Dafür wurden ein Lesespeicher zum Einlesen der Daten und ein Schreibspeicher zum Abspeichern der Daten festgelegt. Dadurch können die gefundenen Rahmen bytesynchronisiert von dem Lesespeicher in den Schreibespeicher kopiert werden. Auch die Struktur des Rahmensynchronisators wurde hier festgelegt. Im vierten Kapitel wurden diese Konzepte in Form von Klassen und Methoden implementiert. Dazu wurden zu Beginn eine Namenskonvention und gewisse Regeln für eine performancebewusste Programmierung festgelegt. Es sind vier Klassen entstanden, die in diesem Kapitel mit Blick auf die Implementierung erläutert werden. Im fünften Kapitel wird beschrieben, wie das Programm getestet wurde, um die richtige Funktionsweise zu bestätigen und die Bearbeitungsgeschwindigkeit zu bestimmen.

Es konnte nachgewiesen werden, dass ein Telemetry-Datensatz anhand von in einer Datei abgelegten Parametern synchronisiert und entsprechend die synchronisierte Datei erzeugt wird. Während dieses Prozesses wird in bestimmten Abständen eine Ausgabe auf dem Bildschirm und in eine Datei getätigt, um den zeitlichen Verlauf darzustellen. Diese Ausgabe enthält zum Einen alle bei der Rahmensychronisation anfallenden Qualitätsparameter und zusätzlich kann ein Reed-Solomon Check durchgeführt werden, der die Anzahl an fehlerhaften Rahmen bestimmt. Die erzielte Bearbeitungsgeschwindigkeit des Datensatzes hing vom jeweiligen System ab. So wurden die spezifizierten 20 MB/s auf dem Entwicklungssystem erreicht. Auf dem Datenrekorder betrug die Geschwindigkeit 17,5 MB/s bei deaktiviertem Reed-Solomon Check und 7,5 MB/s bei aktiviertem Reed-Solomon Check. Der Geschwindigkeitsunterschied kann an der besseren Hardware des Entwicklungssystems gegenüber der des Datenrekorders liegen. Die Vorteile dieser Softwarelösung gegenüber einer vergleichbaren Hardwarelösung ist, dass die Geschwindigkeit maßgeblich von dem System bestimmt wird und die Software relativ einfach auf ein schnelleres System angepasst werden kann. Des Weiteren kann ein Datensatz, nachdem dieser auf der Festplatte des Datenrekorders abgespeichert wurde, beliebig oft mit den unterschiedlichsten Synchronisationsparametern synchronisiert und analysiert werden.

Für die Zukunft könnte für das Programm eine graphische Oberfläche erstellt werden, welche die Eingabe der Pfade zu den Dateien und die Erstellung der Parameterdatei erleichtert. Es ist auch denkbar, dass die in den Data-Set Headern des Datenrekorders enthaltenen Zeitinformationen genutzt werden, um Ereignisse anhand der realen Zeit zu spezifizieren. Dadurch können aufgetretene Probleme an der Empfangsanlage besser analysiert werden.

7 Literatur- /Linkverzeichnis

[1] Wiley J. Larson, James R. Wertz
Mircrocosm, Inc./Space Technology Series
SPACE MISSION ANALYSIS AND DESIGN, Second Edition

[2] European Space Agency(ESA)
ESA PSS-04-103 Issue 1
Telemetry Channel Coding Standard
September 1989

[3] http://www.fb9dv.uni-duisburg.de/vs/en/education/comNet2/Vorlesung/chapter4.pdf
Ohne Angabe des Verfasser
Telekommunikation
Erste Einsicht: 15.09.2007

[4] http://www.freepatentsonline.com
Rahmensynchronisierungsverfahren in einem Zeitmultiplexsystem
Erste Einsicht: 23.09.2007

[5] Recommendation for Space Data System Standards, CCSDS 101.0-B-6 BLUE BOOK
Telemetry Channel Coding
CCSDS Oktober 2002

[6] http://wwwcs.uni-paderborn.de/cs/ag-bloemer/lehre/ac2_SS2007/material/skript.pdf
Johannes Blömer
Algorithmische Codierungstheorie
Erste Einsicht: 14.10.2007
Sommersemester 2007

[7] DENNIS RODDY
Mcgraw-Hill Professional (Februar 2006)
SATELLITE COMMUNICATIONS
Fourth Edition

[8] Reach Technologies Inc.
Suite 103 1581H Hillside Avenue
Victoria,B.C.
Canada
Manual of the Digital Data Recorder

[9] Dietmar Deimling
Heise Heinz (1995)
C++ Tuning

8 Abbildungsverzeichnis

9 Anhang

9.1 Testergebnisse

9.1.1 „Search“ Test

0 zulässige Bitfehler im ASM

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	0
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: c:\Dokumente und Einstellungen\Christian\Desktop\bitsync_file.chr
Ziel : c:\Dokumente und Einstellungen\Christian\Desktop\test2.chr

	Byteposition	Verschiebung
ASM gefunden	288220	2
ASM gefunden	288229	6
ASM gefunden	288729	3
Dateiende	311990	----

1 zulässiger Bitfehler im ASM

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	1
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: c:\Dokumente und Einstellungen\Christian\Desktop\bitsync_file.chr
Ziel : c:\Dokumente und Einstellungen\Christian\Desktop\test2.chr

	Byteposition	Verschiebung
ASM gefunden	9275	0
ASM gefunden	288220	2
ASM gefunden	288229	6
ASM gefunden	288729	3
Dateiende	311990	----

2 zulässige Bitfehler im ASM

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: c:\Dokumente und Einstellungen\Christian\Desktop\bitsync_file.chr
Ziel : c:\Dokumente und Einstellungen\Christian\Desktop\test2.chr

	Byteposition	Verschiebung
ASM gefunden	9275	0
ASM gefunden	288220	2
ASM gefunden	288229	6
ASM gefunden	288729	3
ASM gefunden	291515	7
Dateiende	311990	----

3 zulässige Bitfehler im ASM

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	3
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: c:\Dokumente und Einstellungen\Christian\Desktop\bitsync_file.chr
Ziel : c:\Dokumente und Einstellungen\Christian\Desktop\test2.chr

	Byteposition	Verschiebung
ASM gefunden	9275	0
ASM gefunden	25927	3
ASM gefunden	145262	3
ASM gefunden	288220	2
ASM gefunden	288229	6
ASM gefunden	288729	3
ASM gefunden	291515	7
ASM gefunden	309204	5
Dateiende	311990	----

4 zulässige Bitfehler im ASM

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	4
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: c:\Dokumente und Einstellungen\Christian\Desktop\bitsync_file.chr
Ziel : c:\Dokumente und Einstellungen\Christian\Desktop\test2.chr

	Byteposition	Verschiebung
ASM gefunden	6998	1
ASM gefunden	9275	0
ASM gefunden	15266	2
ASM gefunden	25927	3
ASM gefunden	32516	4
ASM gefunden	50606	6
ASM gefunden	55316	5
ASM gefunden	61284	3
ASM gefunden	87453	5
ASM gefunden	88454	3
ASM gefunden	109087	1
ASM gefunden	145262	3
ASM gefunden	180604	1
ASM gefunden	273026	6
ASM gefunden	283678	7
ASM gefunden	288220	2
ASM gefunden	288229	6
ASM gefunden	288729	3
ASM gefunden	291515	7
ASM gefunden	291744	7
ASM gefunden	309204	5
Dateiende	311990	----

9.1.2 „Locked“ Test

1 Search-Frame/ 1 Flywheel-Frame

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	1
Unbekannte ASM in Folge bis Search	1
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	1351202	882	0	1	4	1.4e-004	0
Search -> Locked	1352481	------	-------	-------	---	-----	-----
Locked -> Search	6240819	3822	0	1	3	2.5e-005	0
Search -> Locked	6242098	------	-------	-------	---	-----	-----
Locked -> Search	10913006	3652	0	1	10	8.6e-005	0
Search -> Locked	10915564	------	-------	-------	---	-----	-----
Locked -> Search	12436295	1189	0	1	13	3.4e-004	0
Search -> Locked	15938197	------	-------	-------	---	-----	-----
Locked -> Search	15943313	4	0	1	16	1.3e-001	0
Search -> Locked	15972730	------	-------	-------	---	-----	-----
Locked -> Search	15979125	5	0	1	14	8.7e-002	0
Search -> Locked	17187780	------	-------	-------	---	-----	-----
Locked -> Search	17190338	2	0	1	15	2.3e-001	0
Search -> Locked	17196733	------	-------	-------	---	-----	-----
Locked -> Search	17203128	5	0	1	15	9.4e-002	0
Search -> Locked	28982718	------	-------	-------	---	-----	-----
Locked -> Search	28986555	3	0	1	9	9.4e-002	0
Search -> Locked	28991671	------	-------	-------	---	-----	-----
Locked -> Search	29004461	10	0	1	9	2.8e-002	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767645	0	10	108	4.4e-006	0

Zeit in Millisekunden: 33516
Zeit in Sekunden: 33
Bearbeitungsgeschwindigkeit: 29.84 MB/s

1 Search-Frame/ 3 Flywheel-Frames

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	1
Unbekannte ASM in Folge bis Search	3
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	12438853	9551	0	7	82	2.7e-004	0
Search -> Locked	15938197	------	-------	-------	---	-----	-----
Locked -> Search	15945871	6	0	3	52	2.7e-001	0
Search -> Locked	15972730	------	-------	-------	---	-----	-----
Locked -> Search	15981683	7	0	3	45	2.0e-001	0
Search -> Locked	17187780	------	-------	-------	---	-----	-----
Locked -> Search	17195454	6	0	4	58	3.0e-001	0
Search -> Locked	17196733	------	-------	-------	---	-----	-----
Locked -> Search	17205686	7	0	3	45	2.0e-001	0
Search -> Locked	28982718	------	-------	-------	---	-----	-----
Locked -> Search	28989113	5	0	3	39	2.4e-001	0
Search -> Locked	28991671	------	-------	-------	---	-----	-----
Locked -> Search	29007019	12	0	3	40	1.0e-001	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767665	0	26	361	1.5e-005	0

Zeit in Millisekunden: 22109
Zeit in Sekunden: 22
Bearbeitungsgeschwindigkeit: 45.23 MB/s

1 Search-Frame/ 5 Flywheel-Frames

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	1
Unbekannte ASM in Folge bis Search	5
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	12441411	9553	0	9	116	3.8e-004	0
Search -> Locked	15938197	------	-------	-------	---	-----	-----
Locked -> Search	15948429	8	0	5	86	3.4e-001	0
Search -> Locked	15972730	------	-------	-------	---	-----	-----
Locked -> Search	15984241	9	0	5	83	2.9e-001	0
Search -> Locked	17187780	------	-------	-------	---	-----	-----
Locked -> Search	17208244	16	0	9	132	2.6e-001	0
Search -> Locked	28982718	------	-------	-------	---	-----	-----
Locked -> Search	29009577	21	0	9	125	1.9e-001	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767678	0	37	542	2.2e-005	0

Zeit in Millisekunden: 21859
Zeit in Sekunden: 21
Bearbeitungsgeschwindigkeit: 45.75 MB/s

2 Search-Frames/ 5 Flywheel-Frames

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	5
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	12441411	9553	0	9	116	3.8e-004	0
Search -> Locked	15938197	------	-------	-------	---	-----	-----
Locked -> Search	15948429	8	0	5	86	3.4e-001	0
Search -> Locked	15972730	------	-------	-------	---	-----	-----
Locked -> Search	15984241	9	0	5	83	2.9e-001	0
Search -> Locked	17196733	------	-------	-------	---	-----	-----
Locked -> Search	17208244	9	0	5	74	2.6e-001	0
Search -> Locked	28982718	------	-------	-------	---	-----	-----
Locked -> Search	29009577	21	0	9	125	1.9e-001	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767671	0	33	484	2.0e-005	0

Zeit in Millisekunden: 21297
Zeit in Sekunden: 21
Bearbeitungsgeschwindigkeit: 46.95 MB/s

3 Search-Frames/ 1 Flywheel-Frame

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	3
Unbekannte ASM in Folge bis Search	1
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	1351202	882	0	1	4	1.4e-004	0
Search -> Locked	1352481	------	-------	-------	---	-----	-----
Locked -> Search	6240819	3822	0	1	3	2.5e-005	0
Search -> Locked	6242098	------	-------	-------	---	-----	-----
Locked -> Search	10913006	3652	0	1	10	8.6e-005	0
Search -> Locked	10915564	------	-------	-------	---	-----	-----
Locked -> Search	12436295	1189	0	1	13	3.4e-004	0
Search -> Locked	15938197	------	-------	-------	---	-----	-----
Locked -> Search	15943313	4	0	1	16	1.3e-001	0
Search -> Locked	15972730	------	-------	-------	---	-----	-----
Locked -> Search	15979125	5	0	1	14	8.7e-002	0
Search -> Locked	17196733	------	-------	-------	---	-----	-----
Locked -> Search	17203128	5	0	1	15	9.4e-002	0
Search -> Locked	28991671	------	-------	-------	---	-----	-----
Locked -> Search	29004461	10	0	1	9	2.8e-002	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767640	0	8	84	3.4e-006	0

Zeit in Millisekunden: 23500
Zeit in Sekunden: 23
Bearbeitungsgeschwindigkeit: 42.55 MB/s

3 Search-Frames/ 3 Flywheel-Frames

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	3
Unbekannte ASM in Folge bis Search	3
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	12438853	9551	0	7	82	2.7e-004	0
Search -> Locked	15938197	------	-------	-------	---	-----	-----
Locked -> Search	15945871	6	0	3	52	2.7e-001	0
Search -> Locked	15972730	------	-------	-------	---	-----	-----
Locked -> Search	15981683	7	0	3	45	2.0e-001	0
Search -> Locked	17196733	------	-------	-------	---	-----	-----
Locked -> Search	17205686	7	0	3	45	2.0e-001	0
Search -> Locked	28991671	------	-------	-------	---	-----	-----
Locked -> Search	29007019	12	0	3	40	1.0e-001	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767654	0	19	264	1.1e-005	0

Zeit in Millisekunden: 21515
Zeit in Sekunden: 21
Bearbeitungsgeschwindigkeit: 46.48 MB/s

5 Search-Frames/ 1 Flywheel-Frame

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	5
Unbekannte ASM in Folge bis Search	1
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\3.Sync-Test\Sync_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	223124	------	-------	-------	---	-----	-----
Locked -> Search	1351202	882	0	1	4	1.4e-004	0
Search -> Locked	1352481	------	-------	-------	---	-----	-----
Locked -> Search	6240819	3822	0	1	3	2.5e-005	0
Search -> Locked	6242098	------	-------	-------	---	-----	-----
Locked -> Search	10913006	3652	0	1	10	8.6e-005	0
Search -> Locked	10915564	------	-------	-------	---	-----	-----
Locked -> Search	12436295	1189	0	1	13	3.4e-004	0
Search -> Locked	28991671	------	-------	-------	---	-----	-----
Locked -> Search	29004461	10	0	1	9	2.8e-002	0
Search -> Locked	30425430	------	-------	-------	---	-----	-----
Locked	286225430	200000	0	0	0	0.0e+000	0
Locked	542025430	200000	0	0	0	0.0e+000	0
Locked	797825430	200000	0	0	0	0.0e+000	0
Dateiende	999998239	158071	0	0	0	0.0e+000	0
Gesamt	999998239	767626	0	5	39	1.6e-006	0

Zeit in Millisekunden: 21922
Zeit in Sekunden: 21
Bearbeitungsgeschwindigkeit: 45.62 MB/s

9.1.3 Test des Bitschlupffensters

Bitschlupffensterbreite 1

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\4.Bitslip-Test\bitslip_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked -> Search	189385367	148073	2	2	36	7.6e-006	0
Search -> Locked	189386645	------	-------	-------	---	-----	-----
Locked -> Search	256529029	52496	2	2	32	1.9e-005	0
Search -> Locked	256530308	------	-------	-------	---	-----	-----
Locked -> Search	415598259	124369	2	2	34	8.5e-006	0
Search -> Locked	415598260	------	-------	-------	---	-----	-----
Locked	671398260	200000	1	0	0	0.0e+000	1
Locked -> Search	688663481	13499	2	2	34	7.9e-005	0
Search -> Locked	688664760	------	-------	-------	---	-----	-----
Locked -> Search	816578829	100011	2	2	32	1.0e-005	0
Search -> Locked	816580109	------	-------	-------	---	-----	-----
Dateiende	999998941	143408	1	0	0	0.0e+000	1
Gesamt	999998941	781856	12	10	168	6.7e-006	2

Zeit in Millisekunden: 34593
Zeit in Sekunden: 34
Bearbeitungsgeschwindigkeit: 28.91 MB/s

Bitschlupffensterbreite 3

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	3
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\4.Bitslip-Test\bitslip_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked -> Search	189385367	148073	2	2	36	7.6e-006	0
Search -> Locked	189386645	------	-------	-------	---	-----	-----
Locked -> Search	256529029	52496	2	2	32	1.9e-005	0
Search -> Locked	256530308	------	-------	-------	---	-----	-----
Locked -> Search	415598259	124369	2	2	34	8.5e-006	0
Search -> Locked	415598260	------	-------	-------	---	-----	-----
Locked	671398260	200000	1	0	0	0.0e+000	1
Locked -> Search	688663481	13499	2	2	34	7.9e-005	0
Search -> Locked	688664760	------	-------	-------	---	-----	-----
Locked	944464761	200000	2	0	0	0.0e+000	2
Dateiende	999998941	43420	0	0	0	0.0e+000	0
Gesamt	999998941	781857	11	8	136	5.4e-006	3

Zeit in Millisekunden: 23640
Zeit in Sekunden: 23
Bearbeitungsgeschwindigkeit: 42.3 MB/s

Bitschlupffensterbreite 4

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	4
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\4.Bitslip-Test\bitslip_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked -> Search	189385367	148073	2	2	36	7.6e-006	0
Search -> Locked	189386645	------	-------	-------	---	-----	-----
Locked -> Search	256529029	52496	2	2	32	1.9e-005	0
Search -> Locked	256530308	------	-------	-------	---	-----	-----
Locked	512330309	200000	1	0	0	0.0e+000	1
Locked -> Search	688663481	137868	3	2	34	7.7e-006	1
Search -> Locked	688664760	------	-------	-------	---	-----	-----
Locked	944464761	200000	2	0	0	0.0e+000	2
Dateiende	999998941	43420	0	0	0	0.0e+000	0
Gesamt	999998941	781857	10	6	102	4.1e-006	4

Zeit in Millisekunden: 23750
Zeit in Sekunden: 23
Bearbeitungsgeschwindigkeit: 42.11 MB/s

Bitschlupffensterbreite 5

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	5
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\4.Bitslip-Test\bitslip_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked -> Search	189385367	148073	2	2	36	7.6e-006	0
Search -> Locked	189386645	------	-------	-------	---	-----	-----
Locked -> Search	256529029	52496	2	2	32	1.9e-005	0
Search -> Locked	256530308	------	-------	-------	---	-----	-----
Locked	512330309	200000	1	0	0	0.0e+000	1
Locked	768130309	200000	2	0	0	0.0e+000	2
Dateiende	999998941	181289	2	0	0	0.0e+000	2
Gesamt	999998941	781858	9	4	68	2.7e-006	5

Zeit in Millisekunden: 24187
Zeit in Sekunden: 24
Bearbeitungsgeschwindigkeit: 41.34 MB/s

Bitschlupffensterbreite 6

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	6
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\4.Bitslip-Test\bitslip_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked -> Search	189385367	148073	2	2	36	7.6e-006	0
Search -> Locked	189386645	------	-------	-------	---	-----	-----
Locked	445186646	200000	2	0	0	0.0e+000	2
Locked	700986646	200000	2	0	0	0.0e+000	2
Locked	956786647	200000	2	0	0	0.0e+000	2
Dateiende	999998941	33786	0	0	0	0.0e+000	0
Gesamt	999998941	781859	8	2	36	1.4e-006	6

Zeit in Millisekunden: 23469
Zeit in Sekunden: 23
Bearbeitungsgeschwindigkeit: 42.61 MB/s

Bitschlupffensterbreite 8

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	2
Unbekannte ASM in Folge bis Search	2
Bitschlupffensterbreite	8
Ausgabeblockgroesse	200000
Reed-Solomon-Check	Ja

Quelle: C:\Dokumente und Einstellungen\Christian\Desktop\4.Bitslip-Test\bitslip_file.chr
Ziel : C:\Dokumente und Einstellungen\Christian\Desktop\finale_test1GB.chr

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	0	------	-------	-------	---	-----	-----
Locked	281379999	200000	1	0	0	0.0e+000	1
Locked	511600000	200000	2	0	0	0.0e+000	2
Locked	767400000	200000	2	0	0	0.0e+000	2
Dateiende	999998941	181860	2	0	0	0.0e+000	2
Gesamt	999998941	781860	7	0	0	0.0e+000	7

Zeit in Millisekunden: 24406
Zeit in Sekunden: 24
Bearbeitungsgeschwindigkeit: 40.97 MB/s

9.1.4 Abschlusstest

9.1.4.a Ohne Reed-Solomon Check

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	3
Unbekannte ASM in Folge bis Search	4
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Datensatzinvertierung	Ja
Reed-Solomon-Check	Nein

Quelle: d:\DataSet20.dat
Ziel : c:\tsx.dat

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	1052	------	-------	-------	---	-----	-----
Locked	255801052	200000	0	22	1925	3.0e-004	0
Locked	511601052	200000	0	0	77	1.2e-005	0
Locked	767401052	200000	0	1	32	5.0e-006	0
Locked	1023201052	200000	0	0	19	3.0e-006	0
Locked	1279001052	200000	0	0	4	6.3e-007	0
Locked	1534801052	200000	0	0	0	0.0e+000	0
Locked	1790601052	200000	0	0	0	0.0e+000	0
Locked	2046401052	200000	0	0	0	0.0e+000	0
Locked	2302201052	200000	0	0	0	0.0e+000	0
Locked	2558001052	200000	0	0	0	0.0e+000	0
Locked	2813801052	200000	0	0	0	0.0e+000	0
Locked	3069601052	200000	0	0	0	0.0e+000	0
Locked	3325401052	200000	0	0	0	0.0e+000	0
Locked	3581201052	200000	0	0	0	0.0e+000	0
Locked	3837001052	200000	0	0	0	0.0e+000	0
Locked	4092801052	200000	0	0	0	0.0e+000	0
Locked	4348601052	200000	0	0	0	0.0e+000	0

Locked	4604401052	200000	0	0	0	0.0e+000	0	
Locked	4860201052	200000	0	0	0	0.0e+000	0	
Locked	5116001052	200000	0	0	0	0.0e+000	0	
Locked	5371801052	200000	0	0	0	0.0e+000	0	
Locked	5627601052	200000	0	0	0	0.0e+000	0	
Locked	5883401052	200000	0	0	0	0.0e+000	0	
Locked	6139201052	200000	0	0	0	0.0e+000	0	
Locked	6395001052	200000	0	0	0	0.0e+000	0	
Locked	6650801052	200000	0	0	0	0.0e+000	0	
Locked	6906601052	200000	0	0	0	0.0e+000	0	
Locked	7162401052	200000	0	0	0	0.0e+000	0	
Locked	7418201052	200000	0	0	0	0.0e+000	0	
Locked	7674001052	200000	0	0	0	0.0e+000	0	
Locked	7929801052	200000	0	0	0	0.0e+000	0	
Locked	8185601052	200000	0	0	0	0.0e+000	0	
Locked	8441401052	200000	0	0	0	0.0e+000	0	
Locked	8697201052	200000	0	0	0	0.0e+000	0	
Locked	8953001052	200000	0	0	0	0.0e+000	0	
Locked	9208801052	200000	0	0	0	0.0e+000	0	
Locked	9464601052	200000	0	0	0	0.0e+000	0	
Locked	9720401052	200000	0	0	0	0.0e+000	0	
Locked	9976201052	200000	0	0	0	0.0e+000	0	
Locked	10232001052	200000	0	0	0	0.0e+000	0	
Locked	10487801052	200000	0	0	0	0.0e+000	0	
Locked	10743601052	200000	0	0	0	0.0e+000	0	
Locked	10999401052	200000	0	0	0	0.0e+000	0	
Locked	11255201052	200000	0	0	0	0.0e+000	0	
Locked	11511001052	200000	0	0	0	0.0e+000	0	
Locked	11766801052	200000	0	0	0	0.0e+000	0	
Locked	12022601052	200000	0	0	0	0.0e+000	0	
Locked	12278401052	200000	0	0	0	0.0e+000	0	
Locked	12534201052	200000	0	0	0	0.0e+000	0	
Locked	12790001052	200000	0	0	0	0.0e+000	0	
Locked	13045801052	200000	0	0	0	0.0e+000	0	
Locked	13301601052	200000	0	0	0	0.0e+000	0	
Locked	13557401052	200000	0	0	0	0.0e+000	0	
Locked	13813201052	200000	0	0	0	0.0e+000	0	
Locked	14069001052	200000	0	0	0	0.0e+000	0	
Locked	14324801052	200000	0	0	0	0.0e+000	0	
Locked	14580601052	200000	0	0	0	0.0e+000	0	
Locked	14836401052	200000	0	0	0	0.0e+000	0	
Locked	15092201052	200000	0	0	0	0.0e+000	0	
Locked	15348001052	200000	0	0	0	0.0e+000	0	
Locked	15603801052	200000	0	0	0	0.0e+000	0	
Locked	15859601052	200000	0	0	0	0.0e+000	0	
Locked	16115401052	200000	0	0	0	0.0e+000	0	
Locked	16371201052	200000	0	0	0	0.0e+000	0	
Locked	16627001052	200000	0	0	0	0.0e+000	0	
Locked	16882801052	200000	0	0	0	0.0e+000	0	
Locked	17138601052	200000	0	0	0	0.0e+000	0	
Locked	17394401052	200000	0	0	0	0.0e+000	0	
Locked	17650201052	200000	0	0	0	0.0e+000	0	
Locked	17906001052	200000	0	0	0	0.0e+000	0	
Locked	18161801052	200000	0	0	0	0.0e+000	0	
Locked	18417601052	200000	0	0	0	0.0e+000	0	
Locked	18673401052	200000	0	0	0	0.0e+000	0	
Locked	18929201052	200000	0	0	0	0.0e+000	0	
Locked	19185001052	200000	0	0	0	0.0e+000	0	
Locked	19440801052	200000	0	0	0	0.0e+000	0	
Locked	19696601052	200000	0	0	0	0.0e+000	0	
Locked	19952401052	200000	0	0	4	6.3e-007	0	
Locked	20208201052	200000	0	0	2	3.1e-007	0	
Locked	20464001052	200000	0	0	2	3.1e-007	0	
Locked	20719801052	200000	0	0	6	9.4e-007	0	
Locked	20975601052	200000	0	0	16	2.5e-006	0	
Locked	21231401052	200000	0	0	19	3.0e-006	0	
Locked	21487201052	200000	0	0	39	6.1e-006	0	
Locked	21743001052	200000	0	0	80	1.3e-005	0	
Locked	21998801052	200000	0	0	332	5.2e-005	0	
Locked -> Search	22062075740	49472	0	4	212	1.3e-004	0	
Dateiende	22355469760	0	0	0	0	0.0e+000	0	
Gesamt	22355469760	17249472	0	27	2769	5.0e-006	0	

Zeit in Millisekunden: 1277781
Zeit in Sekunden: 1277
Bearbeitungsgeschwindigkeit: 17.5 MB/s

9.1.4.b Mit Reed-Solomon Check

Parameter	Wert
Attached Sync Marker	1acffc1d
Rahmenlaenge	1279
Zulaessige Bitfehler im ASM	2
Bakannte ASM in Folge bis Locked	3
Unbekannte ASM in Folge bis Search	4
Bitschlupffensterbreite	1
Ausgabeblockgroesse	200000
Datensatzinvertierung	Ja
Reed-Solomon-Check	Ja

Quelle: d:\DataSet20.dat
Ziel : c:\tsx2.dat

Zustand	Byteposition	Rahmen	R/S-Check	FW-Rahmen	ASM	BER-ASM	Bitslip
Search	0	------	-------	-------	---	-----	-----
Search -> Locked	1052	------	-------	-------	---	-----	-----
Locked	255801052	200000	71695	22	1925	3.0e-004	0
Locked	511601052	200000	14704	0	77	1.2e-005	0
Locked	767401052	200000	3642	1	32	5.0e-006	0
Locked	1023201052	200000	1897	0	19	3.0e-006	0
Locked	1279001052	200000	792	0	4	6.3e-007	0
Locked	1534801052	200000	411	0	0	0.0e+000	0
Locked	1790601052	200000	189	0	0	0.0e+000	0
Locked	2046401052	200000	100	0	0	0.0e+000	0
Locked	2302201052	200000	66	0	0	0.0e+000	0
Locked	2558001052	200000	20	0	0	0.0e+000	0
Locked	2813801052	200000	5	0	0	0.0e+000	0
Locked	3069601052	200000	8	0	0	0.0e+000	0
Locked	3325401052	200000	0	0	0	0.0e+000	0
Locked	3581201052	200000	1	0	0	0.0e+000	0
Locked	3837001052	200000	0	0	0	0.0e+000	0
Locked	4092801052	200000	0	0	0	0.0e+000	0
Locked	4348601052	200000	0	0	0	0.0e+000	0
Locked	4604401052	200000	0	0	0	0.0e+000	0
Locked	4860201052	200000	0	0	0	0.0e+000	0
Locked	5116001052	200000	0	0	0	0.0e+000	0
Locked	5371801052	200000	0	0	0	0.0e+000	0
Locked	5627601052	200000	0	0	0	0.0e+000	0
Locked	5883401052	200000	0	0	0	0.0e+000	0
Locked	6139201052	200000	0	0	0	0.0e+000	0
Locked	6395001052	200000	0	0	0	0.0e+000	0
Locked	6650801052	200000	0	0	0	0.0e+000	0
Locked	6906601052	200000	0	0	0	0.0e+000	0
Locked	7162401052	200000	0	0	0	0.0e+000	0
Locked	7418201052	200000	0	0	0	0.0e+000	0
Locked	7674001052	200000	0	0	0	0.0e+000	0
Locked	7929801052	200000	0	0	0	0.0e+000	0
Locked	8185601052	200000	0	0	0	0.0e+000	0
Locked	8441401052	200000	0	0	0	0.0e+000	0
Locked	8697201052	200000	0	0	0	0.0e+000	0
Locked	8953001052	200000	0	0	0	0.0e+000	0
Locked	9208801052	200000	0	0	0	0.0e+000	0
Locked	9464601052	200000	0	0	0	0.0e+000	0
Locked	9720401052	200000	0	0	0	0.0e+000	0
Locked	9976201052	200000	0	0	0	0.0e+000	0
Locked	10232001052	200000	0	0	0	0.0e+000	0
Locked	10487801052	200000	0	0	0	0.0e+000	0
Locked	10743601052	200000	0	0	0	0.0e+000	0
Locked	10999401052	200000	0	0	0	0.0e+000	0
Locked	11255201052	200000	0	0	0	0.0e+000	0
Locked	11511001052	200000	0	0	0	0.0e+000	0
Locked	11766801052	200000	0	0	0	0.0e+000	0
Locked	12022601052	200000	0	0	0	0.0e+000	0
Locked	12278401052	200000	0	0	0	0.0e+000	0
Locked	12534201052	200000	0	0	0	0.0e+000	0
Locked	12790001052	200000	0	0	0	0.0e+000	0
Locked	13045801052	200000	0	0	0	0.0e+000	0
Locked	13301601052	200000	0	0	0	0.0e+000	0
Locked	13557401052	200000	0	0	0	0.0e+000	0

Locked	13813201052	200000	0	0	0	0.0e+000	0	
Locked	14069001052	200000	0	0	0	0.0e+000	0	
Locked	14324801052	200000	0	0	0	0.0e+000	0	
Locked	14580601052	200000	0	0	0	0.0e+000	0	
Locked	14836401052	200000	0	0	0	0.0e+000	0	
Locked	15092201052	200000	0	0	0	0.0e+000	0	
Locked	15348001052	200000	0	0	0	0.0e+000	0	
Locked	15603801052	200000	0	0	0	0.0e+000	0	
Locked	15859601052	200000	0	0	0	0.0e+000	0	
Locked	16115401052	200000	0	0	0	0.0e+000	0	
Locked	16371201052	200000	0	0	0	0.0e+000	0	
Locked	16627001052	200000	0	0	0	0.0e+000	0	
Locked	16882801052	200000	0	0	0	0.0e+000	0	
Locked	17138601052	200000	0	0	0	0.0e+000	0	
Locked	17394401052	200000	0	0	0	0.0e+000	0	
Locked	17650201052	200000	0	0	0	0.0e+000	0	
Locked	17906001052	200000	0	0	0	0.0e+000	0	
Locked	18161801052	200000	1	0	0	0.0e+000	0	
Locked	18417601052	200000	0	0	0	0.0e+000	0	
Locked	18673401052	200000	1	0	0	0.0e+000	0	
Locked	18929201052	200000	3	0	0	0.0e+000	0	
Locked	19185001052	200000	8	0	0	0.0e+000	0	
Locked	19440801052	200000	14	0	0	0.0e+000	0	
Locked	19696601052	200000	34	0	0	0.0e+000	0	
Locked	19952401052	200000	72	0	4	6.3e-007	0	
Locked	20208201052	200000	163	0	2	3.1e-007	0	
Locked	20464001052	200000	341	0	2	3.1e-007	0	
Locked	20719801052	200000	528	0	6	9.4e-007	0	
Locked	20975601052	200000	1192	0	16	2.5e-006	0	
Locked	21231401052	200000	2154	0	19	3.0e-006	0	
Locked	21487201052	200000	5109	0	39	6.1e-006	0	
Locked	21743001052	200000	10569	0	80	1.3e-005	0	
Locked	21998801052	200000	31431	0	332	5.2e-005	0	
Locked -> Search	22062075740	49472	18576	4	212	1.3e-004	0	
Dateiende	22355469760	0	0	0	0	0.0e+000	0	
Gesamt	22355469760	17249472	163726	27	2769	5.0e-006	0	

Zeit in Millisekunden: 2985985
Zeit in Sekunden: 2985
Bearbeitungsgeschwindigkeit: 7.487 MB/s

9.2 Quelltext des Programms

9.2.1 main.cpp

```
#include "frame_synchronizer.h"
#include <ctime>

bool transform_sync_marker(char* char_value, unsigned long &hex_value);

int main(int argc, char *argv[])
{
	clock_t time1, time2;
	FILE *p_argument_file = NULL;
	unsigned long sync_marker;
	unsigned int bit_errors_asm;
	unsigned int frame_length;
	unsigned int search_frames;
	unsigned int fly_wheel_frames;
	unsigned int bit_slip_window;
	unsigned int reed_solomon_check;
	unsigned long block_length;
	bool invert;
	char line[30] = {0};
	char buffer[20] = {0};
	char name_buffer[9][20] = {{"AttachedSyncMarker"}, {"FrameLength"},
	 {"AllowedASMErrors"}, "SearchFrames", {"FlywheelFrames"}, {"Bitslip"}, {"RS-Check"},
	 {"OutputBlock"}, {"Invert"}};
	char value_buffer[9][20] = {{0}};
	unsigned int values[9] = {18, 11, 16, 12, 14, 7, 8, 11, 6};//Laenge der Namen
	unsigned int status, k;
	bool end_of_file, transformation;

	if(argc != 4)
	{
		cout << "Falsche Anzahl an Argumenten!" << endl;
		cout << "1. Dateipfad zur Parameterdatei" << endl;
		cout << "2. Dateipfad der Quelldatei" << endl;
		cout << "3. Dateipfad der Zieldatei" << endl;
		cout << "Bei Leerzeichen in Dateipfaden den gesamten Dateipfad in \" \"
			 schreiben!" << endl;
		exit(0);
	}
	else
	{
		fopen_s(&p_argument_file, argv[1], "r+t");
		if(p_argument_file == NULL)
		{
			cout << "Parameterdatei konnte nicht geoeffnet werden!" << endl;
			exit(0);
		}
	}

	for(k=0;k<9;k++)
	{
		status = fscanf(p_argument_file,"%s", line);
		if(status == EOF)
		{
			cout << "Zu wenig Parameter in der Datei!" << endl;
			exit(0);
		}
		strncpy(buffer, line, values[k]);
		buffer[values[k]]= '\0';
		if(!strcmp(buffer,name_buffer[k]))
		{
			strcpy(value_buffer[k], &line[values[k]+1]);
		}
		else
		{
			cout << name_buffer[k] << " falsch geschrieben!" << endl;
			exit(0);
		}
```

```
    }
    fclose(p_argument_file);
//Parameter überprüfen und kopieren
    transformation = transform_sync_marker(value_buffer[0], sync_marker);
    if(!transformation)
    {
        cout << "AttachedSyncMarker wurde falsch eingegeben!" << endl;
        exit(0);
    }
    frame_length = atoi(value_buffer[1]);
    bit_errors_asm = atoi(value_buffer[2]);
    if(!((bit_errors_asm >= ZEROBIT) && (bit_errors_asm <= BIT_32)))
    {
        cout << "Parameter AllowedASMErrors unzulaessig! (0 bis 32)" << endl;
        exit(0);
    }
    search_frames = atoi(value_buffer[3]);
    if(search_frames < 1)
    {
        cout << "SearchFrames unzulaessig! (groesser 0)" << endl;
        exit(0);
    }
    fly_wheel_frames = atoi(value_buffer[4]);
    if(fly_wheel_frames < 1)
    {
        cout << "FlywheelFrames unzulaessig! (groesser 0)" << endl;
        exit(0);
    }
    bit_slip_window = atoi(value_buffer[5]);
    if(!((bit_slip_window >= ZEROBIT) && (bit_slip_window <= EIGHTBIT)))
    {
        cout << "Parameter Bitslip unzulaessig! (0 bis 8)" << endl;
        exit(0);
    }
    reed_solomon_check = atoi(value_buffer[6]);
    if((reed_solomon_check < 0) || (reed_solomon_check > 1))
    {
        cout << "Parameter RS-Check ist unzulaessig! (0 oder 1)" << endl;
        exit(0);
    }
    block_length = atol(value_buffer[7]);
    if((atoi(value_buffer[8])) == 0)
        invert = false;
    else
        invert = true;

    ct_FrameSynchronizer test(
        sync_marker,          //Synchronwort (ASM)
        bit_errors_asm,       //zulaessige Bitfehler im ASM um als erkannt zu gelten
        frame_length,         //Rahmenlaenge
        search_frames,        //bekannte ASM in Folge bis Locked
        fly_wheel_frames,     //Unbekannte ASM in Folge bis Search
        bit_slip_window,      //Bitschlupffensterbreite (nach rechts und links)
        argv[2],              //Quelledatei
        argv[3],              //Zieldatei
        reed_solomon_check,   //Reed Solomon Check durchfuehren
        block_length,         //Ausgabeblockgroesse
        invert);              //Invertierung
    test.output_on_screen_and_file(PROGRAMSTART);
    time1 = clock();
    test.open_file_to_read();

    while(1)
    {
        end_of_file = test.search_mode();
        if(end_of_file)//Dateiende erreicht
        {
            test.output_on_screen_and_file(ENDOFFILE);
            break;
        }
        else
        {   // Synchronisation wurde hergestellt
            test.output_on_screen_and_file(LOCKED);
        }

        end_of_file = test.locked_mode();
        if(end_of_file)//true Dateiende erreicht
        {
```

```
                        test.output_on_screen_and_file(ENDOFFILE);
                        break;
                }
                else
                {       // Synchronisation ging verloren
                        test.output_on_screen_and_file(SEARCH);
                }
        }
        time2 = clock();
        cout << endl << "Zeit in Millisekunden: " << time2-time1 << endl;
        cout << "Zeit in Sekunden: " << (time2-time1)/CLOCKS_PER_SEC << endl;
        cout << "Bearbeitungsgeschwindigkeit: " << setprecision(4);
        cout << resetiosflags(ios_base::scientific);
        cout <<(double)((test.getFilePosition()*CLOCKS_PER_SEC/(time2-time1))/1e6);
        cout << " MB/s" << endl;

return 0;
}

bool transform_sync_marker(char* char_value, unsigned long &hex_value)
{
        unsigned long value = 0;
        unsigned int k;
        size_t string_length;

        string_length = strlen(char_value);
        if((string_length < 9) || (string_length > 10))
                return false;

        if(char_value[0] == '0' && char_value[1] == 'x')
        {
                if(string_length == 10)
                {
                        for(k=2;k<10;k++)
                        {
                                value <<= 4;

                                switch(char_value[k])
                                {
                                        case '0': value |= 0x0;
                                                break;
                                        case '1': value |= 0x1;
                                                break;
                                        case '2': value |= 0x2;
                                                break;
                                        case '3': value |= 0x3;
                                                break;
                                        case '4': value |= 0x4;
                                                break;
                                        case '5': value |= 0x5;
                                                break;
                                        case '6': value |= 0x6;
                                                break;
                                        case '7': value |= 0x7;
                                                break;
                                        case '8': value |= 0x8;
                                                break;
                                        case '9': value |= 0x9;
                                                break;
                                        case 'A': value |= 0xA;
                                                break;
                                        case 'B': value |= 0xB;
                                                break;
                                        case 'C': value |= 0xC;
                                                break;
                                        case 'D': value |= 0xD;
                                                break;
                                        case 'E': value |= 0xE;
                                                break;
                                        case 'F': value |= 0xF;
                                }
                        }
                        hex_value = value;
                        return true;
                }
                else
                        return false;
        }
```

```
        else
        {
                if(string_length == 9)
                {
                        hex_value = atol(char_value);
                        return true;
                }
                else
                        return false;
        }
}
```

9.2.2 frame_synchronizer.h

```
#ifndef __FRAME_SYNCHRONIZER_H__
        #define __FRAME_SYNCHRONIZER_H__

        #include "reed_solomon_check.h"
        #include <fstream>
        #include <iostream>
        #include <iomanip>
        #include <climits>
        #include <cstring>
        using namespace std;

        #define FACTOR                      5000
        #define ZEROBYTE                    0
        #define ONEBYTE                     1
        #define TWOBYTE                     2
        #define THREEBYTE                   3
        #define FOURBYTE                    4
        #define FIVEBYTE                    5
        #define SIXBYTE                     6
        #define SEVENBYTE                   7
        #define EIGHTBYTE                   8
        #define ZEROBIT                     0
        #define ONEBIT                      1
        #define TWOBIT                      2
        #define THREEBIT                    3
        #define FOURBIT                     4
        #define FIVEBIT                     5
        #define SIXBIT                      6
        #define SEVENBIT                    7
        #define EIGHTBIT                    8
        #define BIT_16                      16
        #define BIT_24                      24
        #define BIT_32                      32
        #define FOUNDNOTHING                0xFF
        #define CURRENTBYTE                 0
        #define NEXTBYTE                    1
        #define LASTBYTE                    2
        #define SYNC                        0
        #define ENDOFFILE                   1
        #define NOSYNC                      2
        #define PROGRAMSTART                0
        #define LOCKED                      2
        #define SEARCH                      3
        #define BLOCK                       4
        #define FOURMEGABYTES               4194272 //(4*1024*1024 - 32Byte Header)
        #define BYTE_32                     32

        class ct_CompareBitpattern
        {
                unsigned long Mask;
                unsigned char *p_LookUpTable1;
                unsigned char *p_LookUpTable2;
                void initLook_up_tables(void);//Tabellen initialisieren

        public:
                ct_CompareBitpattern(unsigned long pattern) :
                Mask(pattern), p_LookUpTable1(NULL), p_LookUpTable2(NULL)
                {initLook_up_tables();}
                ~ct_CompareBitpattern();
                //neue Maske setzen und Tabellen neu berechnen
                inline void setMask(unsigned long pattern)
                {Mask = pattern; initLook_up_tables();}
```

```
        inline unsigned long getMask(void) {return Mask;}
        inline unsigned int compare(unsigned long bit_pattern)//Bitmuster vergleichen
        {return p_LookUpTable1[(bit_pattern>>BIT_16)]
               + p_LookUpTable2[bit_pattern & 0xFFFF];}
};

class ct_MemoryAndFileManagment : public ct_ReedSolomonCheck
{       //Parameter
        char *p_ReadPath; //Pfad zum Einlesen
        char *p_WritePath; //Pfad zum Schreiben
        bool EnableReedSolomonCheck;
        bool Inverting;
        /****************/
        FILE *p_FileToRead; //Dateipointer zum Einlesen
        FILE *p_FileToWrite; //Dateipointer zum Schreiben
        unsigned long ReadMemoryWidth; //Groesse des Lesepeichers
        unsigned long WriteMemoryWidth; //Groesse des Schreibspeichers
        unsigned long FrameNumber; //Anzahl an Rahmen, die in den Speicher passen
        //Zaehlt die Anzahl an Rahmen im Speicher
        unsigned long FreeSpaceOnWriteMemory;//Angabe in Rahmen
        unsigned char *p_ReadMemory; //Zeiger auf den Speicher zum Lesen
        unsigned char *p_WriteMemory; //Zeiger auf den Speicher zum Schreiben
        unsigned char BufferWriteMemory;
        unsigned long BufferReadMemory;
        unsigned long IndexWriteMemory;
        unsigned long IndexOnIntArray;//IndexReadMemoryLong
        unsigned long IndexOnCharArray;//IndexReadMemoryChar
        unsigned long MaxIndexInt; //MaxIndexLong
        unsigned long MaxIndexChar;
        unsigned long MaxIndexCharOld;
        unsigned long FramesWithErrors;
        unsigned long AllFramesWithErrors;
        unsigned int BitShift; //Bitverschiebung zum Bytesynchronisieren
        unsigned long BeginToCopy; //Index ab dem weiter synchronisiert wird
        __int64 FilePosition;
        bool ReadStatus;     // Wurde schon eingelesen?
        bool FrameAtFileEnd; // Existiert ein Rahmen am Dateiende, der nicht
        mitgezaehlt wurde
        inline void read_frames_from_file(void);//liesst neuen Datenblock ein
        inline void write_synchronized_bytes_to_file(void);
        inline void write_synchronized_bytes_to_file(unsigned long number_of_bytes);
        inline void decreaseFreeSpaceOnWriteMemory(void);//Um eins erniedrigen
        inline void aktualize_FilePosition(void){FilePosition += MaxIndexChar;}
        inline void IndexOnCharArray_plus_one(void);
        inline void IndexOnCharArray_minus_one(void);
        void synchronize_bytes(unsigned long number_of_bytes);
        void synchronize_bytes(void);//synchronisiert bis zum Ende des Lese-Speichers
        void open_file_to_write(void);
public:
        ct_MemoryAndFileManagment(unsigned int frame_length, char *readpath,
                        char *writepath, unsigned int reed_solomon_check, bool invert);
        ~ct_MemoryAndFileManagment();
        inline char* getp_ReadPath(void) {return p_ReadPath;}
        inline char* getp_WritePath(void) {return p_WritePath;}
        inline unsigned long getIndexOnIntArray(void) {return IndexOnIntArray;}
        inline unsigned long getIndexOnCharArray(void) {return IndexOnCharArray;}
        inline unsigned long getBufferReadMemory(void) {return BufferReadMemory;}
        inline bool getEnableReedSolomonCheck(void) {return EnableReedSolomonCheck;}
        inline bool getInverting(void) {return Inverting;}
        inline unsigned int getBitShift(void) {return BitShift;}
        unsigned long get_and_reset_FramesWithErrors(void);
        inline unsigned long getAllFramesWithErrors(void) {return AllFramesWithErrors;}
        inline bool getFrameAtFileEnd(void) {return FrameAtFileEnd;}
        inline bool getReadStatus(void){return ReadStatus;}
        inline __int64 getFilePosition(void) {return FilePosition;}
        inline __int64 getBytePosition(void);
        inline bool get_next_long_value_from_read_memory(unsigned long &value);
        inline unsigned long get_long_value_from_read_memory(unsigned long index_char);
        inline bool get_char_value_from_read_memory(unsigned int byte_offset,
                                        unsigned long &value1, unsigned long &value2);
        inline void setIndexOnIntArray(unsigned long index) {IndexOnIntArray = index;}
        inline void setBufferReadMemory(unsigned long value){BufferReadMemory = value;}
        inline void set_index_char_to_int(void);//konvertieren von char nach int
        inline void set_begining(unsigned int byte, unsigned int bit_shift);
        inline void reset_write_memory_options(void);
        void open_file_to_read(void);
        void rescue_data_after_sync_lose_or_file_end(unsigned int frame_length,
                                                     bool end_of_file);
```

```
                void synchronize_bit_slip(unsigned int mode, unsigned long value1,
                                          unsigned long value2, unsigned int new_bit_shift);
        };


        class ct_FrameSynchronizer : public ct_MemoryAndFileManagment
        {
                /***********************Parameter**************************/
                unsigned int FrameLength; //Rahmenlaenge
                unsigned int BitErrorMax; //zulaessige Bitfehler im Synchronwort
                unsigned int BitSlipWindowSize; //groesse des Fensters * 2
//Anzahl an erkannten Synchronwoertern in Folge um von Search in Locked - Mode zu gelangen
                unsigned int NumberOfKnownPatternsToLock;
//Anzahl an nicht erkannten Synchronwoertern in Folge um von Locked in Search-Mode zu gelangen
                unsigned int NumberOfUnknownPatternsToSearch;
                unsigned long OutputBlockSize; //Groesse der Ausgabebloecke in Anzahl Rahmen
                /*********************************************************/
                ofstream *p_OutputFile;
                unsigned long BitErrorCounter;    //Zaehlt die Bitfehler der Synchronwoerter
                unsigned long BlockBitErrorCounter;
                unsigned long FrameCounter;       //Zaehlt die Anzahl an Rahmen
                unsigned long BlockFrameCounter;
                unsigned long BitSlipCounter;     //Zaehlt das auftreten von Bitschlupf
                unsigned long BlockBitSlipCounter;
                unsigned long FlyWheelFrames;//Zaehlt die Anzahl von nicht erkannten ASM
                unsigned long BlockFlyWheelFrames;

                bool search_bitpattern(void); //sucht das Synchronwort in der Datei
                unsigned int check_next_pattern(void); //untersucht die naechsten Synchronworte
                unsigned int search_in_bitslip_window(unsigned long value1,
                                                      unsigned long value2);
                void open_output_file(char *writepath);
        public:
                ct_FrameSynchronizer(unsigned long pattern, unsigned int biterror,
                unsigned int frame_length, unsigned int n_to_lock,unsigned int n_to_search,
                unsigned int window_size, char *readpath, char *writepath,
                unsigned int rs_check, unsigned long output_block_size, bool invert) :
                BitErrorCounter(0), FrameLength(frame_length), BitErrorMax(biterror),
          NumberOfKnownPatternsToLock(n_to_lock), NumberOfUnknownPatternsToSearch(n_to_search),
               BitSlipWindowSize(window_size), OutputBlockSize(output_block_size),
               co_CompareBitpattern(pattern), ct_MemoryAndFileManagment(frame_length,
              readpath, writepath, rs_check, invert){BitErrorCounter = 0; FrameCounter = 0;
              BitSlipCounter = 0; FlyWheelFrames = 0; BlockBitSlipCounter = 0;
               BlockFlyWheelFrames = 0; open_output_file(writepath);}
               ~ct_FrameSynchronizer() {fclose(p_OutputFile);}

                ct_CompareBitpattern co_CompareBitpattern;
                unsigned long getFrameCounter(void) {return FrameCounter;}
                unsigned long getBitErrorCounter(void) {return BitErrorCounter;}
                unsigned long getBitSlipCounter(void) {return BitSlipCounter;}
                unsigned long getFlyWheelFrames(void) {return FlyWheelFrames;}
                unsigned int getFrameLength(void) {return FrameLength;}
                unsigned int getNumberOfUnknownPatternsToLock(void)
                {return NumberOfKnownPatternsToLock;}
                inline void output_on_screen_and_file(unsigned int mode);
                bool locked_mode(void);
                bool search_mode(void);
        };

        /***********************Inline Methoden*******************************************/

inline void ct_MemoryAndFileManagment::read_frames_from_file(void)
{
        unsigned char buffer[BYTE_32];
        unsigned char byte1, byte2;
        unsigned long k;

        //Header vor jedem Block auslesen
        fread(buffer, ONEBYTE, BYTE_32, p_FileToRead);
        MaxIndexCharOld = MaxIndexChar;
        MaxIndexChar = (unsigned long)fread(p_ReadMemory, ONEBYTE, ReadMemoryWidth,
                                            p_FileToRead);
        MaxIndexInt = MaxIndexChar / FOURBYTE;
        aktualize_FilePosition();

        //4 Bytebloecke drehen und oder invertieren
        if(Inverting)
        {
```

```
                for(k=0;k<ReadMemoryWidth;k=k+4)
                {
                        byte1 = ~(p_ReadMemory[k]);
                        byte2 = ~(p_ReadMemory[k+ONEBYTE]);
                        p_ReadMemory[k] = ~(p_ReadMemory[k+THREEBYTE]);
                        p_ReadMemory[k+ONEBYTE] = ~(p_ReadMemory[k+TWOBYTE]);
                        p_ReadMemory[k+TWOBYTE] = byte2;
                        p_ReadMemory[k+THREEBYTE] = byte1;
                }
        }
        else
        {
                for(k=0;k<ReadMemoryWidth;k=k+4)
                {
                        byte1 = p_ReadMemory[k];
                        byte2 = p_ReadMemory[k+ONEBYTE];
                        p_ReadMemory[k] = p_ReadMemory[k+THREEBYTE];
                        p_ReadMemory[k+ONEBYTE] = p_ReadMemory[k+TWOBYTE];
                        p_ReadMemory[k+TWOBYTE] = byte2;
                        p_ReadMemory[k+THREEBYTE] = byte1;
                }
        }
}

inline unsigned long ct_MemoryAndFileManagment::get_long_value_from_read_memory(register
                                                                    unsigned long index_char)
{
        register unsigned long value;

        if(index_char <= (MaxIndexChar - FOURBYTE))
        {
                value  = (unsigned long)p_ReadMemory[index_char] << BIT_24;
                value |= (unsigned long)p_ReadMemory[index_char+ONEBYTE] << BIT_16;
                value |= (unsigned long)p_ReadMemory[index_char+TWOBYTE] << EIGHTBIT;
                value |= p_ReadMemory[index_char+THREEBYTE];

                return value;
        }
        else
        {
                cout << "get_long_value_from_read_memory : index_char ist zu gross oder
                         MaxIndexChar zu klein, weil Datei zu Ende" << endl;
        }

return 0; //Beim auftreten eines Fehlers wird 0 als Wert zurueckgegeben
}

inline bool ct_MemoryAndFileManagment::get_next_long_value_from_read_memory(register
                                                                    unsigned long &value)
{
        register unsigned long index_on_array;
        // beim Programmstart wird hier das erste Mal eingelesen,
           da IndexOnIntArray von 0 auf 1 gesetzt wird
        IndexOnIntArray++;//und MaxIndexInt noch 0

        if(IndexOnIntArray >= MaxIndexInt) //Maximaler Index erreicht bzw. drueber
        {       //das letzte Element im Speicher
                BufferReadMemory = get_long_value_from_read_memory(MaxIndexChar - FOURBYTE);
                IndexOnIntArray = 0;
                read_frames_from_file();// Wenn nach dem Einlesen MaxIndex = 0 Dateiende
                if(MaxIndexChar < FOURBYTE) // wenn keine Daten eingelesen wurden
                {
                        return true;
                }
        }
        index_on_array = IndexOnIntArray * FOURBYTE;
        value  = (unsigned long)p_ReadMemory[index_on_array] << BIT_24;
        value |= (unsigned long)p_ReadMemory[index_on_array+ONEBYTE] << BIT_16;
        value |= (unsigned long)p_ReadMemory[index_on_array+TWOBYTE] << EIGHTBIT;
        value |= p_ReadMemory[index_on_array+THREEBYTE];

return false;
}

inline void ct_MemoryAndFileManagment::write_synchronized_bytes_to_file(void)
{
        register unsigned long status;
        //Die Datei wird erst erzeugt, wenn synchronisierte Daten vorliegen
```

```
        if(p_FileToWrite == NULL)
                open_file_to_write();

        if(EnableReedSolomonCheck)
        {
                status = check_on_errors(p_WriteMemory, FrameNumber);
                FramesWithErrors += status;
        }

        status = (unsigned long)fwrite(p_WriteMemory, ONEBYTE, WriteMemoryWidth,
                                       p_FileToWrite);
        if(status < WriteMemoryWidth)
                cout << "Es wurden weniger Bloecke in die Datei geschrieben als vorhanden
                        waren!" << endl;
}

inline void ct_MemoryAndFileManagment::write_synchronized_bytes_to_file(unsigned long
                                                                         number_of_bytes)
{
        unsigned long status;
        //Die Datei wird erst erzeugt, wenn synchronisierte Daten vorliegen
        if(p_FileToWrite == NULL)
                open_file_to_write();

        if(EnableReedSolomonCheck)
        {
                status = check_on_errors(p_WriteMemory, FrameNumber - FreeSpaceOnWriteMemory);
                FramesWithErrors += status;
        }

        status = (unsigned long)fwrite(p_WriteMemory, ONEBYTE, number_of_bytes, p_FileToWrite);
        if(status < number_of_bytes)
                cout << "Es wurden weniger Bloecke in die Datei geschrieben als vorhanden
                        waren!" << endl;
}

inline void ct_MemoryAndFileManagment::set_begining(register unsigned int byte, register
                                                    unsigned int bit_shift)
{   // byte kann Programmbeding nur Werte von 0 bis 3 annehmen und bit_shift von 0 bis 7
        register unsigned long value;
        register unsigned int k;
        unsigned char byte_value[4];

        if(IndexOnIntArray < 1) //d.h. wenn 0
        {// es wurde neu eingelesen
                ReadStatus = true;
                value = (MaxIndexCharOld - FOURBYTE) + byte;
                byte_value[0] = (unsigned char) (BufferReadMemory >> BIT_24);
                byte_value[1] = (unsigned char) (BufferReadMemory >> BIT_16);
                byte_value[2] = (unsigned char) (BufferReadMemory >> EIGHTBIT);
                byte_value[3] = (unsigned char)  BufferReadMemory;
                for(k=0;k<(3-byte);k++)
                {
                        p_WriteMemory[k] = (byte_value[k + byte] << bit_shift) |
                           (byte_value[k + byte + 1] >> (EIGHTBYTE - bit_shift));
                }
                IndexWriteMemory = 3 - byte;
                BufferWriteMemory = byte_value[3]; //funktioniert
                BeginToCopy = 0;
        }
        else
        {
                value = FOURBYTE * (IndexOnIntArray - 1) + byte ;
                BufferWriteMemory = p_ReadMemory[value];
                BeginToCopy = value + 1;
        }
        IndexOnCharArray = value;
        BitShift = bit_shift;
}

inline void ct_MemoryAndFileManagment::reset_write_memory_options(void)
{
        FreeSpaceOnWriteMemory = FrameNumber;
        IndexWriteMemory = 0;
}

inline bool ct_MemoryAndFileManagment::get_char_value_from_read_memory(register
 unsigned int byte_offset, register unsigned long &value1, register unsigned long &value2)
```

```
{
	register unsigned long position;
	register unsigned long index;
	register unsigned int carry_over;

		index = IndexOnCharArray;
		position = index + byte_offset; // neuer Index
		//2 * 4Byte werden am Ende von position eingelesen
		if((position + EIGHTBYTE) <= MaxIndexChar)
		{	// die acht Byte befinden sich noch innerhalb des Speichers

			value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
			value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
			value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
			value1 |= (unsigned long)p_ReadMemory[position+THREEBYTE];

			value2  = (unsigned long)p_ReadMemory[position+FOURBYTE] << BIT_24;
			value2 |= (unsigned long)p_ReadMemory[position+FIVEBYTE] << BIT_16;
			value2 |= (unsigned long)p_ReadMemory[position+SIXBYTE] << EIGHTBIT;
			value2 |= (unsigned long)p_ReadMemory[position+SEVENBYTE];

			IndexOnCharArray = position; // Maximal MaxIndexChar - 8 Byte
			decreaseFreeSpaceOnWriteMemory();
		}
		else
		{	//kann nicht negativ werden
			if(!ReadStatus)
				carry_over = (position + EIGHTBYTE) - MaxIndexChar;
			else
				carry_over = (position + EIGHTBYTE) - MaxIndexCharOld;
			//Wie viele Bytes vom naechsten Speicherblock
			switch (carry_over)
			{
	// case 0 wird mit der if((position + EIGHTBYTE) <= MaxIndexChar) Anweisung abgefangen
		case 1 :	IndexOnCharArray = position; //MaxIndexChar-7

			value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
			value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
			value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
			value1 |= (unsigned long)p_ReadMemory[position+THREEBYTE];

			value2  = (unsigned long)p_ReadMemory[position+FOURBYTE] << BIT_24;
			value2 |= (unsigned long)p_ReadMemory[position+FIVEBYTE] << BIT_16;
			value2 |= (unsigned long)p_ReadMemory[position+SIXBYTE] << EIGHTBIT;
			BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
			decreaseFreeSpaceOnWriteMemory();
			synchronize_bytes();
			read_frames_from_file();
			ReadStatus = true;
			if(MaxIndexChar < ONEBYTE)
			{
				FrameAtFileEnd = true;
				return true;//Dateiende oder Fehler
			}
			value2 |= (unsigned long)p_ReadMemory[0];
		break;

			case 2 :	IndexOnCharArray = position; //MaxIndexChar-6

			value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
			value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
			value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
			value1 |= (unsigned long)p_ReadMemory[position+THREEBYTE];

			value2  = (unsigned long)p_ReadMemory[position+FOURBYTE] << BIT_24;
			value2 |= (unsigned long)p_ReadMemory[position+FIVEBYTE] << BIT_16;
			BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
			decreaseFreeSpaceOnWriteMemory();
			synchronize_bytes();
			read_frames_from_file();
			ReadStatus = true;
			if(MaxIndexChar < TWOBYTE)
			{
				FrameAtFileEnd = true;
				return true;//Dateiende oder Fehler
			}
			value2 |= (unsigned long)p_ReadMemory[0] << EIGHTBIT;
			value2 |= (unsigned long)p_ReadMemory[1];
```

```
break;

case 3 :       IndexOnCharArray = position; //MaxIndexChar - 5;

        value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
        value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
        value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
        value1 |= (unsigned long)p_ReadMemory[position+THREEBYTE];

        value2  = (unsigned long)p_ReadMemory[position+FOURBYTE] << BIT_24;
        BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
        decreaseFreeSpaceOnWriteMemory();
        synchronize_bytes();
        read_frames_from_file();
        ReadStatus = true;
        if(MaxIndexChar < THREEBYTE)
        {
                FrameAtFileEnd = true;
                return true;//Dateiende oder Fehler
        }
        value2 |= (unsigned long)p_ReadMemory[0] << BIT_16;
        value2 |= (unsigned long)p_ReadMemory[1] << EIGHTBIT;
        value2 |= (unsigned long)p_ReadMemory[2];
break;

case 4 :       IndexOnCharArray = position; //MaxIndexChar - 4;

        value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
        value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
        value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
        value1 |= (unsigned long)p_ReadMemory[position+THREEBYTE];
        BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
        decreaseFreeSpaceOnWriteMemory();
        synchronize_bytes();
        read_frames_from_file();
        ReadStatus = true;
        if(MaxIndexChar < FOURBYTE)
        {
                FrameAtFileEnd = true;
                return true;//Dateiende oder Fehler
        }
        value2  = (unsigned long)p_ReadMemory[0] << BIT_24;
        value2 |= (unsigned long)p_ReadMemory[1] << BIT_16;
        value2 |= (unsigned long)p_ReadMemory[2] << EIGHTBIT;
        value2 |= (unsigned long)p_ReadMemory[3];
break;

case 5 :       IndexOnCharArray = position; //MaxIndexChar - 3;

        value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
        value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
        value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
        BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
        decreaseFreeSpaceOnWriteMemory();
        synchronize_bytes();
        read_frames_from_file();
        ReadStatus = true;
        if(MaxIndexChar < FIVEBYTE)
        {
                FrameAtFileEnd = true;
                return true;//Dateiende oder Fehler
        }
        value1 |= (unsigned long)p_ReadMemory[0];
        value2  = (unsigned long)p_ReadMemory[1] << BIT_24;
        value2 |= (unsigned long)p_ReadMemory[2] << BIT_16;
        value2 |= (unsigned long)p_ReadMemory[3] << EIGHTBIT;
        value2 |= (unsigned long)p_ReadMemory[4];
break;

case 6 :       IndexOnCharArray = position; //MaxIndexChar - 2;

        value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
        value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
        BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
        decreaseFreeSpaceOnWriteMemory();
        synchronize_bytes();
        read_frames_from_file();
        ReadStatus = true;
```

```
                    if(MaxIndexChar < SIXBYTE)
                    {
                            FrameAtFileEnd = true;
                            return true;//Dateiende oder Fehler
                    }
                    value1 |= (unsigned long)p_ReadMemory[0] << EIGHTBIT;
                    value1 |= (unsigned long)p_ReadMemory[1];
                    value2  = (unsigned long)p_ReadMemory[2] << BIT_24;
                    value2 |= (unsigned long)p_ReadMemory[3] << BIT_16;
                    value2 |= (unsigned long)p_ReadMemory[4] << EIGHTBIT;
                    value2 |= (unsigned long)p_ReadMemory[5];
            break;

            case 7 :        IndexOnCharArray = position; //MaxIndexChar - 1;

                    value1 = (unsigned long) p_ReadMemory[position] << BIT_24;
                    BufferReadMemory = get_long_value_from_read_memory(position - FOURBYTE);
                    decreaseFreeSpaceOnWriteMemory();
                    synchronize_bytes();
                    read_frames_from_file();
                    ReadStatus = true;
                    if(MaxIndexChar < SEVENBYTE)
                    {
                            FrameAtFileEnd = true;
                            return true;//Dateiende oder Fehler
                    }

                    value1 |= (unsigned long)p_ReadMemory[0] << BIT_16;
                    value1 |= (unsigned long)p_ReadMemory[1] << EIGHTBIT;
                    value1 |= (unsigned long)p_ReadMemory[2];
                    value2  = (unsigned long)p_ReadMemory[3] << BIT_24;
                    value2 |= (unsigned long)p_ReadMemory[4] << BIT_16;
                    value2 |= (unsigned long)p_ReadMemory[5] << EIGHTBIT;
                    value2 |= (unsigned long)p_ReadMemory[6];
            break;
    //case carry_over >= 8 / carry_over kann hier Werte von 8 bis byte_offset+7 annehmen
            default:        position = carry_over - EIGHTBYTE;//Index fuer neue Position
                            IndexOnCharArray = position; // 0 bis byte_offset-1
                    if(!ReadStatus) //position < (byte_offset - 7))
                    {       //Daten nicht nochmal abspeichern und neuen Satz laden!!!!!
            BufferReadMemory = get_long_value_from_read_memory(MaxIndexChar - FOURBYTE);
                            synchronize_bytes();
                            read_frames_from_file();
                            if(MaxIndexChar < carry_over) // Dateiende erreicht
                            { // befindet sich noch ein kompletter Rahmen im Speicher
                                    if(MaxIndexChar >= position)
                                    {
                                            FrameAtFileEnd = true;
                                            synchronize_bytes(position + ONEBYTE);
                                            decreaseFreeSpaceOnWriteMemory();
                                    }
                                    return true;
                            }
                    }
                    else
                    {
                            ReadStatus = false;
                            if(MaxIndexChar < carry_over) // Dateiende erreicht
                            { // befindet sich noch ein kompletter Rahmen im Speicher
                                    if(MaxIndexChar >= position)
                                    {
                                            FrameAtFileEnd = true;
                                            synchronize_bytes(position + ONEBYTE);
                                            decreaseFreeSpaceOnWriteMemory();
                                    }
                                    return true;
                            }
                    }

                    value1  = (unsigned long)p_ReadMemory[position] << BIT_24;
                    value1 |= (unsigned long)p_ReadMemory[position+ONEBYTE] << BIT_16;
                    value1 |= (unsigned long)p_ReadMemory[position+TWOBYTE] << EIGHTBIT;
                    value1 |= (unsigned long)p_ReadMemory[position+THREEBYTE];

                    value2  = (unsigned long)p_ReadMemory[position+FOURBYTE] << BIT_24;
                    value2 |= (unsigned long)p_ReadMemory[position+FIVEBYTE] << BIT_16;
                    value2 |= (unsigned long)p_ReadMemory[position+SIXBYTE] << EIGHTBIT;
                    value2 |= (unsigned long)p_ReadMemory[position+SEVENBYTE];
```

```
                        decreaseFreeSpaceOnWriteMemory();
                        };
                }
return false;
}

inline void ct_MemoryAndFileManagment::decreaseFreeSpaceOnWriteMemory(void)
{
        register unsigned long value;//wird fuer verschiedene Funktionen verwendet

        value = FreeSpaceOnWriteMemory;
        value--;

        if(value) //wenn value > 0
        {
                FreeSpaceOnWriteMemory = value;
        }
        else
        {  //wenn value == 0 ist der Lesespeicher schon voll!
                value = WriteMemoryWidth - IndexWriteMemory;
                synchronize_bytes(value);
                write_synchronized_bytes_to_file();
                reset_write_memory_options();
        }
}

inline void ct_MemoryAndFileManagment::set_index_char_to_int(void)
{   // nachschauen, ob schon durch get_char_value_from_read_memory eingelesen wurde
        // in dem Fall ist IndexOnIntArray = 0
        if(!ReadStatus)
        {
                IndexOnIntArray = IndexOnCharArray / FOURBYTE;
        }
        else
        {
                IndexOnIntArray = 0;
                ReadStatus = false;
        }
}

inline void ct_MemoryAndFileManagment::IndexOnCharArray_plus_one(void)
{
        IndexOnCharArray++;
        if(IndexOnCharArray >= MaxIndexChar)
                IndexOnCharArray = 0;
}

inline void ct_MemoryAndFileManagment::IndexOnCharArray_minus_one(void)
{
        if(IndexOnCharArray > 0) //Wenn auf 0, wurde eingelesen
                IndexOnCharArray--;
        else //somit muss der alte MaxIndex verwendet werden
                IndexOnCharArray = MaxIndexCharOld - ONEBYTE;
}

inline __int64 ct_MemoryAndFileManagment::getBytePosition(void)
{
        __int64 position;

        if(ReadStatus)
                position = FilePosition - (MaxIndexCharOld - IndexOnCharArray);
        else
                position = FilePosition - (MaxIndexChar - IndexOnCharArray);

return position;
}

inline void ct_FrameSynchronizer::output_on_screen_and_file(unsigned int mode)
{
unsigned long frames_with_errors;

switch (mode)
{
case PROGRAMSTART : //Nach dem Start des Programms
        cout  << endl << endl;
        OutputFile << endl << endl;
        cout << "     |          Parameter              |    Wert   |" << endl;
        OutputFile << "     |        Parameter              |    Wert   |" << endl;
```

```
cout << "     |-----------------------------------|-----------|" << endl;
OutputFile << "     |-----------------------------------|-----------|" << endl;
cout.unsetf(ios_base::dec);
OutputFile.unsetf(ios_base::dec);
cout.setf(ios_base::hex);
OutputFile.setf(ios_base::hex);
cout << "     | Attached Sync Marker              |" << setw(10)
     << co_CompareBitpattern.getMask() << " |" << endl;
OutputFile << "     | Attached Sync Marker              |" << setw(10)
           << co_CompareBitpattern.getMask() << " |" << endl;
cout.setf(ios_base::dec);
OutputFile.setf(ios_base::dec);
cout << "     | Rahmenlaenge                      |" << setw(10) << FrameLength
     << " |" << endl;
OutputFile << "     | Rahmenlaenge                      |" << setw(10) << FrameLength
           << " |" << endl;
cout << "     | Zulaessige Bitfehler im ASM       |" << setw(10) << BitErrorMax
     << " |" << endl;
OutputFile << "     | Zulaessige Bitfehler im ASM       |" << setw(10) << BitErrorMax
           << " |" << endl;
cout << "     | Bekannte ASM in Folge bis Locked  |" << setw(10)
     << NumberOfKnownPatternsToLock << " |" << endl;
OutputFile << "     | Bekannte ASM in Folge bis Locked  |" << setw(10)
           << NumberOfKnownPatternsToLock << " |" << endl;
cout << "     | Unbekannte ASM in Folge bis Search |" << setw(10)
     << NumberOfUnknownPatternsToSearch << " |" << endl;
OutputFile << "     | Unbekannte ASM in Folge bis Search |" << setw(10)
           << NumberOfUnknownPatternsToSearch << " |" << endl;
cout << "     | Bitschlupffensterbreite           |" << setw(10) << BitSlipWindowSize
     << " |" << endl;
OutputFile << "     | Bitschlupffensterbreite           |" << setw(10)
           << BitSlipWindowSize << " |" << endl;
cout << "     | Ausgabeblockgroesse               |" << setw(10) << OutputBlockSize
     << " |" << endl;
OutputFile << "     | Ausgabeblockgroesse               |" << setw(10)
           << OutputBlockSize << " |" << endl;
cout << "     | Invertierung                      |";
OutputFile << "     | Invertierung                      |";
if(getInverting())
{
       cout << "      Ja   |" << endl;
       OutputFile << "      Ja   |" << endl;
}
else
{
       cout << "     Nein  |" << endl;
       OutputFile << "     Nein  |" << endl;
}
cout << "     | Reed-Solomon-Check                |";
OutputFile << "     | Reed-Solomon-Check                |";
if(getEnableReedSolomonCheck())
{
       cout << "      Ja   |" << endl;
       OutputFile << "      Ja   |" << endl;
}
else
{
       cout << "     Nein  |" << endl;
       OutputFile << "     Nein  |" << endl;
}
cout << "     |-----------------------------------|-----------|" << endl;
OutputFile << "     |-----------------------------------|-----------|" << endl;
cout << "      Quelle: " << getp_ReadPath() << endl;
OutputFile << "      Quelle: " << getp_ReadPath() << endl;
cout << "      Ziel  : " << getp_WritePath() << endl << endl << endl;
OutputFile << "      Ziel  : " << getp_WritePath() << endl << endl << endl;

cout << "       |      Zustand     | Byteposition |  Rahmen  | R/S-Check | FW-Rahmen |
        ASM  | BER-ASM | Bitslip |" << endl;
OutputFile << "       |      Zustand     | Byteposition |  Rahmen  | R/S-Check |
              FW-Rahmen |  ASM  | BER-ASM | Bitslip |" << endl;
cout << "       |------------------|--------------|----------|-----------|-----------|
                -------|---------|---------|" << endl;
OutputFile << "       |------------------|--------------|----------|-----------|
                 -----------|-------|---------|---------|" << endl;
cout << "       |      Search      |           0  |  ------  |  -------  |  -------  |
         ---  |  -----  |  -----  |" << endl;
OutputFile << "       |      Search      |           0  |  ------  |  -------  |
```

```
                    -------  |  ---  |  -----  |  -----  |" << endl;
break;

case ENDOFFILE : //Dateiende
       frames_with_errors = get_and_reset_FramesWithErrors();
       BitErrorCounter += BlockBitErrorCounter;
       FrameCounter += BlockFrameCounter;
       FlyWheelFrames += BlockFlyWheelFrames;
       BitSlipCounter += BlockBitSlipCounter;

cout << "       |     Dateiende    | " << setw(11) << getFilePosition() << "  | " << setw(8);
OutputFile << "       |     Dateiende    | " << setw(11) << getFilePosition() << "  | "
          << setw(8);
cout << BlockFrameCounter << " | " << setw(8) << frames_with_errors << "  |  " << setw(7);
OutputFile << BlockFrameCounter << " | " << setw(8) << frames_with_errors << "  |  "
          << setw(7);
cout << BlockFlyWheelFrames << "  | " << setw(5) << BlockBitErrorCounter << " | "
     << setprecision(1);
OutputFile << BlockFlyWheelFrames << "  | " << setw(5) << BlockBitErrorCounter << " | "
          << setprecision(1);
cout << setiosflags(ios_base::scientific);
OutputFile << setiosflags(ios_base::scientific);
       if(BlockFrameCounter != 0)
       {
              cout << ((double)BlockBitErrorCounter/(BlockFrameCounter*BIT_32));
              OutputFile << ((double)BlockBitErrorCounter/(BlockFrameCounter*BIT_32));
       }
       else
       {
              cout << (double)BlockFrameCounter;//Hat den Wert null -> BER = 0
              OutputFile << (double)BlockFrameCounter;//Hat den Wert null -> BER = 0
       }
       cout << "|    " << setw(2) << BlockBitSlipCounter << "   |" << endl;
       OutputFile << "|    " << setw(2) << BlockBitSlipCounter << "   |" << endl;

cout << "       |       Gesamt     | " << setw(11) << getFilePosition() << "  | " << setw(8);
OutputFile << "       |       Gesamt     | " << setw(11) << getFilePosition() << "  | "
          << setw(8);
cout << FrameCounter << " | " << setw(8) << getAllFramesWithErrors() << "  |  " << setw(7);
OutputFile << FrameCounter << " | " << setw(8) << getAllFramesWithErrors() << "  |  "
          << setw(7);
cout << FlyWheelFrames << "  | " << setw(5) << BitErrorCounter << " | " << setprecision(1);
OutputFile << FlyWheelFrames << "  | " << setw(5) << BitErrorCounter << " | "
          << setprecision(1);
       cout << setiosflags(ios_base::scientific);
       OutputFile << setiosflags(ios_base::scientific);
       if(FrameCounter != 0)
       {
              cout << ((double)BitErrorCounter/(FrameCounter*BIT_32));
              OutputFile << ((double)BitErrorCounter/(FrameCounter*BIT_32));
       }
       else
       {
              cout << (double)FrameCounter;//Hat den Wert null -> BER = 0
              OutputFile << (double)FrameCounter;//Hat den Wert null -> BER = 0
       }
       cout << "|    " << setw(2) << BitSlipCounter << "   |" << endl;
       OutputFile << "|    " << setw(2) << BitSlipCounter << "   |" << endl;
       cout << "       |------------------|--------------|----------|-----------|-----------|
                -------|";
       OutputFile << "       |------------------|--------------|----------|-----------|
                    -----------|-------|";
       cout << "---------|---------|" << endl;
       OutputFile << "---------|---------|" << endl;
break;

case LOCKED :
       cout << "       | Search -> Locked | " << setw(11);
       OutputFile << "       | Search -> Locked | " << setw(11);
       cout << getBytePosition() - NumberOfKnownPatternsToLock * FrameLength;
       OutputFile << getBytePosition() - NumberOfKnownPatternsToLock * FrameLength;
       cout << "  |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |" << endl;
OutputFile << "  |  ------  |  -------  |  -------  |  ---  |  -----  |  -----  |" << endl;
break;

case SEARCH :
       frames_with_errors = get_and_reset_FramesWithErrors();
       BitErrorCounter += BlockBitErrorCounter;
```

```
        FrameCounter += BlockFrameCounter;
        FlyWheelFrames += BlockFlyWheelFrames;
        BitSlipCounter += BlockBitSlipCounter;

cout << "       | Locked -> Search | " << setw(11) << getBytePosition() << "  | " << setw(8);
        OutputFile << "       | Locked -> Search | " << setw(11) << getBytePosition() << "  | "
                   << setw(8);
cout << BlockFrameCounter << " | " << setw(8) << frames_with_errors << "  |  " << setw(7);
        OutputFile << BlockFrameCounter << " | " << setw(8) << frames_with_errors << "  |  "
                   << setw(7);
        cout << BlockFlyWheelFrames << "  | " << setw(5) << BlockBitErrorCounter << " | "
             << setprecision(1);
        OutputFile << BlockFlyWheelFrames << "  | " << setw(5) << BlockBitErrorCounter << " | "
                    << setprecision(1);
        cout << setiosflags(ios_base::scientific)
             << ((double)BlockBitErrorCounter/(BlockFrameCounter*BIT_32));
        OutputFile << setiosflags(ios_base::scientific)
                   << ((double)BlockBitErrorCounter/(BlockFrameCounter*BIT_32));
        cout << "|    " << setw(2) << BlockBitSlipCounter << "   |" << endl;
        OutputFile << "|    " << setw(2) << BlockBitSlipCounter << "   |" << endl;

        BlockBitErrorCounter = 0;
        BlockFrameCounter = 0;
        BlockFlyWheelFrames = 0;
        BlockBitSlipCounter = 0;
break;

case BLOCK : //Ausgabe nach erreichen der Blockgroesse
        frames_with_errors = get_and_reset_FramesWithErrors();
        BitErrorCounter += BlockBitErrorCounter;
        FrameCounter += BlockFrameCounter;
        FlyWheelFrames += BlockFlyWheelFrames;
        BitSlipCounter += BlockBitSlipCounter;

cout << "       |      Locked      | " << setw(11) << getBytePosition() << "  | " << setw(8);
OutputFile << "       |      Locked      | " << setw(11) << getBytePosition() << "  | "
           << setw(8);
cout << BlockFrameCounter << " | " << setw(8) << frames_with_errors << "  |  " << setw(7);
OutputFile << BlockFrameCounter << " | " << setw(8) << frames_with_errors << "  |  "
           << setw(7);
cout << BlockFlyWheelFrames << "  | " << setw(5) << BlockBitErrorCounter << " | "
     << setprecision(1);
OutputFile << BlockFlyWheelFrames << "  | " << setw(5) << BlockBitErrorCounter << " | "
           << setprecision(1);
        cout << setiosflags(ios_base::scientific)
             << ((double)BlockBitErrorCounter/(BlockFrameCounter*BIT_32));
        OutputFile << setiosflags(ios_base::scientific)
                   << ((double)BlockBitErrorCounter/(BlockFrameCounter*BIT_32));
        cout << "|    " << setw(2) << BlockBitSlipCounter << "   |" << endl;
        OutputFile << "|    " << setw(2) << BlockBitSlipCounter << "   |" << endl;

        BlockBitErrorCounter = 0;
        BlockFrameCounter = 0;
        BlockFlyWheelFrames = 0;
        BlockBitSlipCounter = 0;
break;
}
```

9.2.3 frame_synchronizer.cpp

```
#include "frame_synchronizer.h"

ct_CompareBitpattern::~ct_CompareBitpattern()
{
        if(p_LookUpTable1 != NULL)
                delete [] p_LookUpTable1;
        if(p_LookUpTable2 != NULL)
                delete [] p_LookUpTable2;
}
void ct_CompareBitpattern::initLook_up_tables(void)
{
        unsigned int k, l;
        unsigned int value;
        unsigned short bits;
        unsigned short pattern1, pattern2;
```

```
	//Bei einer Reinitialisierung, muessen die alten Speicherbereiche aufgeloest werden
	if(p_LookUpTable1 != NULL)
		delete [] p_LookUpTable1;
	if(p_LookUpTable2 != NULL)
		delete [] p_LookUpTable2;

	p_LookUpTable1 = new unsigned char[USHRT_MAX + ONEBYTE];
	p_LookUpTable2 = new unsigned char[USHRT_MAX + ONEBYTE];

	if(p_LookUpTable1 == NULL)
	{
		cout << "Kein Speicher fuer LookUpTable1 vorhanden!" << endl;
		exit(0);
	}

	if(p_LookUpTable2 == NULL)
	{
		cout << "Kein Speicher fuer LookUpTable2 vorhanden!" << endl;
		exit(0);
	}

	pattern1 = (short)(Mask >> BIT_16);
	pattern2 = (short) Mask;
	for(k=0;k<=USHRT_MAX;k++)
	{
		value = 0;
		bits = k^pattern1;
		for(l=0;l<16;l++)
		{
			value += (bits & 1);
			bits = bits >> 1;
		}
		p_LookUpTable1[k] = value;

		value = 0;
		bits = k^pattern2;
		for(l=0;l<16;l++)
		{
			value += (bits & 1);
			bits = bits >> 1;
		}
		p_LookUpTable2[k] = value;
	}
}

ct_MemoryAndFileManagment::ct_MemoryAndFileManagment(unsigned int frame_length,
char *read_path, char *write_path, unsigned int reed_solomon_check, bool invert) :
ct_ReedSolomonCheck(frame_length)
{
	size_t string_length;

	ReadMemoryWidth = FOURMEGABYTES;
	WriteMemoryWidth = frame_length * FACTOR; //Speichergroesse
	FrameNumber = FACTOR;
	MaxIndexInt = 0;
	MaxIndexChar = 0; //bekommt den Rueckgabewert von read
	FramesWithErrors = 0;
	AllFramesWithErrors = 0;
	IndexWriteMemory = 0;
	IndexOnIntArray = 0;
	IndexOnCharArray = 0;
	FilePosition = 0;
	ReadStatus = false;
	FrameAtFileEnd = false;
	Inverting = invert;
	if(reed_solomon_check == 0)
		EnableReedSolomonCheck = false;
	else
		EnableReedSolomonCheck = true;

	FreeSpaceOnWriteMemory = FACTOR; //Anzahl an Rahmen, die in den Speicher passen
	p_FileToRead = NULL;
	p_FileToWrite = NULL;
	//Speicher initialisieren
	p_ReadMemory = new unsigned char[FOURMEGABYTES];
	p_WriteMemory = new unsigned char[frame_length * FACTOR];
	if(p_ReadMemory == NULL)
	{
```

```
                cout << "Kein Speicher fuer p_ReadMemory vorhanden!" << endl;
                exit(0);
        }

        if(p_WriteMemory == NULL)
        {
                cout << "Kein Speicher fuer p_WriteMemory vorhanden!" << endl;
                exit(0);
        }
        //Pfad zum Lesen kopieren
        string_length = strlen(read_path);
        p_ReadPath = new char[string_length + ONEBYTE];
        if(p_ReadPath == NULL)
        {
                cout << "Kein Speicher fuer p_ReadPath vorhanden!" << endl;
                exit(0);
        }
        strcpy_s(p_ReadPath, string_length + ONEBYTE, read_path);
        //Pfad zum Schreiben kopieren
        string_length = strlen(write_path);
        p_WritePath = new char[string_length + ONEBYTE];
        if(p_WritePath == NULL)
        {
                cout << "Kein Speicher fuer p_WritePath vorhanden!" << endl;
                exit(0);
        }
        strcpy_s(p_WritePath, string_length + ONEBYTE, write_path);
}


ct_MemoryAndFileManagment::~ct_MemoryAndFileManagment()
{       //Speicher freigeben
        if(p_ReadMemory != NULL)
                delete [] p_ReadMemory;
        if(p_WriteMemory != NULL)
                delete [] p_WriteMemory;
        if(p_ReadPath != NULL)
                delete [] p_ReadPath;
        if(p_WritePath != NULL)
                delete [] p_WritePath;
        //Streams schliessen
        if(p_FileToRead != NULL)
                fclose(p_FileToRead);
        if(p_FileToWrite != NULL)
                fclose(p_FileToWrite);
}

void ct_FrameSynchronizer::open_output_file(char *write_path)
{
        size_t string_length;
        char *p_OutputPath;
    char *zeichen = NULL;
        //Pfad zur Datei zum schreiben kopieren
        string_length = strlen(write_path);
        p_OutputPath = new char[string_length + FOURBYTE];
        if(p_OutputPath == NULL)
        {
                cout << "Kein Speicher fuer p_OutputPath vorhanden!" << endl;
                exit(0);
        }
        strcpy_s(p_OutputPath, string_length + ONEBYTE, write_path);

        zeichen = strrchr(p_OutputPath,'.');//Letzten '.' suchen
        zeichen++;//damit der '.' erhalten bleibt
        strcpy(zeichen,"txt");//hinter dem '.' die Endung txt hineinschreiben

        OutputFile.open(p_OutputPath, ios_base::out | ios_base::trunc );
        if(!(OutputFile.is_open()))
        {
                cout << "Outputdatei konnte nicht geoeffnet werden!" << endl;
                exit(0);
        }
        delete [] p_OutputPath;
}

void ct_MemoryAndFileManagment::open_file_to_read(void)
{
        fopen_s(&p_FileToRead, p_ReadPath, "r+b");
```

```
	if(p_FileToRead == NULL)
	{
		cout << "Datei zum Lesen konnte nicht geoeffnet werden!" << endl;
		exit(0);
	}
	//Dateiheader des Datenrekorders auslassen
	_fseeki64(p_FileToRead, 65536, SEEK_SET);

}

void ct_MemoryAndFileManagment::open_file_to_write(void)
{
	fopen_s(&p_FileToWrite, p_WritePath, "w+b");
	if(p_FileToWrite == NULL)
	{
		cout << "Datei zum Schreiben konnte nicht geoeffnet werden!" << endl;
		exit(0);
	}
}

bool ct_FrameSynchronizer::search_bitpattern(void)
{
	register unsigned long zmemory, zpattern; //Zwischenspeicher
	register unsigned long value;
	register unsigned int bit_error_max;
	register bool end_of_file;

	bit_error_max = BitErrorMax;
	reset_write_memory_options();
	end_of_file = get_next_long_value_from_read_memory(zpattern);
	if(end_of_file)//Dateiende
		return true;

	while(1) //solange suchen, bis ein SW gefunden wurde oder das Dateiende erreicht wird
	{
		end_of_file = get_next_long_value_from_read_memory(zmemory);
		if(end_of_file)//Dateiende
			return end_of_file;

		value = co_CompareBitpattern.compare(zpattern);
		if(value <= bit_error_max)
		{
			set_begining(ZEROBYTE, ZEROBIT);
			return false;
		}

		zpattern = zpattern << 1;
		zpattern = zpattern | (zmemory >> 31);
		value = co_CompareBitpattern.compare(zpattern);
		if(value <= bit_error_max)
		{
			set_begining(ZEROBYTE, ONEBIT);
			return false;
		}

		zpattern = zpattern << 1;
		zpattern = zpattern | (zmemory >> 30);
		value = co_CompareBitpattern.compare(zpattern);
		if(value <= bit_error_max)
		{
			set_begining(ZEROBYTE, TWOBIT);
			return false;
		}

		zpattern = zpattern << 1;
		zpattern = zpattern | (zmemory >> 29);
		value = co_CompareBitpattern.compare(zpattern);
		if(value <= bit_error_max)
		{
			set_begining(ZEROBYTE, THREEBIT);
			return false;
		}

		zpattern = zpattern << 1;
		zpattern = zpattern | (zmemory >> 28);
		value = co_CompareBitpattern.compare(zpattern);
		if(value <= bit_error_max)
		{
```

```
        set_begining(ZEROBYTE, FOURBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 27);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ZEROBYTE, FIVEBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 26);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ZEROBYTE, SIXBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 25);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ZEROBYTE, SEVENBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 24);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ONEBYTE, ZEROBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 23);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ONEBYTE, ONEBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 22);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ONEBYTE, TWOBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 21);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ONEBYTE, THREEBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 20);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
        set_begining(ONEBYTE, FOURBIT);
        return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 19);
```

```
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(ONEBYTE, FIVEBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 18);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(ONEBYTE, SIXBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 17);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(ONEBYTE, SEVENBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 16);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, ZEROBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 15);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, ONEBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 14);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, TWOBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 13);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, THREEBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 12);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, FOURBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 11);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, FIVEBIT);
	return false;
}
```

```
zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 10);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, SIXBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 9);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(TWOBYTE, SEVENBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 8);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(THREEBYTE, ZEROBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 7);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(THREEBYTE, ONEBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 6);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(THREEBYTE, TWOBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 5);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(THREEBYTE, THREEBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 4);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(THREEBYTE, FOURBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 3);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
	set_begining(THREEBYTE, FIVEBIT);
	return false;
}

zpattern = zpattern << 1;
zpattern = zpattern | (zmemory >> 2);
value = co_CompareBitpattern.compare(zpattern);
if(value <= bit_error_max)
{
```

```
                        set_begining(THREEBYTE, SIXBIT);
                        return false;
                }

                zpattern = zpattern << 1;
                zpattern = zpattern | (zmemory >> 1);
                value = co_CompareBitpattern.compare(zpattern);
                if(value <= bit_error_max)
                {
                        set_begining(THREEBYTE, SEVENBIT);
                        return false;
                }
                //steht fuer eine Verschiebung von 32 Bit, was wiederrum 0 Bit entspricht
                zpattern = zmemory;   }
}

unsigned int ct_FrameSynchronizer::check_next_pattern(void)
{
        unsigned long memory_one, memory_two;
        unsigned int approach = 0;
        unsigned int bit_error = 0;
        unsigned int count;
        unsigned int bit_shift;
        unsigned int status;
        bool end_of_file;

        bit_shift = getBitShift();
        do
        {
         approach++; // erster, zweiter, dritter ... Versuch

        end_of_file = get_char_value_from_read_memory(FrameLength, memory_one, memory_two);
                if(end_of_file)//bei Dateiende
                        return ENDOFFILE; // Status hat den Wert = 1 -> Dateiende wurde erreicht

                if(bit_shift != ZEROBIT)
                {
                memory_one = (memory_one << bit_shift) | (memory_two >> (BIT_32 - bit_shift));
                }

                count = co_CompareBitpattern.compare(memory_one);
                bit_error += count;

                if(count > BitErrorMax)
                {
                        status = getIndexOnCharArray();
        // Wenn neu eingelesen wurde, wird von 0 angefangen zu suchen, sonst IndexOnIntArray--
                        if(status < (approach * FrameLength))
                        {
                                setIndexOnIntArray(0);
                        }
                        else
                        {   //IndexOnIntArray kann hier nicht 0 sein, IndexOnIntArray-- erlaubt
                                setIndexOnIntArray(getIndexOnIntArray() - 1);
                        }
                        return NOSYNC;  //Fehlschlag
                }
        }while(approach < NumberOfKnownPatternsToLock);
        BlockBitErrorCounter = bit_error;
        BlockFrameCounter = approach;

return SYNC; //Erfolg
}
                        //Anzahl an Bytes die in den WriteMemory geschrieben werden
void ct_MemoryAndFileManagment::synchronize_bytes(register unsigned long number_of_bytes)
{

        register unsigned long i;
        register unsigned long offset;
        register unsigned int bit_shift;
        register unsigned char *p_read_memory_char = NULL;
        register unsigned char *p_write_memory_char = NULL;

        if(number_of_bytes > ZEROBYTE) //wenn 0 Bytes synchronisiert werden, nichts machen
        {
                if(BeginToCopy == MaxIndexChar)
                        BeginToCopy = 0;
```

```
                offset = IndexWriteMemory;
                if((offset + number_of_bytes) <= WriteMemoryWidth) //zu Debugzwecken!
                {
                        bit_shift = BitShift;
                        p_read_memory_char = &p_ReadMemory[BeginToCopy];
                        p_write_memory_char = &p_WriteMemory[offset];

                        if(bit_shift != ZEROBIT)
                        {
                                p_write_memory_char[0] = (BufferWriteMemory << bit_shift) |
                                          (p_read_memory_char[0] >> (EIGHTBIT - bit_shift));

                                for(i=1;i<number_of_bytes;i++)
                                {

                                        p_write_memory_char[i] = (p_read_memory_char[i-1] <<
                             bit_shift) | (p_read_memory_char[i] >> (EIGHTBIT - bit_shift));
                                }
                        }
                        else
                        {
                                p_write_memory_char[0] = BufferWriteMemory;

                                for(i=1;i<number_of_bytes;i++)
                                {

                                        p_write_memory_char[i] = p_read_memory_char[i-1];
                                }
                        }

                        BeginToCopy += number_of_bytes;
                        IndexWriteMemory += number_of_bytes; // richtig
                        BufferWriteMemory = p_read_memory_char[number_of_bytes-ONEBYTE];
                }
                else
                {
                        cout << "number_of_bytes ist zu hoch zum Synchronisieren!" << endl;
                        cout << (offset + number_of_bytes) - MaxIndexChar << endl;
                }
        }
}
      // synchronisiert bis zum Ende des ReadMemory
void ct_MemoryAndFileManagment::synchronize_bytes(void)
{ // beachten, das nicht zu viel in den Speicher geschrieben wird
        register unsigned long number_of_bytes;
        register unsigned long offset;
        register unsigned long i;
        register unsigned int bit_shift;
        register unsigned char *p_read_memory_char = NULL;
        register unsigned char *p_write_memory_char = NULL;

        if(BeginToCopy > MaxIndexChar)//tritt nur am Ende der Datei auf
                BeginToCopy = 0;

        number_of_bytes = MaxIndexChar - BeginToCopy;
        if(number_of_bytes > ZEROBYTE) //wenn 0 Bytes synchronisiert werden, nichts machen
        {
                offset = IndexWriteMemory;
                if((offset + number_of_bytes) <= WriteMemoryWidth) //zu Debugzwecken!
                {
                        bit_shift = BitShift;
                        p_read_memory_char = &p_ReadMemory[BeginToCopy];
                        p_write_memory_char = &p_WriteMemory[offset];

                        if(bit_shift != ZEROBIT)
                        {
                                p_write_memory_char[0] = (BufferWriteMemory << bit_shift) |
                                          (p_read_memory_char[0] >> (EIGHTBIT - bit_shift));

                                for(i=1;i<number_of_bytes;i++)
                                {

                                        p_write_memory_char[i] = (p_read_memory_char[i-1] <<
                                    bit_shift) | (p_read_memory_char[i] >> (EIGHTBIT - bit_shift));
                                }
                        }
                        else
                        {
```

```
                                p_write_memory_char[0] = BufferWriteMemory;

                                for(i=1;i<number_of_bytes;i++)
                                {

                                        p_write_memory_char[i] = p_read_memory_char[i-1];
                                }
                        }

                        BeginToCopy = 0;//Da bis zum Speicherende synchronisiert wird
                        IndexWriteMemory += number_of_bytes;
                        BufferWriteMemory = p_read_memory_char[number_of_bytes-ONEBYTE];
                }
                else
                {
                        cout << "number_of_bytes ist zu hoch zum Synchronisieren von
                                 synchronize_bytes(void)!" << endl;
                        cout << (offset + number_of_bytes) - MaxIndexChar << endl;
                }
        }
}

bool ct_FrameSynchronizer::locked_mode(void)
{
        register unsigned long memory_one;
        register unsigned long memory_two;
        register unsigned long solution;
        register unsigned int approach = 0;
        register unsigned int bit_error = 0;
        register unsigned int bit_shift;
        register unsigned int status;
        register bool end_of_file;

        do
        {
                bit_shift = getBitShift();
        end_of_file = get_char_value_from_read_memory(FrameLength, memory_one, memory_two);
                if(end_of_file) //Dateiende?
                {
                        if(getFrameAtFileEnd())
                                BlockFrameCounter++;

                        rescue_data_after_sync_lose_or_file_end(FrameLength, end_of_file);
                        return end_of_file;
                }

                if(bit_shift != ZEROBIT)
                {
                solution = (memory_one << bit_shift) | (memory_two >> (BIT_32 - bit_shift));
                }
                else
                {
                        solution = memory_one;
                }
                bit_error = co_CompareBitpattern.compare(solution);

                if(bit_error > BitErrorMax)
                {
        //auf Bitschlupf pruefen und erst dann approach erhoehen wenn nichts festgestellt wurde
                        status = search_in_bitslip_window(memory_one, memory_two);
                        if(status != FOUNDNOTHING)
                        {
                                BlockBitSlipCounter++;
                                bit_error = status;
                        }
                        else
                        {
                                approach++;
                                BlockFlyWheelFrames++;
                        }
                }
                else
                {
                        approach = 0;
                }
                BlockBitErrorCounter += bit_error;
                BlockFrameCounter++;
```

```
                if(BlockFrameCounter == OutputBlockSize)
                        output_on_screen_and_file(BLOCK);

        }while(approach < NumberOfUnknownPatternsToSearch);
        //Synchronisieren der Daten und IndexOnIntArray fuer das Suchen des SW neu festlegen
        rescue_data_after_sync_lose_or_file_end(FrameLength, end_of_file);
        set_index_char_to_int();

return false; //Verlust der Synchronisation (kein Dateiende -> false)
}
//Maximal bis 8 Wenn Bitslip erkannt, wird erst alles abgespeichert und dann BitShift
aktuallisiert!
// gibt 0xFF bei Misserfolg zurueck, sonst die Bitfehleranzahl im erkannten Synchronwort
unsigned int ct_FrameSynchronizer::search_in_bitslip_window(unsigned long value1, unsigned
long value2)
{
        unsigned long index_on_char_array;
        unsigned int k;
        unsigned int z_memory;
        unsigned int bit_shift;
        unsigned int bit_error;
        unsigned int shift_plus;
        int shift_minus; // kann negative Werte annehmen

        bit_shift = getBitShift();
        index_on_char_array = getIndexOnCharArray();

        for(k=1;k<=BitSlipWindowSize;k++)
        {
                shift_plus = bit_shift + k;
                shift_minus = bit_shift - k;
                if(shift_plus < EIGHTBIT)
                {
                        z_memory = (value1 << shift_plus) | (value2 >> (BIT_32 - shift_plus));
                        bit_error = co_CompareBitpattern.compare(z_memory);
                        if(bit_error <= BitErrorMax)
                        {
                                synchronize_bit_slip(CURRENTBYTE, value1, value2, shift_plus);
                                return bit_error;
                        }
                }
                else // shift_plus >= EIGHTBIT
                {
                        z_memory = (value1 << shift_plus) | (value2 >> (BIT_32 - shift_plus));
                        bit_error = co_CompareBitpattern.compare(z_memory);
                        if(bit_error <= BitErrorMax)
                        {
                        synchronize_bit_slip(NEXTBYTE, value1, value2, (shift_plus - EIGHTBIT));
                                return bit_error;

                        }
                }

                if(shift_minus > ZEROBIT)
                {
                        z_memory = (value1 << shift_minus) | (value2 >> (BIT_32 - shift_minus));
                        bit_error = co_CompareBitpattern.compare(z_memory);
                        if(bit_error <= BitErrorMax)
                        {
                                synchronize_bit_slip(CURRENTBYTE, value1, value2, shift_minus);
                                return bit_error;
                        }
                }
                else
                {
                        if(shift_minus == ZEROBIT)
                        {
                                z_memory = value1;
                                bit_error = co_CompareBitpattern.compare(z_memory);
                                if(bit_error <= BitErrorMax)
                                {
                                synchronize_bit_slip(CURRENTBYTE, value1, value2, ZEROBIT);
                                        return bit_error;
                                }
                        }
                        else
                        {
                        if((getReadStatus() == true) || (index_on_char_array < FOURBYTE))
```

```
                        {
                                z_memory = getBufferReadMemory();

                                switch (index_on_char_array)
                                {
                                        case 1 :        z_memory = (z_memory << EIGHTBIT) |
                                          (get_long_value_from_read_memory(0) >> BIT_24);
                        setBufferReadMemory(z_memory); // ReadMemory mit diesem Wert belegen
                                        break;
                                        case 2 :        z_memory = (z_memory << BIT_16) |
                                         (get_long_value_from_read_memory(1) >> BIT_16);
                        setBufferReadMemory(z_memory); // ReadMemory mit diesem Wert belegen
                                        break;
                                        case 3 :        z_memory = (z_memory << BIT_24) |
                                       (get_long_value_from_read_memory(2) >> EIGHTBIT);
                        setBufferReadMemory(z_memory); // ReadMemory mit diesem Wert belegen
                                };
                                }
                                else
                                {//4 Bytes vor der Stelle, auf die IndexOnCharArray zeigt holen
                        z_memory = get_long_value_from_read_memory(getIndexOnCharArray() - FOURBYTE);
                        setBufferReadMemory(z_memory); // ReadMemory mit diesem Wert belegen
                                }

                        z_memory = (z_memory << (BIT_32 + shift_minus)) | (value1 >> (-shift_minus));
                                bit_error = co_CompareBitpattern.compare(z_memory);
                                if(bit_error <= BitErrorMax)
                                {
                        synchronize_bit_slip(LASTBYTE, value1, value2, (EIGHTBYTE + shift_minus));
                                        return bit_error;
                                }
                        }
                }
        }
 return FOUNDNOTHING; // Wenn innerhalb des Fensters kein Synchronwort gefunden wurde
}

void ct_MemoryAndFileManagment::synchronize_bit_slip(unsigned int mode, unsigned long value1,
unsigned long value2,

         unsigned int new_bit_shift)
{
        unsigned long number_bytes;
        unsigned int i;
        unsigned char char_value[8];
        unsigned char *p_write_memory;
        bool correct = false;

        char_value[7] = (unsigned char)(value2 >> EIGHTBIT);
        char_value[6] = (unsigned char)(value2 >> BIT_16);
        char_value[5] = (unsigned char)(value2 >> BIT_24);
        char_value[4] = (unsigned char) value1;
        char_value[3] = (unsigned char)(value1 >> EIGHTBIT);
        char_value[2] = (unsigned char)(value1 >> BIT_16);
        char_value[1] = (unsigned char)(value1 >> BIT_24);
        char_value[0] = (unsigned char)BufferReadMemory;

        if(!ReadStatus)
        {
                if(BeginToCopy == ReadMemoryWidth)
                        number_bytes = IndexOnCharArray + ONEBYTE;
                else
                        number_bytes = (IndexOnCharArray - BeginToCopy) + ONEBYTE;
        }
        else
        {
                correct = true;
                number_bytes = (MaxIndexCharOld - IndexOnCharArray) - ONEBYTE;
                p_write_memory = &p_WriteMemory[IndexWriteMemory - number_bytes];
        }

        switch (mode)
        {
                case CURRENTBYTE : if(correct)
                        {
                                for(i=0;i<number_bytes;i++)
                                {
                                        if(new_bit_shift != ZEROBIT)
```

```
                                {
                                        p_write_memory[i] = (char_value[i+1] <<
                  new_bit_shift) | (char_value[i+2] >> (EIGHTBYTE - new_bit_shift));
                                }
                                else
                                {
                                        p_write_memory[i] = char_value[i+1];
                                }
                        }
                }
                else
                {
                        synchronize_bytes(number_bytes);
                }
                BitShift = new_bit_shift;
        break;

        case NEXTBYTE :         if(correct)
                {
                        number_bytes--;
                        for(i=0;i<number_bytes;i++)
                        {
                                if(new_bit_shift != ZEROBIT)
                                {
                                        p_write_memory[i] = (char_value[i+2] <<
                     new_bit_shift) | (char_value[i+3] >> (EIGHTBIT - new_bit_shift));
                                }
                                else
                                {
                                        p_write_memory[i] = char_value[i+2];
                                }
                        }
                        IndexWriteMemory--;//Index muss im nachhinein geaendert werden
                }
                else
                {
                        synchronize_bytes(number_bytes);
                        BufferWriteMemory = char_value[2];
                        BeginToCopy++; //Kann hier nicht zu gross werden
                }
                IndexOnCharArray_plus_one();
                BitShift = new_bit_shift;
        break;

        case LASTBYTE :         if(correct)
                {
                        number_bytes++;
                        for(i=0;i<number_bytes;i++)
                        {
                                if(new_bit_shift != ZEROBIT)
                                {
                                        p_write_memory[i] = (char_value[i] <<
                   new_bit_shift) | (char_value[i+1] >> (EIGHTBIT - new_bit_shift));
                                }
                                else
                                {
                                        p_write_memory[i] = char_value[i];
                                }
                        }
                        IndexWriteMemory++;//Index muss im nachhinein geaendert werden
                }
                else
                {   // funktioniert!
                        synchronize_bytes(number_bytes);
                        BufferWriteMemory = char_value[0];
                        BeginToCopy--; //Kann hier minimal 1 sein
                }
                IndexOnCharArray_minus_one();
                BitShift = new_bit_shift;
        }
}

bool ct_FrameSynchronizer::search_mode(void)
{
        unsigned int status;
        bool end_of_file;

        do
```

```
	{
		end_of_file = search_bitpattern();
		if(end_of_file)// Dateiende erreicht
			return true;

		status = check_next_pattern();
		if(status == ENDOFFILE)// Dateiende erreicht
			return true;
	}
	while(status != 0);

return false; // gibt eine 0 zurueck
}

void ct_MemoryAndFileManagment::rescue_data_after_sync_lose_or_file_end(unsigned int
frame_length, bool end_of_file)
{
	unsigned long number_of_bytes;

// wurde in der Methode decreasFreeSpaceOnWriteMemory abgespeichert, ist das Ergebnis = 0
	number_of_bytes = (FrameNumber - FreeSpaceOnWriteMemory) * frame_length;

	if((number_of_bytes != 0) && (!end_of_file) && (!ReadStatus))
	{
		synchronize_bytes((IndexOnCharArray - BeginToCopy) + ONEBYTE);
	}
	write_synchronized_bytes_to_file(number_of_bytes);
}


unsigned long ct_MemoryAndFileManagment::get_and_reset_FramesWithErrors(void)
{
	unsigned long frames_with_errors;

	frames_with_errors = FramesWithErrors;
	AllFramesWithErrors += frames_with_errors;
    FramesWithErrors = 0;

return frames_with_errors;
}
```

9.2.4 reed_solomon_check.h

```
#ifndef __REED_SOLOMON_CHECK_H__
	#define __REED_SOLOMON_CHECK_H__

	#define SYMBOLWIDTH		8	//Bit
	#define ASMLENGTH		4	//Synchronwortlaenge
	#define CODEWORDLENGTH		255	//2^SYMBOLWIDTH-1
	#define INFORMATIONLENGTH	223
	#define PARITYLENGTH		32
	#define ARRAYSIZE		256	//2^SYMBOLWIDTH
	#define OFFSET			112

	class ct_ReedSolomonCheck
	{
		unsigned char p_PrimitivePolynomial[SYMBOLWIDTH + 1];
		unsigned char p_PowerOfB[ARRAYSIZE];
		int p_IndexOfB[ARRAYSIZE]; // Datentyp signed da -1 als Wert
		unsigned char **p_Blocks;  // Interleavingbloecke
		unsigned char *p_PseudoRandomSequence;
		unsigned char InterleavingDepth;

		void generate_prs(void);
		void generate_galois_field(void);
		unsigned char get_prn_code(unsigned int i){return p_PseudoRandomSequence[i];}
		unsigned int decode_reed_solomon(register unsigned char *array_to_check);

	public:
		ct_ReedSolomonCheck(unsigned int frame_length);
		~ct_ReedSolomonCheck(void);
		unsigned long check_on_errors(unsigned char* frames, unsigned long number);
	};
#endif;
```

9.2.5 reed_solomon_check.cpp

```
#include "reed_solomon_check.h"
#include <iostream>
	using namespace std;

ct_ReedSolomonCheck::ct_ReedSolomonCheck(unsigned int frame_length)
{
	unsigned int k;
	InterleavingDepth = (frame_length - ASMLENGTH) / CODEWORDLENGTH;

	p_Blocks = new unsigned char*[InterleavingDepth];
	for(k=0;k<InterleavingDepth;k++)
		p_Blocks[k] = new unsigned char[CODEWORDLENGTH];

	p_PseudoRandomSequence = new unsigned char[frame_length - ASMLENGTH];

	p_PrimitivePolynomial[0] = 1;// 1+
	p_PrimitivePolynomial[1] = 1;// x+
	p_PrimitivePolynomial[2] = 1;// x^2+
	p_PrimitivePolynomial[3] = 1;// x^3+
	p_PrimitivePolynomial[4] = 1;// x^4+
	p_PrimitivePolynomial[5] = 0;
	p_PrimitivePolynomial[6] = 1;// x^6+
	p_PrimitivePolynomial[7] = 0;
	p_PrimitivePolynomial[8] = 1;// x^8

	generate_galois_field();
	generate_prs();
}

ct_ReedSolomonCheck::~ct_ReedSolomonCheck(void)
{
	unsigned int k;

	for(k=0;k<InterleavingDepth;k++)
	{
		if(p_Blocks[k] != NULL)
			delete [] p_Blocks[k];
	}
	delete [] p_Blocks;
	delete [] p_PseudoRandomSequence;
}

void ct_ReedSolomonCheck::generate_galois_field(void)
{
	unsigned int i, mask;
	unsigned char l7, l6, l5, l4, l3, l2, l1, l0;
	unsigned char bit7, bit6, bit5, bit4, bit3, bit2, bit1, bit0;

	mask = 1;
	p_PowerOfB[SYMBOLWIDTH] = 0;
	for(i=0;i<SYMBOLWIDTH;i++)
	{
		p_PowerOfB[i] = mask;

		if(p_PrimitivePolynomial[i] != 0)
			p_PowerOfB[SYMBOLWIDTH] ^= mask;

		mask <<= 1;
	}

	mask >>= 1;
	for(i=SYMBOLWIDTH+1;i<CODEWORDLENGTH;i++)
	{		// mask = 128  1 0 0 0 0 0 0 0 (Bitdarstellung)
	if(p_PowerOfB[i-1] >= mask)// wenn das achte Bit eine 1 ist
		p_PowerOfB[i] = p_PowerOfB[SYMBOLWIDTH] ^ ((p_PowerOfB[i-1] ^ mask) << 1);
	else//weil int, wird Bit 8 zu 0 gesetzt, sonst wuerde die 1 an Stelle Bit 9 rutschen!
		p_PowerOfB[i] = p_PowerOfB[i-1] << 1;
	}
/************** B in die Dual-Basis-Darstellung tranformieren ************************/
	// Basis der Transformation
	l7 = p_PowerOfB[70];  // 01111011
```

```
        l6 = p_PowerOfB[200]; // 01111001
        l5 = p_PowerOfB[178]; // 00101011
        l4 = p_PowerOfB[229]; // 00111111
        l3 = p_PowerOfB[48];  // 00001001
        l2 = p_PowerOfB[167]; // 10000111
        l1 = p_PowerOfB[8];   // 01011111
        l0 = p_PowerOfB[40];  // 00110111

        for(i=0;i<CODEWORDLENGTH;i++)
        {
                bit7 = p_PowerOfB[i] & 1;
                bit6 = (p_PowerOfB[i] >> 1) & 1;
                bit5 = (p_PowerOfB[i] >> 2) & 1;
                bit4 = (p_PowerOfB[i] >> 3) & 1;
                bit3 = (p_PowerOfB[i] >> 4) & 1;
                bit2 = (p_PowerOfB[i] >> 5) & 1;
                bit1 = (p_PowerOfB[i] >> 6) & 1;
                bit0 = (p_PowerOfB[i] >> 7) & 1;

p_PowerOfB[i] = bit0*l0 ^ bit1*l1 ^ bit2*l2 ^ bit3*l3 ^ bit4*l4 ^ bit5*l5 ^ bit6*l6 ^ bit7*l7;
                p_IndexOfB[p_PowerOfB[i]] = i;
        }
        p_IndexOfB[0] = -1;
}

void ct_ReedSolomonCheck::generate_prs(void)
{
        unsigned char pr_register = 0xFF; //Initialisierung alles Einsen
        unsigned char bit1, bit3, bit5, bit8;
        unsigned int k, l, frame_length;

        frame_length = InterleavingDepth * CODEWORDLENGTH;
        p_PseudoRandomSequence[0] = pr_register;

        for(k=1;k<frame_length;k++)
        {
                for(l=0;l<8;l++)
                {
                        bit8 = (pr_register >> 7) & 1;
                        bit5 = (pr_register >> 4) & 1;
                        bit3 = (pr_register >> 2) & 1;
                        bit1 = pr_register & 1;

                        pr_register = pr_register << 0x01;
                        pr_register |= (bit1 ^ bit3 ^ bit5 ^ bit8);
                }
                p_PseudoRandomSequence[k] = pr_register;
        }
}

unsigned int ct_ReedSolomonCheck::decode_reed_solomon(register unsigned char *array_to_check)
{
        register unsigned int i, j;
        register unsigned int syn_error = 0;
        register unsigned int syndrome;
        register int transformed_array[CODEWORDLENGTH];

        for(i=0;i<CODEWORDLENGTH;i++)// Wert in Indexform bringen
                transformed_array[i] = p_IndexOfB[array_to_check[i]];

        for(i=OFFSET;i<=143;i++)
        {
                syndrome = 0;
                for(j=0;j<CODEWORDLENGTH;j++)
                {
                        if(transformed_array[j] != -1) // Wert != 00000000 (binaer)
                        syndrome ^= p_PowerOfB[(transformed_array[j] + i*j) % CODEWORDLENGTH];
                }
                if(syndrome != 0) // Syndrome ungleich 0, ist ein Fehler aufgetretten
                        syn_error = 1;
        }
return syn_error;
}

unsigned long ct_ReedSolomonCheck::check_on_errors(register unsigned char *frames,
                                                   register unsigned long number)
{
        register unsigned long k;
```

```
        register unsigned int l, i, z;
        register unsigned int status;
        register unsigned int interleaving_depth;
        register unsigned int frame_status;
        register unsigned long all_frames = 0;

        interleaving_depth = InterleavingDepth;

        for(k=0;k<number;k++)
        {
                frames = &frames[ASMLENGTH];// Synchronwort auslassen
                i=0;
                for(z=0;z<CODEWORDLENGTH;z++)
                {
                        for(l=0;l<interleaving_depth;l++)
                        {
                         // Deinterleaving         Derandomizing
                                p_Blocks[l][z] = *frames ^ p_PseudoRandomSequence[i];
                                frames++;
                                i++;
                        }
                }
                frame_status = 0;
                for(l=0;l<interleaving_depth;l++)
                {
                        status = decode_reed_solomon(p_Blocks[l]); //R-S Check
                        if(status != 0)
                                frame_status = 1;
                }
                if(frame_status != 0)
                        all_frames++;
        }
return all_frames;
}
```

Versicherung über die Selbständigkeit

„Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §25(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht."

Hamburg, den 28. Februar 2008

Christian Koch