Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von
Carsten Polenz
aus Marl

Juni 1996

Aufbau eines Prototypen für die Auslese der MSGC's bei HERA-B

Die Diplomarbeit wurde von Carsten Polenz ausgeführt am
Physikalischen Institut der Universität Heidelberg
unter der Betreuung von
Herrn Prof.Dr. Franz Eisele

Inhaltsverzeichnis

Kapitel 1

Einleitung

Am Elektron-Proton-Speicherring HERA, dessen Betrieb 1992 aufgenommen wurde, können Elektronen und Protonen mit einer Schwerpunktsenergie von ≈ 300 GeV zur Kollision gebracht werden. Hierbei verwendet HERA einen Elektronenstrahl mit ≈ 30 GeV und einen Protonenstrahl mit ≈ 820 GeV Schwerpunktsenenergie. Bisher sind bei am HERA-Speicherring 3 Experimente installiert, von denen zwei, H1 und ZEUS, Elektronen-Protonen-Kollisionen vermessen, während HERMES allein den Elektronenstrahl nutzt.

Als viertes Experiment ist das HERA-B Experiment vorgesehen, das 1998 in Betrieb gehen soll. Bei HERA-B wird der Protonenstrahl zur Erzeugung von B-Mesonen benutzt werden, um damit die CP-Verletzung im $B^0 - System$ genauer zu untersuchen.

Der Aufbau des Experimentes stellt an die Kollaboration aufgrund der geplanten, sehr hohen Ereignisrate hohe Anforderungen, sowohl an den Detektor selbst, als auch an die Datenerfassung.

In Heidelberg werden z.Zt. sowohl der Silizium Vertex Detektor (MPI K Heidelberg, u.a.) als auch der Inner Tracker (Univ. Heidelberg, Univ. Siegen, Univ. Zürich) mitentwickelt.

Gegenstand dieser Arbeit war die Entwicklung eines Prototypen der Auslesesteuerung (Front-End Driver Board) für diese beiden Subdetektoren mit dem Ziel, ein funktionstüchtiges Auslesesystem für die Detektortests im Sommer 1996 bereitzustellen.

In Kapitel 2 wird das HERA-B Experiment und dessen Datenerfassung vorgestellt. Kapitel 3 diskutiert das Konzept des Front-End Driver. Anschließend wird in Kapitel 4 die zentrale Steuereinheit, der Controllerbaustein, besprochen, dessen Entwicklung wesentlicher Bestandteil der vorliegenden Arbeit ist. Kapitel 5 schließt die Arbeit mit der Vorstellung der Ergebnisse der Inbetriebnahme des Front-End Driver ab.

Kapitel 2

Das HERA-B Experiment und der Aufbau der HERA-B DAQ

Neben der Suche nach dem Ursprung von Masse bei Elementarteilchen (Higgs-Mechanismus), steht eine weitere Frage im Mittelpunkt des physikalischen Interesses:
Ist das Standardmodell korrekt hinsichtlich seiner Beschreibung der CP-Verletzung?
Um die Erkenntnisse, die man aus der Analyse des K^0-Systems gewonnen hat, zu ergänzen, wollen mehrere Experimente in den nächsten Jahren das B^0-System genauer untersuchen. Eines dieser Experimente ist HERA-B, das im Februar 1995 genehmigt wurde und im Jahr 1998 seinen vollen Betrieb aufnehmen soll. Das Experiment wird von einer Kollaboration verschiedener deutscher Universitäten und internationaler Forschungseinrichtungen[1] und Hochschulen getragen. Bei HERA-B wird man sich im B^0-System auf die Zerfälle

$$B^0 \to J/\psi K_s^0$$

bzw.

$$\overline{B}^0 \to J/\psi K_s^0$$

konzentrieren (s.Abbildung 2.1). Diese Zerfälle, die man als den "Gold-Plated Mode" bezeichnet, bieten sich durch ihre einfache Signatur an, die eine effiziente Selektion der korrekten Ereignisse erlaubt. Der Abbildung entnimmt man außerdem, daß das J/ψ weiter in ein Leptonenpaar l^+l^- und das K_s^0 in ein Pionenpaar $\pi^+\pi^-$ zerfällt. Das Maß für die Messung

[1] z.B.: UCLA, University of Massachusetts, Institute of Theoretical and Experimental Physics, Universität Siegen, Universität Zürich u.s.w.

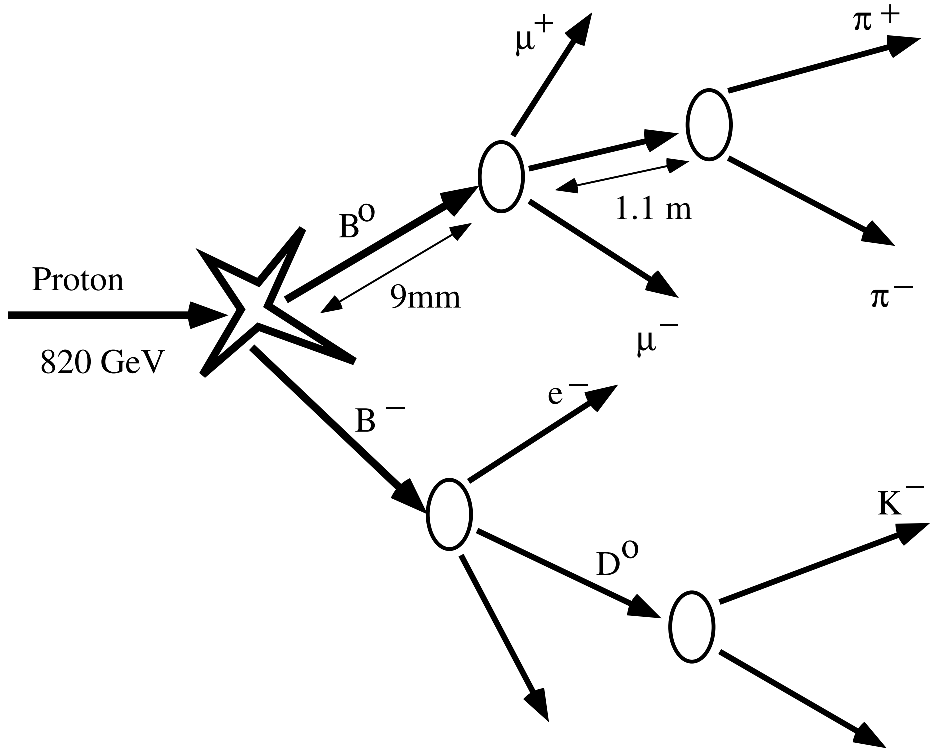

Abbildung 2.1: Die Abbildung zeigt ein sogenanntes Gold-Plated-Event, auf dem das Hauptaugenmerk der physikalischen Analyse bei HERA-B liegt. [DESIGN]

der CP-Verletzung in diesem Experiment ist das zeitabhängige Verhältnis der Übergangsrate von B^0 bzw. $\overline{B}^0$ in den gleichen Endzustand $J/\psi K_s^0$. Es muß daher auch festgestellt werden, ob ein B^0 oder ein $\overline{B}^0$ zerfallen ist. Dieses gelingt effizient durch die Bestimmung der Ladung des Leptons in den semileptonischen Zerfällen $b \rightarrow c + e^-\overline{\nu_e}$ bzw. $b \rightarrow \overline{c} + e^+\overline{\nu_e}$. Das experimentelle Ziel ist die Bestimmung des Winkels β des sog. Unitaritätsdreiecks auf eine Genauigkeit von

$$\delta sin\beta \leq 0.05.$$

Einige kurze Überlegungen erläutern die Konsequenzen aus dieser Forderung:

Nimmt man die Verzweigungsverhältnisse, den totalen, inelastischen Wirkungsquerschnitt und die Rekonstruktionseffizienz des Detektors zusammen, ergibt sich für den Gold-Plated Mode eine Unterdrückung um den Faktor $3 * 10^{-12}$, d.h. im Durchschnitt findet man unter $3 * 10^{12}$ Ereignissen einen interessanten Zerfall. Zur Erlangung der obengenannten Genauigkeit werden 3000 Ereignisse benötigt, insgesamt muß man also 10^{15} Zerfälle über die gesamte Betriebsdauer des Experimentes betrachten. Dieses ist nur mit einer hohen Wechselwirkungsrate ($\geq$ 30 MHz) zu erreichen, was allerdings eine effektive Datenselektion verlangt, aber als weitere Komplikation sehr hohe Teilchendichten und damit eine massive Strahlenbelastung des Detektors mit sich bringt.

2.1 Das Detektorsystem

HERA-B wird den Protonenstrahl des HERA-Speicherringes[2] benutzen. Es handelt sich hierbei um ein sog. Fixed-Target Experiment, bei dem die primären Wechselwirkungen, d.h. die Erzeugung von u.a. B-Mesonen, durch Metalldrähte, die in den äußeren Bereich des Protonenstrahles (Halo) hineinreichen, verursacht werden. Der vollständige Detektor ist in Abbildung2.2 dargestellt.

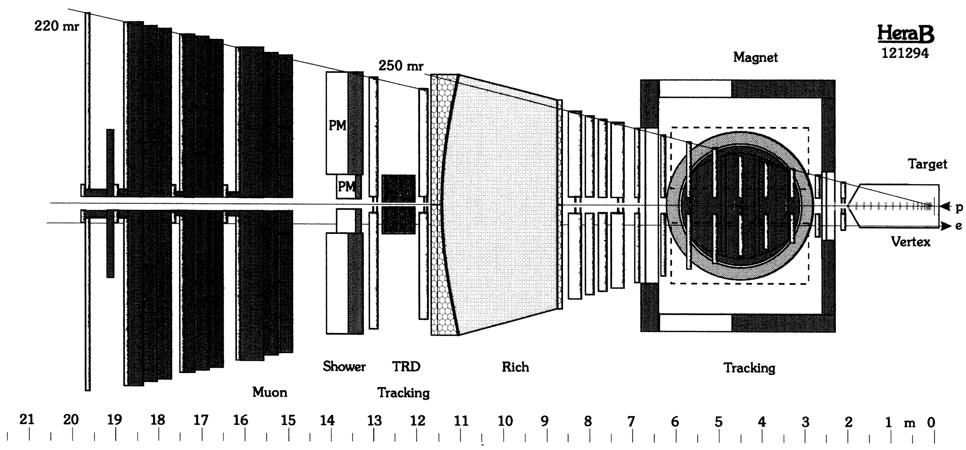

Abbildung 2.2: Das HERA-B Spektrometer in Gesamtüberblick. [DESIGN]

Er besteht aus den folgenden Hauptkomponenten:

- Ein Vertex Detektor (Vertex) aus Siliziumstreifendetektoren zur Bestimmung des genauen Wechselwirkungspunktes und des Sekundärvertices, in dem das B-Meson weiter zerfällt.
- Ein magnetisches Spektrometer aus einem Dipolmagneten und Spurkammern, zur Messung der Teilchenimpulse.

[2]Hadron-Elektron-Ring-Anlage: einziger ep-Speicherring, der im Juni 1992 am DESY Hamburg in Betrieb ging.

- Ein Trackingsystem (Inner Tracker und Outer Tracker), zur Spurrekonstruktion der Sekundärteilchen.
- Ein RICH-Zähler (Ring Image Cherenkov) zur Unterscheidung von Protonen, Kaonen und Pionen.
- Ein Elektromagnetisches Kalorimeter zur Bestimmung von Elektronenpaaren
- Einen Myonen Detektor zur Identifizierung von Myonen-Paaren

Das Spurkammersystem gliedert sich in Bereiche unterschiedlicher Granularität. Der innere Bereich, der zwischen 6 und 25 cm vom Strahl entfernt ist (Inner Tracker), wird von MSGC's[3] abgedeckt, die unter anderem am Physikalischen Institut in Heidelberg mitentwickelt werden, der äußere Bereich jenseits von 25 cm Abstand zum Strahl, das Outer Tracker System, besteht aus Honeycomb Driftkammern unterschiedlicher Zellengröße. Damit wird erreicht, daß die sog. Occupancy, die Teilchendichte pro Kammerkanal, im inneren Detektorbereich weitgehend gleich dem äußeren Detektorbereich ist.

2.2 Das Triggerkonzept bei HERA-B

Das physikalische Ziel des Experimentes birgt direkte Konsequenzen für den Aufbau und die Struktur der Datenaufnahme. Zur Bestimmung des Winkels β auf die gewünschte Genauigkeit von $\delta sin\beta \approx 0.05$ müssen 3000 Ereignisse des Zerfalls $B^0 \rightarrow J/\psi K^0$ innerhalb der gesamten Experimentierdauer akkumuliert werden. Insgesamt gibt es aber eine Unterdrückung dieser Ereignisse um den Faktor $3 * 10^{-12}$, d.h für die Ansammlung von 3000 Ereignissen müssen insgesamt 10^{15} Wechselwirkungen registriert und ausgewertet werden. Diese extrem hohe Datenrate hat unmittelbare Konsequenzen für die Datenaufnahme:

- Man benötigt eine sehr hohe Datenrate, um überhaupt innerhalb einer akzeptablen Experimentierzeit, z.B. 5 Jahre, genügend Daten zu sammeln. Nimmt man die schon erwähnten 5 Jahre als Ausgangspunkt, so läßt sich leicht errechnen, daß man eine durchschnittliche Ereignisrate von 20 MHz benötigt.
- Diese vorgenannte Bedingung hat zur Folge, daß eine effektive, mehrstufige Datenselektion zu erfolgen hat (Triggerarchitektur), die mehrere, gleichzeitige Ereignisse im Detektor [4] verarbeiten kann.

[3] Micro Strip Gaseous Chambers

[4] Man rechnet bei HERA-B mit 4 - 6 Ereignissen pro Bunchcrossing, d.h. mit 30 Millionen Ereignissen pro Sekunde

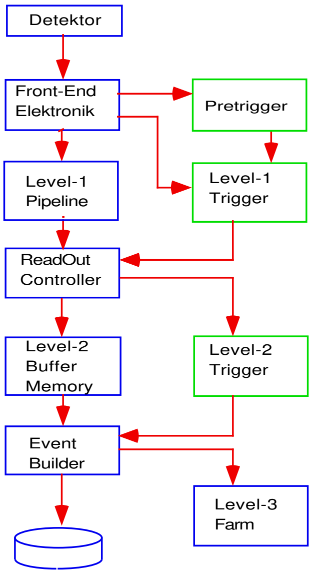

Abbildung 2.3: Die hohen Anforderungen an die Datenreduktion führen zu einer dreistufigen Triggerarchitektur. [DESIGN]

- Desweiteren wird man es sich während der Betriebszeit nicht leisten können, die Datennahme für die Zeit der Auslese des Systems anzuhalten, da jede Totzeit des Detektors zwangsläufig zu ungewolltem Datenverlust und damit zu einer verlängerten Experimentierzeit führen wird.

In Abbildung 2.3 ist das grundsätzliche Konzept und die Struktur der Datenreduktion dargestellt. Mit jedem Bunchcrossing [5] des Protonenstrahles werden die gesamten Detektordaten von der Front-End Elektronik ausgelesen und in einer sog. Pipeline-Struktur zwischengespeichert [6].

Um eine Datenreduktion auf ein Niveau, welches die Massenspeicherung auf Magnetbänder erlaubt, zu erreichen, hat man ein dreistufiges Selektionssystem für die Detektordaten, das On-Line durchgeführt wird, vorgesehen.

Das Prinzip jeder dieser Selektionstufen ist gleich. Zuerst werden die gesamten Eingangsdaten für diese Stufe in einen Zwischenspeicher eingelesen. Dieser Speicher muß groß genug gewählt sein, um die gesamten Daten für die durchschnittliche Dauer der Triggerentscheidung vorzuhalten.

---

[5] Bei HERA beträgt die Bunchcrossingfrequenz 10,416 MHz

[6] Dieser Level-1 Zwischenspeicher absolviert 2 Aufgaben:

- normale Pipeline, d.h. ein Ringspeicher, der Daten sequentiell einliest und nach einer Verzögerung von Taktrate * Pipeline = Verzögerungszeit überschreibt.
- gezielter Zugriff auf einzelne Speicherelemente unabhängig von der augenblicklichen Position des Schreibezeigers.

In der ersten Triggerstufe bei HERA-B rechnet man mit einer Dauer von ungefähr 10 Mikrosekunden für die Triggerentscheidung, und so hat man einen Zwischenspeicher von 128 Datensätzen, was einer Speicherdauer von ca. 12,3 Mikrosekunden entspricht, vorgesehen.

Ein bestimmter Teil der Detektordaten, z.B. in der 1. Triggerstufe ausgewählte Detektorelemente, wird separat zu der jeweiligen Triggereinheit geführt, die die Selektionsentscheidung fällt.

Nach dieser Entscheidung werden die noch interessierenden Daten aus dem Zwischenspeicher ausgelesen und der nächsten Verarbeitungsstufe zugeführt, die restlichen Daten verworfen.

Die verschiedenen Prätrigger

Durch die sogenannnten Prätrigger werden die **ROI's** ("regions of interest") bestimmt, die dann nachfolgend durch den First Level Trigger Spuralgorithmus mit Informationen aus ausgewählten Kanälen des Tracking Systems weiterprozessiert werden. Insgesamt gibt es bei HERA-B drei verschiedene Typen von Prätriggern, die jeweils verschiedene Signaturen des Zerfalls $B^0 \to J/\psi K^0$ ausnutzen.

Im elektromagnetischen Kalorimeter werden Elektronenkandidaten erkannt, die sog. "Padchambers" des Myonen-Systems, d.h. die Kammerebenen **MU3** und **MU4** detektieren Myonenkandidaten, und Hadronen mit großem Transversalimpuls werden durch drei Padchamberebenen (**PT1**, **PT2** und **PT3**) entdeckt (s. Abbildung 2.4).

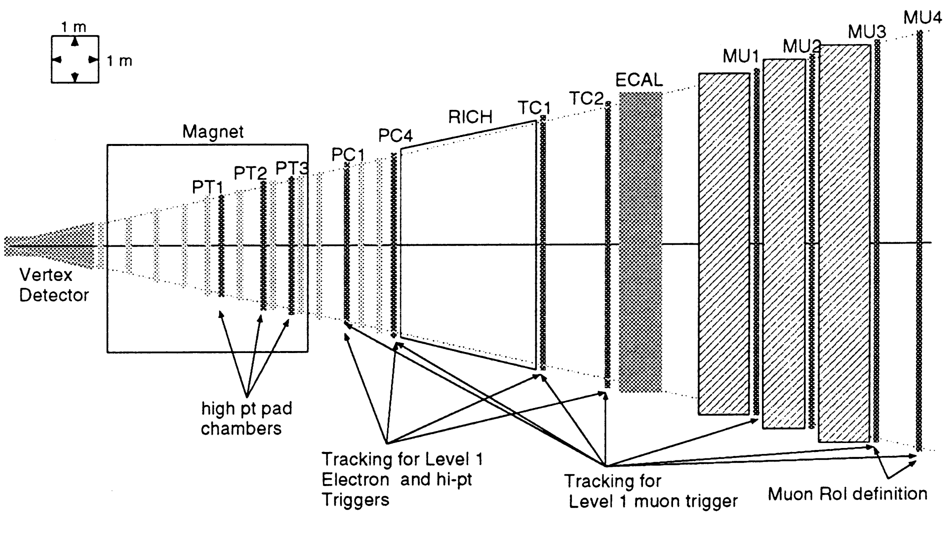

Abbildung 2.4: Dieses ist eine Flächenansicht des HERA-B Spektrometers. Die Elemente, die für den Prätrigger und die 1.Triggerstufe benutzt werden, sind markiert. [MEDINNIS]

Die einzelnen Prätrigger liefern an den First Level Trigger geometrische Informationen über die Größe und die Position der einzelnen ROI's. Desweiteren ermitteln sie erste, wenn auch ungenaue, Impulsinformationen über die Kandidaten. Die durchschnittliche Anzahl von ROI's, die pro Bunchcrossing gefunden werden, beträgt 5.

Der elektronische Prätrigger basiert vollständig auf der Messung der Pulshöhe des elektromagnetischen Kalorimeters. Die Triggerentscheidung basiert dann auf Clustern aus 3x3 Zellen, wobei die zentrale Zelle mindestens die Hälfte der im Cluster deponierten Gesamtenergie akkumuliert haben muß.

Der Muonentrigger wertet Hits in den letzten beiden Muonenkammern **MU3** und **MU4** aus. Ausgehend von jedem Treffer in der hinteren Kammer **MU4**, unter Berücksichtigung der Bedingung, daß das Teilchen vom Target kommmen muß, sucht er die korrespondierende Zelle in der Kammer **MU3**, die ca. 1m vor der Kammer **MU4** liegt.

Der Hadronen Prätrigger schließlich bringt jede Zelle aus der Kammerebene **PT1** mit jeweils 3 in horizontaler Reihe nebeneinander liegenden Zellen aus **PT2** und 2 Zellen aus **PT3** in Verbindung [7].

Der First Level Trigger

Die Informationen aus den jeweiligen Prätriggern werden im First Level Trigger mit zusätzlichen Daten aus ausgewählten Kammern des Outer und des Inner Trackers in Verbindung gebracht. Aus der linken Hälfte von Abbildung 2.2 werden die einzelnen Schritte der Spurrekonstruktion deutlich:

1. Die Spurrekonstruktion wird, bei Myonen von der Kammer **MU4**, beim Elektronen vom **ECAL** ausgehend, sukzessive durch die Superlayer **TC2**, **TC1**, **PC4** und **PC1** durchgeführt. Im Outer Tracker wird geprüft, welche Kammern gleichen Stereowinkels eines Superlayers angesprochen haben. Die Granularität der Triggerinformation beträgt 5 bzw. 10 mm. Im Inner Tracker wird hingegen ein OR aus 4 benachbarten Streifen gebildet, d.h. die Streifenbreite beträgt 1,2mm. Wenn in einem Superlayer eine korrekte Hitinformation gefunden wurde (s.Abbildung 2.2), werden die Spurparameter aktualisiert und an die nächste Superlage weitergegeben. Ansonsten wird für diesen Trackkandidaten die Rekonstruktion abgebrochen.

2. Danach werden die Spurparameter dann in Impulsinformationen übersetzt, unter der Annahme, daß die Spuren vom Target stammen.

3. Anschließend wird, basierend auf den gefundenen Trackparametern, geschnitten.

[7] Dieser Trigger ist erst vor kurzem hinzugefügt worden, da Simulationen andeuten, daß die Effizienz und die Unterdrückung akzeptabel sind

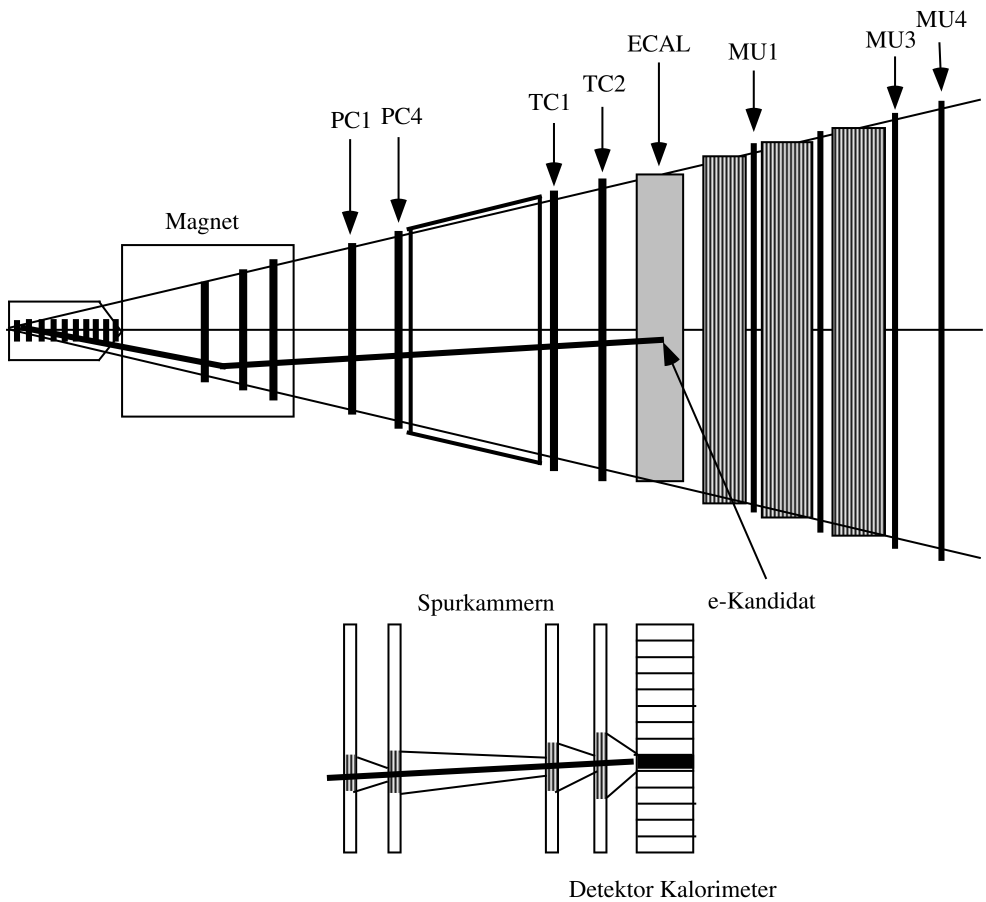

Abbildung 2.5: Dieses Bild zeigt das Konzept für den First Level Trigger bei HERA-B. [DESIGN]

4. Die gesamte Information wird jetzt gesammelt und zu Spurpaaren kombiniert, um die invariante Masse des jeweiligen Teilchenpaares zu bestimmen.

5. Auf diesen Daten basierend können jetzt verschiedene Triggerentscheidungen gefällt werden.

Der Second Level Trigger

Die Strategie für die 2.Triggerstufe besteht aus der geometrischen Verlängerung der ROI's zum Target hin und der darauf aufbauenden Verfeinerung dieser Zonen, die vom First Level Trigger generiert wurden. Die dazu benötigten Schritte sind in Abbildung 2.6 dargestellt:

- **FLT Spur Refit**: Mittels eines Kalman-Filter Algorithmus und unter Ausnutzung der relevanten Driftzeiten und Informationen zusätzlicher

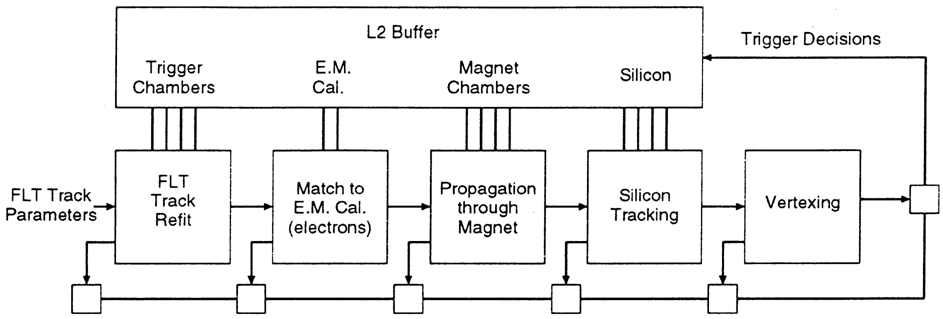

Abbildung 2.6: Die logische Struktur des SLT. [MEDINNIS]

Spurkammern [8] wird die vom FLT generierte Spur nochmals gefitted. Hierbei sollen insbesondere "Ghost Tracks" eliminiert werden.

- **Anpassung an das ECAL**: Aufbauend auf die vorhergegangene Verbesserung der Spurparameterinformationen kann jetzt für die Elektronenkandidaten die Anpassung der interessanten Spuren an die Zentren der ECAL Hits vorgenommen werden.

- **Verbindung zum Vertex Detektor**: Um die Informationen des Silizium Trackings nutzen zu können, müssen die ROI's weiter eingegrenzt werden. Dieses geschieht, indem die vorhandenen Spuren, unter Ausnutzung aller zum Target hin gelegenen Superlayer des Tracking Systems, durch den Magneten hin zum Silizium Vertex Detektor verfolgt werden.

- **Silizium Tracking**: Jetzt nimmt man die Informationen aus dem Silizium Vertex Detektor hinzu, um die Spuren zum Target hin zu verfeinern.

- **Vertexing**: In diesem, dem letzten Schritt, wird die Qualität der Spurrekonstruktion derart überprüft, daß man Triggerspuren, die von unterschiedlichen Vertices stammen, voneinander trennt. Desweiteren ist es zu diesem Zeitpunkt möglich, auf die Bedingung Targetposition $\neq$ Vertexposition zu schneiden, falls eine größere Unterdrückung benötigt wird.

Zusammenfassung

In Tabelle 2.1 ist eine kurze Zusammenfassung der verschiedenen Triggerstufen wiedergegeben.

[8] Es werden für die zweite Triggerstufe alle Inner Tracker Kanäle und auch die gesamte Outer Tracker Information genutzt

Level	Funktion	Time Scale	Input Rate
0	Prätrigger, definieren ROI's aus: • e.m. Kalorimeter • Muonen Filter • Pad Chambers	1μsec	10 MHz
1	• Finde Spuren hinter dem Magneten die mit ROI's assoziierbar sind • Schneide auf die Zwei-Teilchen Masse oder auf p_t	10 μ sec	10 MHz
2	Verfeinere die Spuren aus der 1.Triggerstufe: • Füge die Driftzeit hinzu • Verlängere die Spurrekonstruktion durch den Magneten und den Silizium Tracker • Überprüfe den Vertex für 2-Spur Trigger	10 msec	50 kHz
3	• Vollständige Spurrekonstruktion und Vertex Fit, korrigiere Vielfachstreuung • Verfeinere die Massen, Vertex und p_t Schnitte, führe Teilchenidentifizierung durch	30 msec	2 kHz
4	Vollständige Ereignisrekonstruktion. Kalibration, Akzeptiere für die Physikanalyse	2 sec	100 Hz

Tabelle 2.1: Eine kurze Zusammenfassung des Triggersystems bei HERA-B. [MEDINNIS]

Der First Level Trigger, wie auch die einzelnen Prätrigger, haben eine Dateneingangsrate von 10 MHz. (s. Abbildung 2.3). Die Tiefe der Pipeline, die die Detektordaten zwischenspeichert, beschränkt die maximale Triggerlatency auf $12,3\mu s$, die im Durchschnit aber nur 10 Mikrosekunden betragen soll.

Der Second Level Trigger hat eine durchschnittliche Dateneingangsrate von 50 kHz und eine mittlere Latency von 10msec. Die Triggerrate soll auf 2kHz reduziert werden.

Nach der dritten Triggerstufe, die 30 msec für ihre Entscheidung benötigt, schreibt die Datenerfassung die interessanten Ereignisse mit einer Rate von 100 Hz auf Band. Für die vollständige Ereignisrekonstruktion werden in der 4.Stufe dann 2 sec benötigt.

Der Tabelle entnimmt man weiterhin, daß in der ersten Stufe eine Reduktion um den Faktor 200, in der zweiten Stufe um den Faktor 25 und in der dritten Stufe um den Faktor 20, insgesamt also für die gesamte On-Line Analysekette eine Reduktion der behandelten Daten um den Faktor 10^5 anvisiert wird, bevor die Daten zur Off-Line Physikanalyse auf Band geschrieben werden.

Desweiteren ist ersichtlich, daß die Triggerentscheidungen der jeweiligen Stufe auf einer immer größer werdenden Datenbasis, d.h. unter Ausnutzung von immer mehr Detektorelementen, gefällt werden, je weiter sie in der Auslesekette voranschreiten. In der dritten und vierten Stufe werden dann komplette Ereignisse zur Entscheidungsfindung herangezogen.

2.3 Die gemeinsame Architektur der DAQ

Motivation für eine gemeinsame Architektur der DAQ

Detektortyp		Anzahl Kanäle
SVD	Analoge Pulshöhen (FADC's)	$\approx$ 170.000
Inner Tracker	“	$\approx$ 140.000
Outer Tracker	Driftzeiten (TDC's)	$\approx$ 100.000
RICH	Hits (Diskriminator)	$\approx$ 160.000
ECAL	Analoge Pulshöhen (14 Bit ADC's)	$\approx$ 6.000
Muon	Hits (Diskriminator)	$\approx$ 30.000
		$\approx$ 600.000

Tabelle 2.2: Übersicht über die Anzahl der Daten kanäle bei HERA-B. [DESIGN]

In der Tabelle 2.2 ist zu erkennen, daß HERA-B auch durch die große Anzahl an Datenkanälen besondere Anforderungen an die Elektronik stellt. Um dieses System beherrschen und auch warten zu können, hat man sich entschlossen, schon sehr frühzeitig im Datenfluß zu einer gemeinsamen Architektur der Datenaufnahme überzugehen.

Das Konzept dieser Hardwarearchitektur ist in Abbildung 2.7 dargestellt. Die Abbildung spiegelt in den drei vertikalen Spalten den Informationsfluß für die Kontrolldaten zur Steuerung der Datenaufnahme (links), für die Triggerentscheidungen (mitte) und für den Fluß der Detektordaten

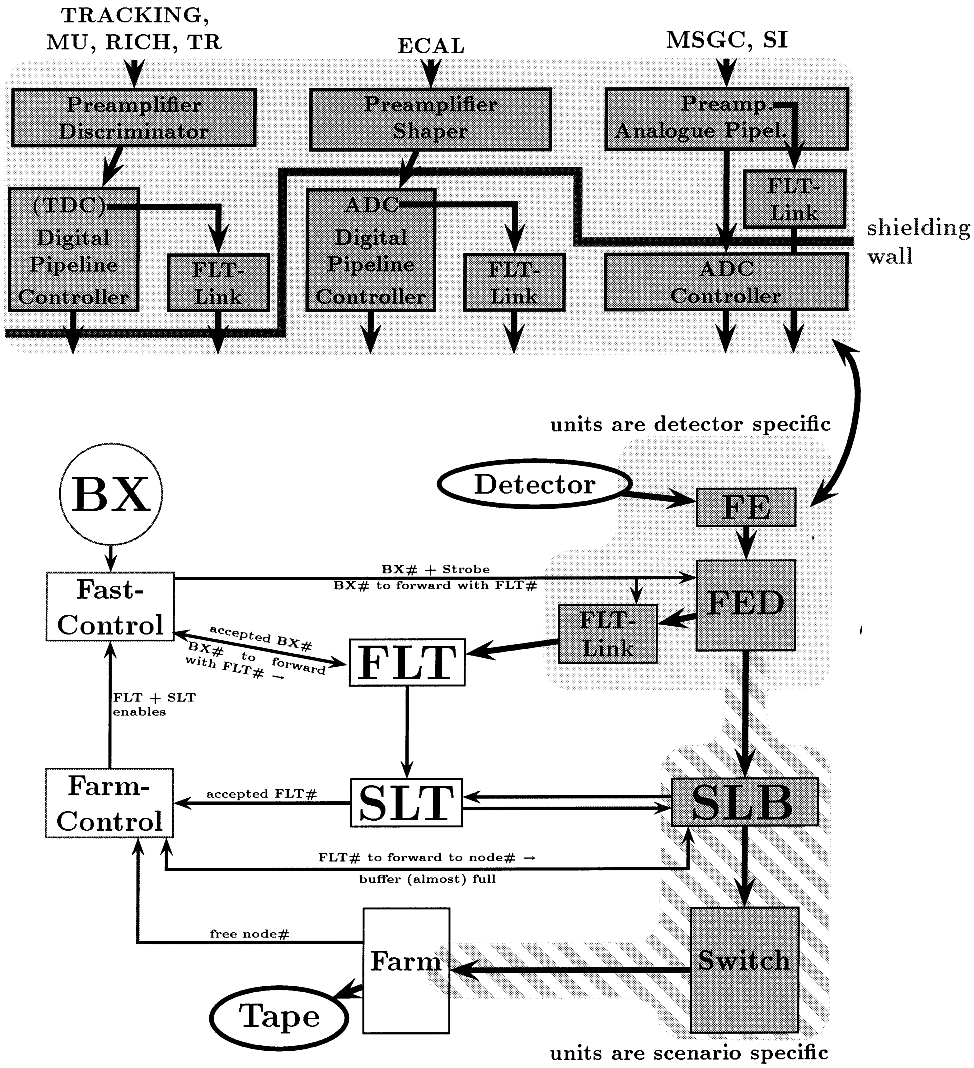

Abbildung 2.7: Diese Abbildung gibt die gemeinsame Datenaufnahme von HERA-B wieder. [COMMON]

(rechts) wider. Betrachtet man hingegen die horizontalen Verbindungen, so sind die Kontroll-, Trigger- und Datenspeichereinheiten für die jeweilige Verarbeitungsstufe aufgezeichnet.

Man erkennt, daß für die Speicherung der Daten einerseits detektorspezifische Einheiten (Front-End Elektronik, Auslesechips), andererseits detektorglobale Elemente (Second Level Buffer, Farm) geplant sind.

Der Front-End Driver (**FED**) stellt die Schnittstelle zwischen den verschiedenen Komponenten der gemeinsamen Datenerfassung und der detektorspezifischen Elektronik dar:

- Von der Fast Control (**FCS**) empfängt das FED die zur Steuerung der Datenauslese benötigten Daten. FLT# ist die fortlaufende Triggernummer, BX# ist die fortlaufende aktuelle Bunchcrossingnummer und BX-FLT# ist die Bunchcrossingnummer zu der getriggerte Ereignis geschah.
- Die Detektorelektronik (Front-End) liefert an das FED die Detektordaten akzeptierter Ereignisse.
- Die akzeptierten Ereignisse werden in einem vereinheitlichten Datenausgabeformat an den Second Level Buffer (**SLB**) versendet.

Kapitel 3

Der Front End Driver für die Mikrostreifendetektoren

Das hier diskutierte Front-End-Driver Board implementiert bei HERA-B die Schnittstelle zwischen detektorspezifischer Ausleseelektronik des SVD und des Inner Tracker und der globalen Datenaufnahme des Experimentes. Sowohl der Silizium Vertex Detektor (MPI K Heidelberg, u.a.), als auch der Inner Tracker, mit sog. MSGC's [1] (Univ. Heidelberg, Univ. Siegen und Univ Zürich) werden in Heidelberg mitentwickelt. Da bei beiden Detektoren Signalamplituden analog ausgelesen werden, lag es nahe, dieselbe Ausleseelektronik für beide Komponenten zu wählen und das Front-End-Driver Board auch in Heidelberg zu entwickeln. Dieses sollte von der HERA-B Gruppe am Physikalischen Institut in Zusammenarbeit mit der dortigen Elektronikwerkstatt durchgeführt werden.

Ruft man sich den Platz, der für das FED Board innerhalb der HERA-B Auslese vorgesehen ist, in Erinnerung, so lassen sich die folgenden Aufgaben direkt daraus ableiten:

- Adressat für die Fast Control, d.h. für den First Level Trigger, als "ReadOut Controller"; das FED nimmt die benötigten Informationen über die getriggerten Ereignisse entgegen.
- Daraus ableitend steuert der Front-End Driver, wie der Name sagt, die Auslese der Level-1 Pipeline, d.h. im Fall der SVD [2] und MSGC Subdetektoren die analogen Auslesechips.
- Analog-/Digitalwandlung der Pulshöheninformationen für diese beiden Detektorkomponenten.
- Zusammenstellen des für alle FED gleichen Datenausgabeformates, inklusive der zur Markierung einzelner Ereignisse benötigten "Header-

[1] Micro Strip Gaseous Chambers
[2] Silizium Vertex Detektor

und Trailerdaten"; Versendung dieses Datensatzes gemäß der Spezifikation des SHARC Link Protokolles über den SHARCLink zum Second Level Buffer.

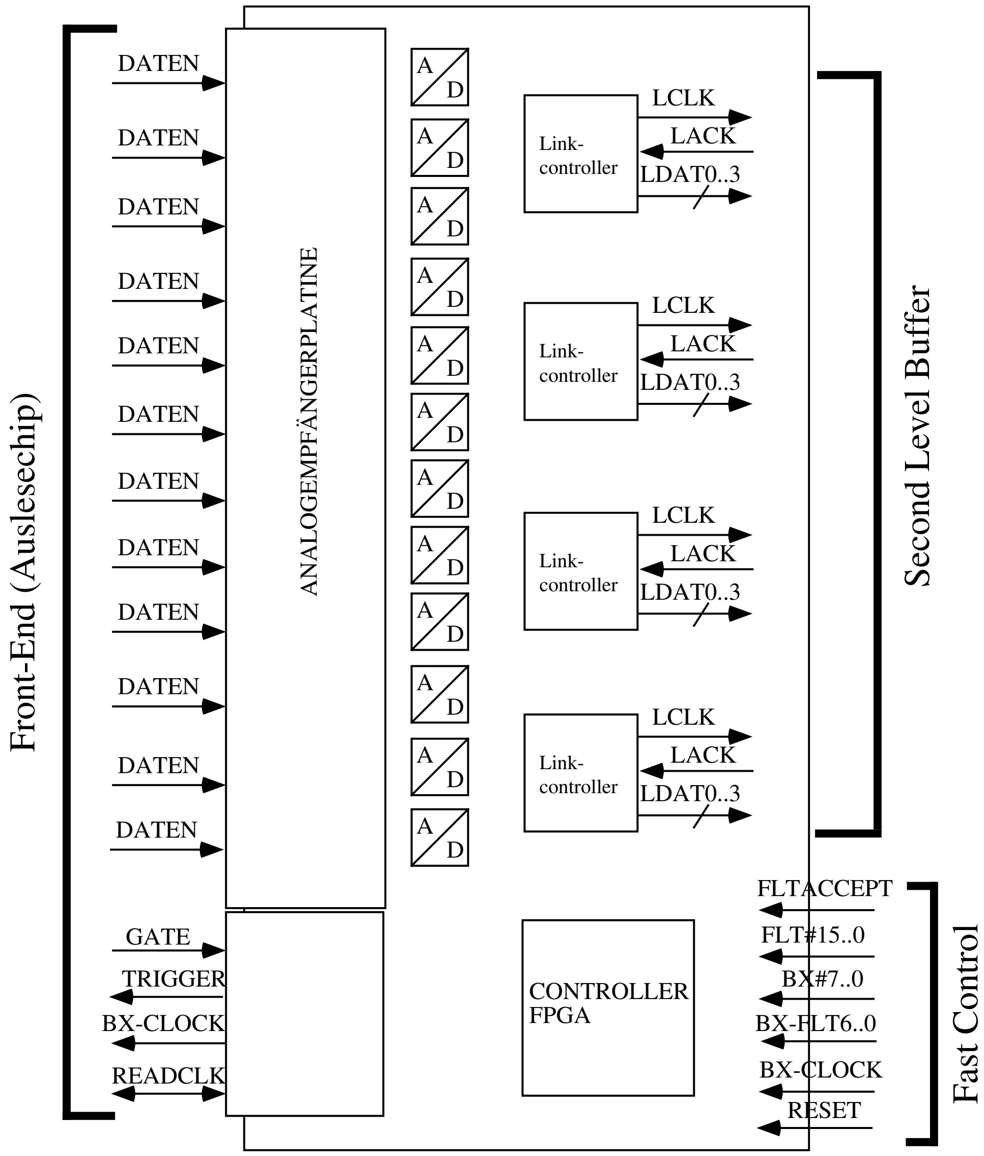

Abbildung 3.1: Ein Blockdiagramm der Front-End Driver Karte.

Bei der Planung des Prototypen hat man sich entschlossen, die Abarbeitung der einzelnen Teilaufgaben in verschiedene funktionelle Einheiten zu kapseln (s.Abb.3.1).

Es gibt eine Signalschnittstelle zur Fast Control, die Kommunikation zu den jeweiligen Auslesechips wird über ein entsprechendes Interface abgewickelt und zur Analog-/Digitalwandlung sind Flash ADC's vorgesehen. Desweiteren gibt es spezielle Controllerchips, die den Datenausgabetransfer zum Second Level Buffer durchführen. Die Steuerung der Funktion der einzelnen Komponenten auf dem Board übernimmt ein eigener Controller Baustein, der Hauptgegenstand dieser Arbeit ist. Die ganze Karte ist über ein VME-Bus Interface programmierbar.

3.1 Die Schnittstelle zum Front End

Der analoge Auslesechip

Für die totzeitlose Auslese bei HERA-B müssen die Detektordaten in der Level-1 Pipeline für den Zeitraum der Triggerentscheidung zwischengespeichert werden. Zur analogen Auslese von MSGC's und Silizium Vertex Detektor wird dieser Zwischenspeicher in die analogen Auslesechips integriert sein. Die Anforderungen an den analogen Auslesechip bei HERA-B lauten:

- 128 kanaliger Vorverstärker mit sehr kurzer Anstiegszeit ($\leq$ 50ns) und geringem Rauschen. Die Samplingfrequenz für Pulshöheninformationen muß 10,416 MHz betragen.
- ein 128 Stufen tiefer Ringpuffer für jeden der 128 Kanäle als Level-1 Zwischenspeicher.
- ein Auslesemultiplexer, der die 128 Kanäle eines Ereignisses seriell multiplext und versendet. Die Auslesefrequenz des Multiplexers sollte $f_{Read} \geq 20.8 MHz$ betragen.
- Totzeitlose Auslese wird durch gleichzeitiges Schreiben und Lesen des Zwischenspeichers möglich.
- ein Multieventbuffer zur Zwischenspeicherung schon getriggerter Ereignisse solange noch vorherige Ereignisse versendet werden.
- der Chip sollte "daisy-chaining"[3] erlauben.

Ein Vertreter eines solchen "analogen Pipelinechips" ist der Helix-128, ein Mitglied der sog. "Helix-Familie" von Vorverstärkerchips, die im Heidelberger ASIC-Labor entworfen wurden [FEUERSTACK]. Der Entwurf des Helix-128 ist diesen Anforderungen entsprechend spezifiziert. Zur Zeit ist dieser Chip allerdings noch nicht erhältlich (ab Herbst 1996). Der kleinere Bruder des Helix-128, der Helix-32, hingegen existiert schon mit der vollen Funktionstüchtigkeit. Im Gegensatz zu seinem großen Bruder hat dieser Chip nur 32 Kanäle und einen Zwischenspeicher von 64 Stufen Tiefe.

[3] "Daisy-Chaining" bezeichnet das sequentielle Auslese mehrerer Chips hintereinander, z.B. Chip1 Kanäle 0..127, danach Chip2 Kanäle 0..127.

Bx-Clock	Die HERA-Bunchcrossingclock (10,416 MHz), von der Fast Control verteilt, vom FED an das Front-End weiterverteilt.
Trigger	Liefert einen Trigger an den Auslesechip, damit dieser das nächste Datum an das FED versenden kann.
ReadOutClock	Die Samplingfrequenz für die FADC's; kann sowohl nach außen geliefert als auch von dort empfangen werden.
DATA-VALID	Das "DATA-VALID"-signal liefert der Auslesechip an das FED, um das Anliegen von Analogdaten zu signalisieren.

Tabelle 3.1: Diese Tabelle beschreibt die einzelne Steuersignale, die zur Kommunikation zwischen Front-End und Front-End-Driver benötigt werden.

Weitere Kandidaten für die Auslese sind der sog. FELIX-Chip und der APV-5 Auslesechip. Diese werden im Rahmen der RD-20 Kollaboration am CERN entwickelt und sollen auch die obengenannten Anforderungen erfüllen [FEUERSTACK].

Alle diese Bausteine sind noch nicht verfügbar. Für die Tests des HERA-B Detektors in diesem Jahr hat man sich entschieden, den APC-128 Auslesechip zu benutzen. Dieser Chip hat 128 Eingangskanäle aber nur einen analogen Zwischenspeicher von 32 Stufen. Für die endgültige Auslese ist er aber insbesondere dadurch ungeeignet, daß für jeden Auslesezyklus das Einlesen der Daten in den Zwischenspeicher angehalten werden muß . Überdies kann man den APC-128 nur mit einer maximalen Frequenz von $\approx$1 MHz auslesen [BAUER].

Die Frontplatte

Die Verbindung Auslesechip $\longleftrightarrow$ Front-End Driver läßt sich in die digitalen Steuersignale zur Kommunikation mit dem Front-End und die analogen Detektorinformationen aufteilen. Beim Entwurf hat man dieses berücksichtigt und die unterschiedlichen Aufgaben auf zwei verschiedene Komponenten verteilt. Diese sind auf zwei Aufsteckplatinen implementiert, so daß man für künftige Änderungen einfach diese Platinen ersetzt. In Tabelle 3.1 sind die Steuersignale, die die Kommunikation Front-End $\longleftrightarrow$ Front-End-Driver abwickeln, zusammen mit einer kurzen Funktionsbeschreibung aufgeführt.

Die "BX-Clock" ist der HERA-Systemtakt, mit dem die Detektorinformationen von der Detektorelektronik ausgelesen werden muß . Diese Taktfrequenz wird vom FED über einen Ausgang an das Front-End weitergeführt.

Das "Trigger"-signal wird vom FED generiert und zeigt dem Auslesechip an, daß das Ereignis, das als nächstes im Ringpuffer zum Überschreiben ansteht, ausgelesen wird.

Die "Readoutclock" kann sowohl an das FED geliefert werden, als auch vom FED erzeugt und an den Auslesechip weiterverteilt werden. Dieses Signal ist über Jumper entweder als Ein- oder Ausgang zu betreiben. Für den Betrieb des APC-128 bei den diesjährigen Tests des Silizium Vertex Detektors wird aufgrund der niedrigen Auslesefrequenz und des komplizierten Steuerprotokolls die Auslesefrequenz von außen zugeführt sein.

Liegen analoge Daten an den Eingängen des FED an, so signalisiert dieses der Auslesechip anhand des "DATA-VALID"-Signales dem FED.

Der Analogempfänger

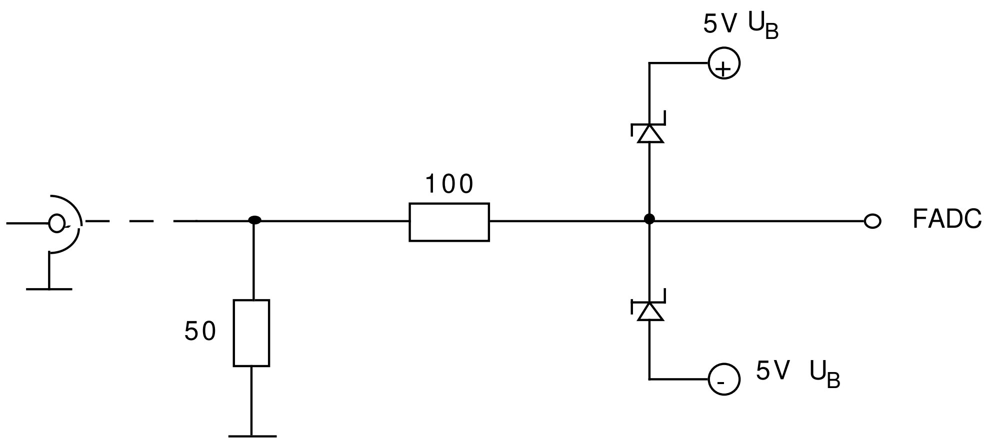

Abbildung 3.2: Dieses Bild zeigt die Beschaltung der Analogempfängerplatine. [RUSNYAK]

Für die Tests in diesem Jahr hat man sich zu einer elektrischen Übertragung der Detektordaten entschlossen. In der Abbildung 3.2 ist die Schaltung für einen Kanal dargestellt. Der Eingangspegel des Analogsignales soll zwischen 0..2,5V betragen.

Die Auslesechips sind direkt an die auszulesenden Kammern gebondet. Das FED Board wird im Kontrollraum untergebracht sein, damit man diese Komponente leichter warten und austauschen kann, ohne auf die Strahlzeiten Rücksicht nehmen zu müssen. Insgesamt sind die Daten über eine Strecke von 40 Metern zu übermitteln. Zur Verminderung einer Einstreuung über diese Strecke wird beim endgültigen Aufbau eine optoelektronische Übertragung gewählt werden.

Um die Anzahl der zur Übermittlung benötigten Kabel möglichst gering zu halten, wird man eine bestimmte Zahl von Kammerkanälen seriell, d.h. zeitlich hintereinander abfolgend, übertragen. Es ist geplant im endgültigen Aufbau 256 Kanäle (zwei 128 Kanal Chips hintereinandergeschaltet) über ein Kabel zu übertragen.

Der Analog-/Digitalwandler

Das FED übernimmt für die Analogauslese, neben Anderem, auch die Aufgabe eines FADC's. Unter Berücksichtigung des Platzbedarfs aller Komponenten hat man sich entschieden, 12 FADC's auf dem Board zu installieren. Ruft man sich die serielle Übertragung von 256 Kanälen eines Ereignisses in Erinnerung, so kann man mit diesem Aufbau 3072 Detektorkanäle (12 * 256) mit einem FED verarbeiten. Bei diesen Bausteinen wurden A/D Wandler von Signal Processing Technologies (SPT7855) mit den folgenden Eigenschaften ausgewählt:

- 10 Bit Auflösung
- 25 Megasamples pro Sekunde
- 0..5V Eingangspegel (typisch 4.0V)
- 135 mW Leistungsaufnahme
- geringe Eingangsimpedanz von 5pF

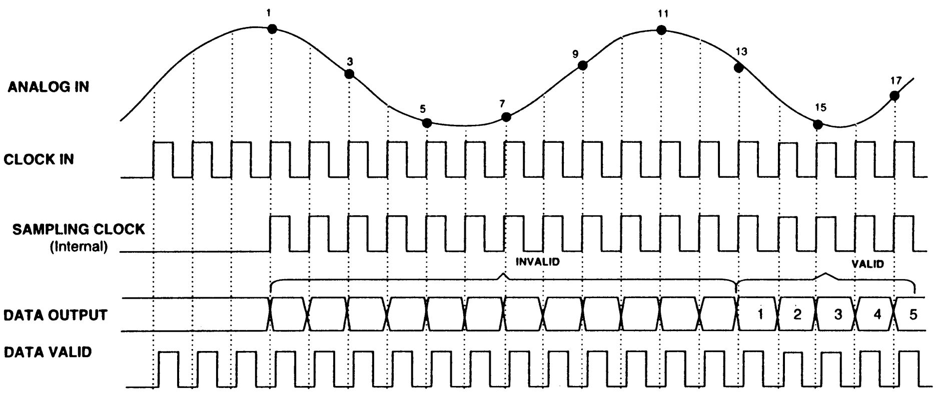

Abbildung 3.3: Die FADC's verzögern die Ausgabe der digitalen Daten um genau 12 Taktzyklen. [SIGNAL]

Ich möchte noch kurz eine Eigenschaft dieser Bausteine ansprechen, die direkte Konsequenzen auf den "ReadOutSequenzer" und das Timing zwischen den einzelnen Boardkomponenten hat. Wie aus Abbildung 3.3 zu erkennen ist, werden die Daten durch eine 12-stufige Pipeline verzögert, d.h. ein Eingangsdatum liegt erst 12 Taktzyklen später als digitaler Wert am Ausgang an, was impliziert, daß alle Steuersignale zur Versendung digitalisierter ADC-Daten dieses Pipelining emulieren müssen.

3.2 Schnittstelle zum Fast Control System

Die Fast-Control verteilt die von den Subdetektoren zur Datenauslese benötigten Signale. Dieses geschieht über einen speziellen Kabelbus, die Signale sind über die Reihen a,c des P2 Steckers der VME-Karte verfügbar.

Signal	Beschreibung
BX-CLOCK	HERA-Maschinentakt, der als differentielles Signal (TTL) verteilt wird
BX#	Die Bunchcrossingnummer ist eine 8-Bit Zahl, die das augenblickliche Bunchcrossing markiert.
FLT-ACCEPT	Auf die steigende Flanke dieses Signales sind die FLT# und BX-FLT# gültig.
FLT#	Diese 16-Bit Zahl gibt die globale First Level Trigger Nummer an.
BX-FLT#	Diese 7-Bit Zahl gibt die Bunchcrossingnummer des vom FLT gekennzeichneten Ereignisses an.
RESET	Das globale Reset Signal, das die gesamte Datenauslese von HERA-B in einen bestimmten Anfangszustand zurücksetzt.

Tabelle 3.2: In der obigen Tabelle sind die vom FED zur Datenauslese benötigten Signale aufgeführt. [FCSSPEC]

Die Bunchcrossingclock (BX-CLOCK), d.h. der HERA-Maschinentakt, ist die Referenzfrequenz für den gesamten Detektor, deren Verteilung möglichst sorgfältig sicherzustellen ist. Man wird dieses Signal differentiell über die Pins A31 und C31 betreiben, wobei ein maximaler Jitter von 0,5ns nicht überschritten werden darf, um eine genaue Synchronisation zu gewährleisten.

FLT-ACCEPT ist ein sog. Strobesignal, auf dessen positive Flanke die beiden für die Ereignismarkierung benötigten Daten BX-FLT# und FLT# in einem Register abgespeichert werden müssen.

Die aktuelle Bunchcrossingnummer (BX#) wird zur Synchronisation des augenblicklichen Zeitpunktes mit dem Zeitpunkt des vom FLT getriggerten Ereignisses benötigt. Auf diesen Punkt werde ich noch bei der Diskussion der Controllerschaltung zurückkommen.

Mit der FLT# wird eine fortlaufende Markierung eines vom First Level Trigger akzeptierten Ereignisses vorgenommen.

BX-FLT# gibt die Stelle innerhalb der 128 Stufen tiefen Pipeline an, in der das akzeptierte Ereignis zwischengespeichert ist.

Zur Synchronisation des gesamten Detektors wird die gesamte Datenauslese mit einem globalen RESET Signal in einen definierten Anfangszustand gebracht.

In den beiden folgenden Abbildungen sind Impulsdiagramme für die Fast Control von HERA-B verdeutlicht. Die BX# wechselt jeweils auf die negative Flanke der BX-Clock und soll vom FED auf die positive Taktflanke übernommen werden.

FLT# und BX-FLT# liegen für den Zeitraum zwischen zwei negativen Taktflanken an und sollen, im Falle eines Strobe auf die positive Flanke von FLT-ACCEPT, die zeitgleich zur positiven Flanke des Systemtaktes liegt, vom FED übernommen werden.

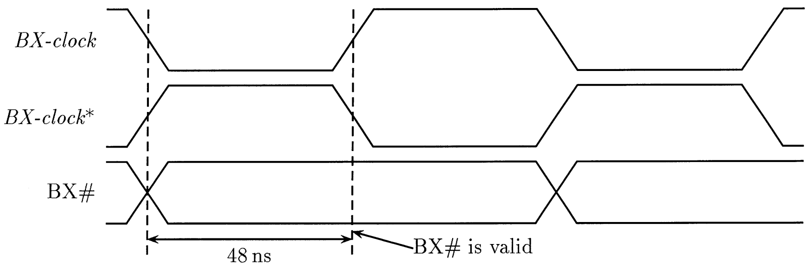

Abbildung 3.4: Das Timing Diagramm für die Bunchcrossingnummer. [FCSSPEC]

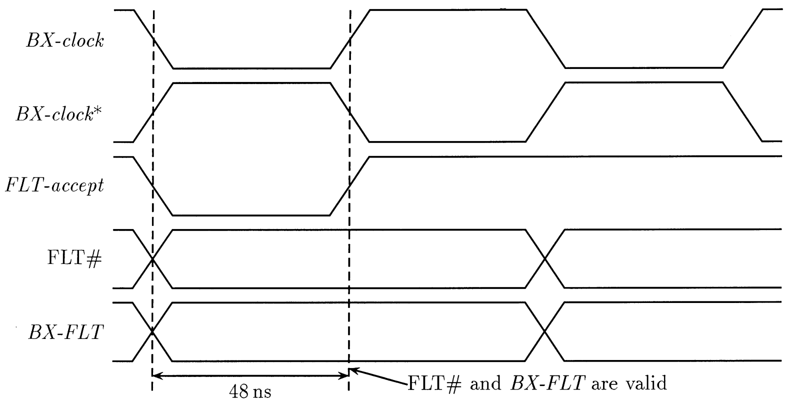

Abbildung 3.5: Das Timing von FLT# und BX-FLT# und FLT-ACCEPT. [FCSSPEC]

Diese Triggersignale können vom First Level Trigger in vertauschter Reihenfolge erzeugt werden. Von der Fast Control werden diese Trigger aber in die korrekte Reihenfolge gebracht. Allerdings können die Trigger noch direkt aufeinanderfolgen [FEUERSTACK]. Das FED hat dann die Aufgabe, diese Trigger zum korrekten Zeitpunkt an die Auslesechips weiterzugeben. Für die serielle Auslese eines Ereignisses aus der Pipeline wird ein Mindestzeitraum benötigt. Falls innerhalb dieses Zeitraumes ein weiterer Trigger verteilt wird, müssen die Informationen über dieses Ereignis zwischengespeichert werden. Hierfür ist im FED ein Multieventbuffer für die Ereignismarkierungen vorgesehen.

3.3 Schnittstelle zum Second Level Buffer

Für den Second Level Buffer hat sich als Implementation das sogenannte SHARC-Board [WURTH] durchgesetzt. Diese Karte beherbergt 6 Digitale Signalprozessoren **(DSP)** von Analog Devices **(ADSP-21060)**, den "SHARC-Prozessor", der sich durch eine 40 MIPS CPU, 512 kBytes SRAM on-chip, 6 parallele Linkports, 10 DMA Kanäle und 2 serielle Ports kennzeichnet. Durch die 6 Linkports, die jeweils bis zu 40 MByte/s Bandbreite besitzen, verfügt der Prozessor über ausgezeichnete I/O-Kapazitäten, und ist damit für ein größeres Array von Prozessoren, wie es bei HERA-B geplant ist, geradezu prädestiniert.

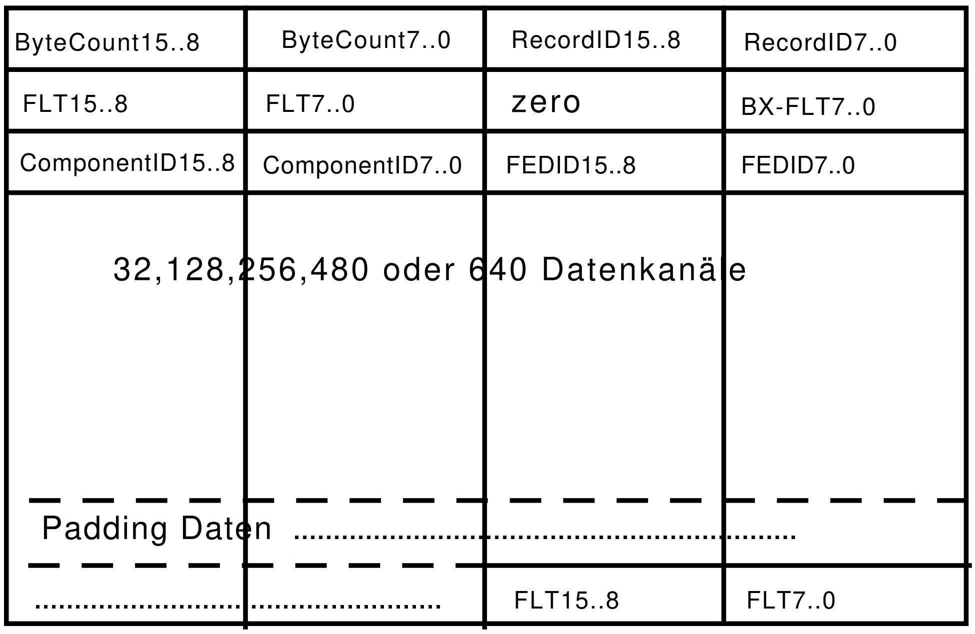

Abbildung 3.6: Das Ausgabe Format des Front-End Driver an den Second Level Buffer. [FORMAT]

An den Second Level Buffer werden die Daten von allen FED's in einem vereinheitlichten Datenprotokoll versendet (s.Abb. 3.6). Da die Datenauslese bei HERA-B asynchron abläuft, d.h. es wird nicht zwingend sichergestellt, daß die Trigger in einem festen Abstand zu einem festgelegten Referenzzeitpunkt erfolgen, muß die Datenidentifizierung anderweitig sichergestellt werden. Dieses geschieht durch eine entsprechende Markierung mittels Header- und Trailerdaten. Diese lauten wie folgt:

- **ByteCount** ByteCount gibt die Länge des Ausgabedatenformates in Byte an.
- **RecordID** Diese Zahl identifiziert den entsprechenden Daten Record.
- **ComponentID** ComponentID identifiziert die Software Detektor Komponente; einige größere, physische Komponenten mögen in mehr als eine Software Komponente aufgeteilt sein.
- **FEDID** Dieses ist die Identifizierung der jeweiligen FED Einheit; sie wird für die Adressierung von Detektorkanälen relativ zu ihr selbst benötigt.
- **BX-FLT** Diese 7-Bit Zahl identifiziert den Zeitpunkt des Bunchcrossings innerhalb eines Bereiches von 0..127, für den ein FLT-Accept generiert wurde.
- **FLT#** Die Ereignisnummer für jedes FLT-Accept; diese Zahl läuft von 0..65535, so daß ein Tagging bei einer FLT-Rate von 50kHz möglich ist.

ByteCount, RecordID, ComponentID und FEDID werden während der Initialisierung des FED-Controllers über einen VME-Bus Zugriff in Register geladen.

Nach Versendung der Headerinformationen folgen in dem vereinbarten Datenprotokoll die digitalisierten Pulshöheninformationen der Detektorkanäle. Je nach Auslesechip variiert die Anzahl der zu übertragenden Datenkanäle. Beim HELIX-32 sind es z.B. nur 32 Kanäle, in einem Aufbau mit zwei hintereinandergehängten HELIX-128 müssen 256 Detektorkanäle mit einem Datensatz versendet werden.

Sind alle Detektorinformationen übersandt, muß , aufgrund der Speicherstruktur des SHARC Prozessors, zu dem die Daten geschickt werden, der Datensatz soweit mit "Dummydaten" aufgefüllt werden ("Padding"), bis der Anzahl der insgesamt mit einem kompletten Datensatz verschickten Byte genau durch 4 und 6 teilbar ist.

Im letzten Schritt wird dann noch zum Abschluß die FLT# als Trailer versandt.

Die Übertragung der Daten an den Second Level Buffer wird über speziellen I/O-Ports des SHARC-Prozessors, die Link-Ports, durchgeführt. Betrachten wir zuerst die Funktionsweise dieser Ports, bevor wir auf die Abwicklung des speziellen SHARC-Link Protokolls von Seiten des FED eingehen.

Der Abbildung 3.7 entnimmt man, daß die 6 Links aus Buffern und Ports aufgebaut sind und bidirektional, d.h. sowohl als Eingang als auch

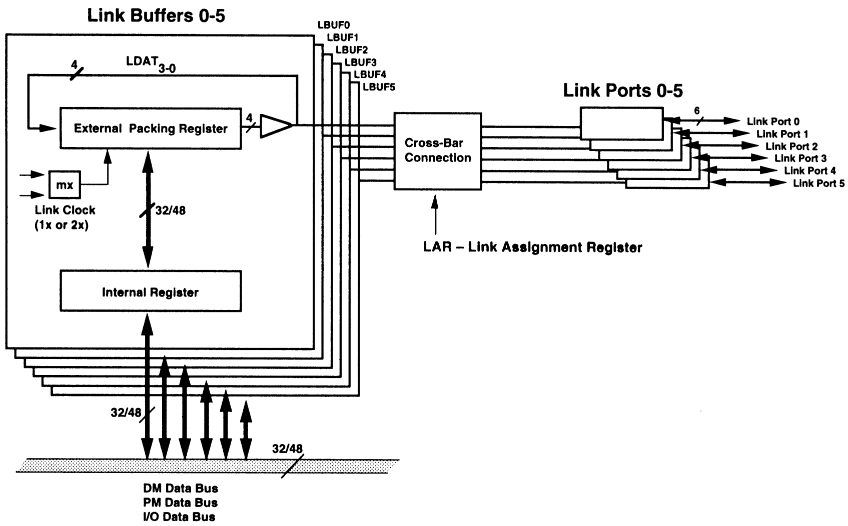

Abbildung 3.7: Aufbau des Link Buffers im ADSP-2106x. [SHARCDATA]

als Ausgang betrieben werden können. Die Zuordnung von Linkports zu Linkbuffern ist frei programmierbar.

Die Linkbuffer besitzen zwei Register von 32- bzw. 48-Bit Breite, wobei das interne Register über einen DMA-Transfer geleert werden kann. Im Folgenden interessiert uns aber nur der Empfang von Daten durch den SHARC-Prozessor.

Ist das externe der beiden Register vollgeschrieben, so wird dessen Inhalt in das intern zugängliche Register umgeladen und das nächste Wort kann empfangen werden. Das interne Register wird dann per DMA-Transfer entleert.

Sobald dieses geschehen ist, ist sichergestellt, daß das gerade empfangene Wort in das interne Register umgeladen werden kann und unverzüglich ein weiteres Wort akzeptiert wird - somit kann der SHARC per "Handshake" signalisieren: das Wort ist korrekt angekommen, der Prozessor ist bereit zum Empfang des nächsten Wortes.

Die Daten werden in 4-Bit Stücken (sog. "Nibbles") parallel übertragen. Das Linkacknowledge-Signal (LACK) geht frühestens nach 2 Nibbles auf low und muß spätestens bei Anliegen des letzten Nibbles (32-Bit Modus: 8.Nibble - 48-Bit Modus: 12.Nibble) wieder auf high gezogen sein (s.Abb.3.8). Ist dieses nicht geschehen, so hat der Sender nach Absetzen dieser Nibbles die Linkclock auf "HIGH" zu belassen[4] - der SHARC liest die Daten auf die fallende Flanke von Linkclock in sein externes Register. Erst bei der

[4]Der Sender muß das LACK-Signal überprüfen.

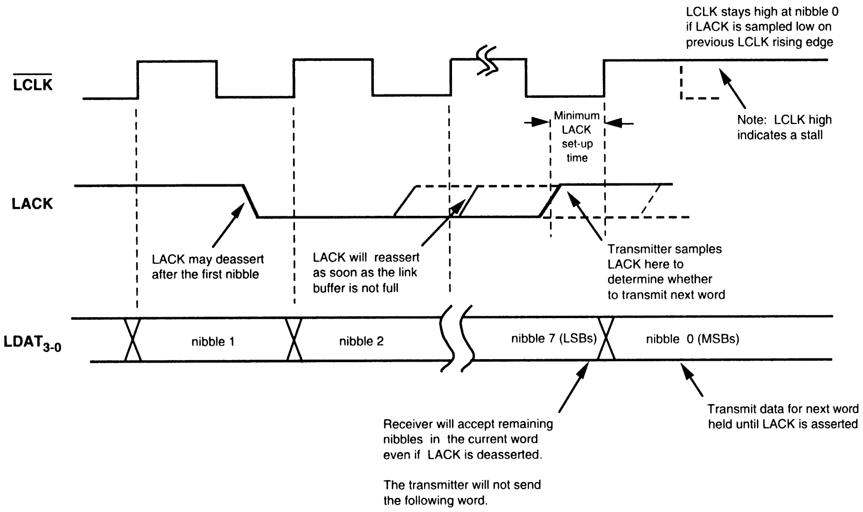

Abbildung 3.8: Das Timing für den SHARC Link. [SHARCDATA]

nächsten fallenden Flanke hat der Sender das nächste Nibble angelegt, so daß der SHARC das erste Datum übernehmen kann. Der Link kann mit einer variablen Frequenz von bis zu 80MHz betrieben werden.

Für die Abwicklung dieses Protokolls von Seiten des FED ist ein spezieller Baustein, der sog. Linkcontroller, vorgesehen. Wir haben für die Implementierung den ispL1032-80 Baustein der Firma Lattice, einen programmierbaren Chip benutzt [RAUSCH]. Dieser Linkcontroller hat zwei Augaben parallel zu erfüllen:

1. Die Entgegennahme von neuen Daten, sowohl von den FADC's als auch vom Controller und

2. die korrekte Übermittlung dieser Daten zum Second Level Buffer.

Auf der Eingangsseite liegen 8-Bit Daten sowohl für die Header-/Trailerinformationen als auch beim Auffüllen des Datensatzes oder der Übermittlung von Fehlercode vor. Andererseits haben wir aber auch 10-Bit Daten bei der Übermittlung von FADC- bzw. Testdaten anliegen. Diese Daten werden zwischengespeichert, um dann in 4-Bit Worten vom Linkcontroller versandt zu werden.

Hierzu haben wir uns jetzt der folgenden Vorgehensweise bedient: der Linkcontroller durchläuft immer Zyklen von drei Zuständen Hi-,Mid und Lownibble (s.Abbildung 3.9). Damit jeder Zyklus innerhalb eines Takte der Auslesefrequenz abgeschloßen ist, taktet man den Chip mit der 3-fachen Auslesefrequenz. Bei den 10-Bit Daten hängt man 2 Nullen unten an das

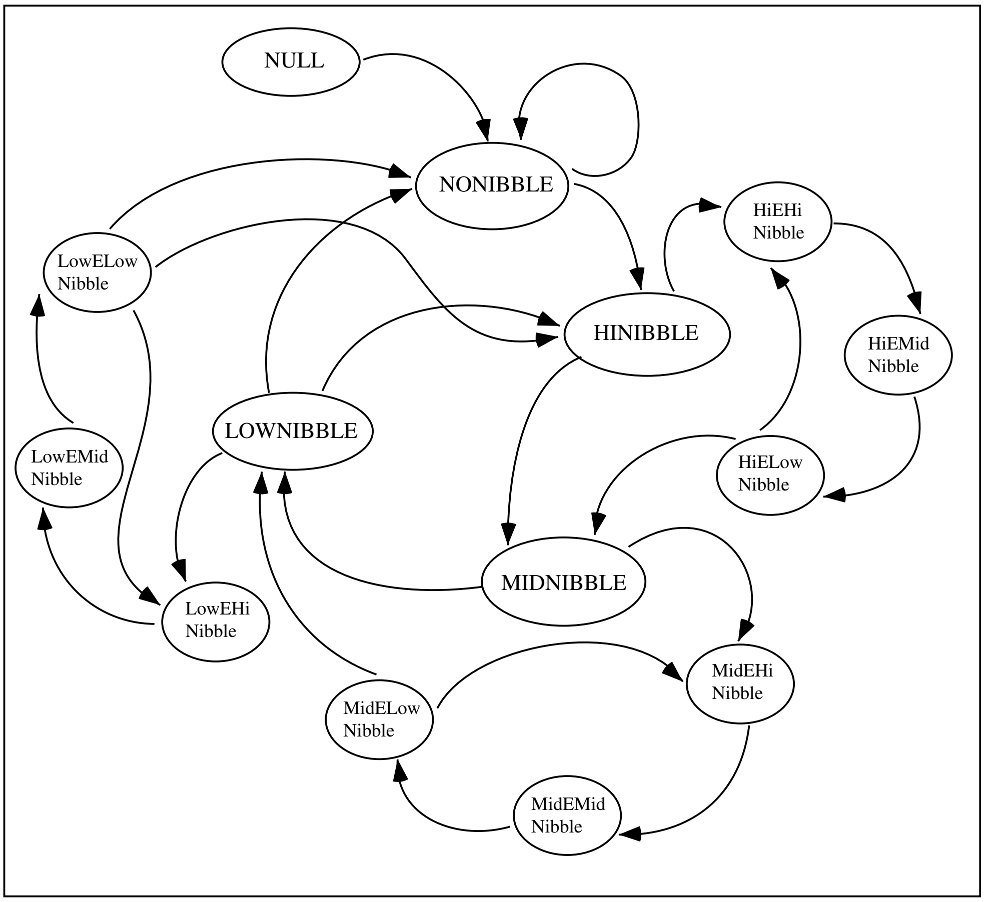

Abbildung 3.9: Das Bubblediagramm des Linkcontroller Bausteines. [RAUSCH]

jeweilige 10-Bit Wort an[5] und übermittelt 12-Bit Wörter. Bei der Versendung der 8-Bit Daten hingegen läßt man bei Durchlaufen des 3.Zustandes innerhalb eines Zyklus' die LinkClock auf "High", so daß der SHARC keine Daten einliest. Diesen 3-taktigen Zyklus muß man immer aufrecht erhalten, damit eine feste Phasenbeziehung zum Takt des Xilinx Controllers nicht aufgegeben wird.

Was passiert, wenn der SHARC-Prozessor die hereinkommenden Daten nicht so schnell aus seinen Linkbuffern entleeren kann?

In diesem Fall legt die Zustandsmaschine 3-taktige Warteschleifen ein (s.Abb. 3.9), bis der SHARC sein "WEITER"-Signal anzeigt, d.h. das LACK auf "High" zieht. Der Linkcontroller signalisiert außerdem dem Controller Chip das Aussetzen des SHARC-Prozessors über sein "WACK"-Signal [6].

[5] z.B.:0010111011 — 00 ergibt ein 12-Bit Wort.

[6] **W**ord **Ack**nowledge

Durch diese Wartezyklen bleibt sichergestellt, daß die starre Phasenbeziehung zwischen Linkcontroller und Controller ständig besteht.

Augenblicklich wird der SHARC-Link von uns mit einem 32-Bit Buffer betrieben, so daß der Linkcontroller nach jedem 8.Nibble das LACK-Signal überprüft.

3.4 Die Bereitstellung der verschiedenen Systemtakte

Für den Betrieb der einzelnen Komponenten des FED sind insgesamt drei verschiedene Systemtakte in einem festen Verhältnis zueinander notwendig:

1. Der HERA-Systemtakt von 10,416 MHz, die sog. Bunchcrossingclock (BX-CLOCK), die von der Fast Control an alle Front-End's und FED's geliefert wird.
2. Die sog. Auslesefrequenz (ReadOutClock) taktet die Übertragung der analogen Daten zum Front-End Driver, die A/D-Wandlung und den Teil des Controllers, der das Versenden des Ausgabedatenformates steuert. Für den Fall, daß 256 Kanäle in einem Datensatz seriell übertragen werden, ist vorgesehen, daß diese Auslesefrequenz genau zweimal der HERA-Systemclock entspricht [7].
3. Die im folgenden LinkClock genannte Taktfrequenz treibt den SHARC-Link, d.h. die Verbindung FED $\rightarrow$ Second Level Buffer.Dieser Takt beträgt genau das 3-fache der Auslesefrequenz.

Die HERA-Clock wird aus Gründen der detektorinternen Synchronisation immer von außen zugeführt sein [8]. Für die anderen Systemtakte bieten sich, wie in Abbildung 3.10 skizziert, verschiedene Szenarien an:

1. Die BX-Clock wird über den P2-Stecker zugeführt, die Auslesefrequenz hingegen über die Frontplatte. Die LinkClock wird dann vom FED phasenstabil zur Auslesefrequenz generiert.
2. Einzig die BX-Clock wird von außen an das FED zugeführt. Die Karte erzeugt mittels der Frequenzvervielfacher die beiden anderen Systemtakte.
3. Die BX-Clock wird von einem eingebauten Quarz erzeugt, daraus werden dann die anderen, benötigten Systemtakte auf dem FED generiert.

[7] Allerdings will man sich für die Zukunft die Option offenhalten, die Front-End Pipeline mit einem höheren Takt auszulesen (z.B. 26 MHz), damit der Zeitraum für die Übertragung der Daten vom Detektor zum FED verkürzt wird

[8] i.e.. von der Fast Control

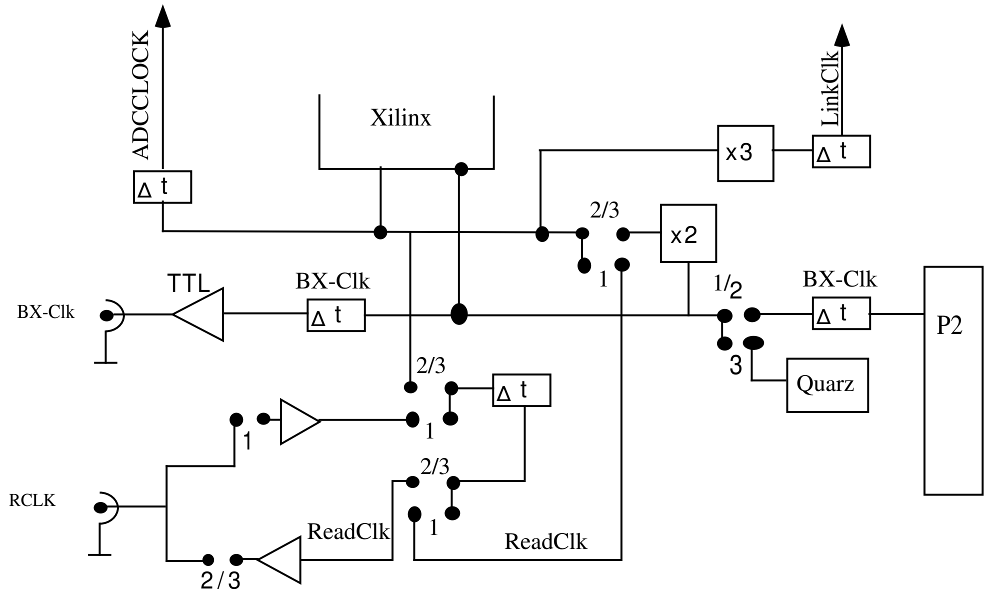

Abbildung 3.10: Es gibt drei verschieden Möglichkeiten die Systemtakte auf dem FED zu verteilen bzw. zu generieren. [RAUSCH]

Die verschiedenen Szenarien können über Jumper eingestellt werden (s.Abb. 3.11). Die Synchronisation der von den Vervielfachern erzeugten Frequenzen verwendet eine PLL[9]-Schaltung. Diese Schaltung ist im Anhang dokumentiert. Sie ist auf einer Aufsteckplatine plaziert, so daß man in einer veränderten Ausleseumgebung nur diese Platine auswechseln und die entsprechenden Jumper umstecken muß.

Auf dem Board sind drei programmierbare [10] und drei mit Jumpern steckbare Delay-Lines vorgesehen.

Die drei programmierbaren Delay-Lines dienen der Synchronisation von Signalen zwischen Front-End und Front-End Driver. Diese sind:

- BX-CLOCK.
- Trigger
- Readoutclock

Bei den Bausteinen, die Verzögerungszeiten von 10ns bis zu 137,5ns in 256 Schritten zu jeweils 0,5ns zulassen, handelt es sich um 8-Bit Delays vom Typ DS 1020-50 der Firma Dallas Semiconductors [RAUSCH]. Diese Verzögerungszeiten können individuell für jeden einzelnen Baustein über einen VME-Bus Zugriff eingestellt werden.

[9]Phase-Locked-Loop

[10]Es ist noch ein weiterer, nicht beschalteter Baustein bestückt, um notfalls ein zusätzliches Signal feinabzustimmen.

Jumper für die Clockverteilung

Modus	204	205	206	207	208	209	210	211	212	213
3	x	o	x	o	x	x	o	x	o	o
2	x	o	x	o	x	o	o	x	o	x
1	o	x	o	x	o	o	x	o	x	x

x = Jumper stecken
o = Jumper offenlassen

Abbildung 3.11: Die Tabelle gibt die Jumperbelegung für die Clockszenarien wieder. [RAUSCH]

Über die steckbaren Delay-Lines werden die auf der Karte verteilten Systemtakte synchronisiert.

- Die Samplingfrequenz der FADC's wird mittels der Jumper 214 und 215 über eine Inverterkette verzögert.
- Mit den Jumpern 216 und 217 läßt sich der Systemtakt für die Linkcontroller, die LinkClock, feineinstellen.
- Jumper 218 und 219 stellen die Verzögerung der BX-Clock auf dem FED-Board ein.

Kapitel 4

Diskussion des zentralen Controllers

Der Auslesesequenzer ist die zentrale Steuereinheit für sämtliche Funktionen der Front-End Driver Karte. Nach Auflistung der verschiedenen Aufgaben, die der Controller zu erfüllen hat, beschäftigt sich der folgende Abschnitt mit der funktionalen Beschreibung des Chips. Für den Entwurf dieses Controllers wurde am Physikalischen Institut ein neuer Ansatz gewählt: Beschreibung der Funktionalität des Bausteines mit einer Hardwarebeschreibungssprache (VERILOG HDL) und die anschließende Implementierung mit dem CADENCE-Entwicklungssystem. Abschnitt 4.3 befaßt sich mit dieser Thematik.

4.1 Aufgaben des Controllers

Im laufenden Betrieb sehen die regelmäßig wiederkehrenden Aufgaben für die Karte wie folgt aus:

1. Entgegennahme eines FLT-akzeptierten Ereignisses von der Fast Control und Abspeichern der benötigten Daten in einem Zwischenspeicher.

2. Triggern des Auslesechips zum korrekten Zeitpunkt, damit dieser das passende Ereignis aus seinem Speicher ausliest und an das FED versendet.

3. Anschließend versendet das FED die Headerinformationen für dieses Ereignis über den SHARC-Link und wartet auf das DATAVALID ("Start") Signal des Front-End.

4. Bei Anliegen der Analogdaten beginnt die Karte diese zu digitalisieren und verschickt, nebst abschließenden Trailerdaten, den restlichen Teil des korrekten Datensatzes an den Second Level Buffer.

5. Gleichzeitig kann das FED weiterhin getriggerte Ereignisse entgegennehmen und den Auslesechip anstoßen.
6. Nach Abschluß des Versendevorganges wartet die Karte auf das nächste Ereignis.

4.2 Funktionale Beschreibung

Die Abwicklung der verschiedenen, oben beschriebenen Aufgaben wurde im Entwurf in zwei Funktionsblöcke gekapselt.

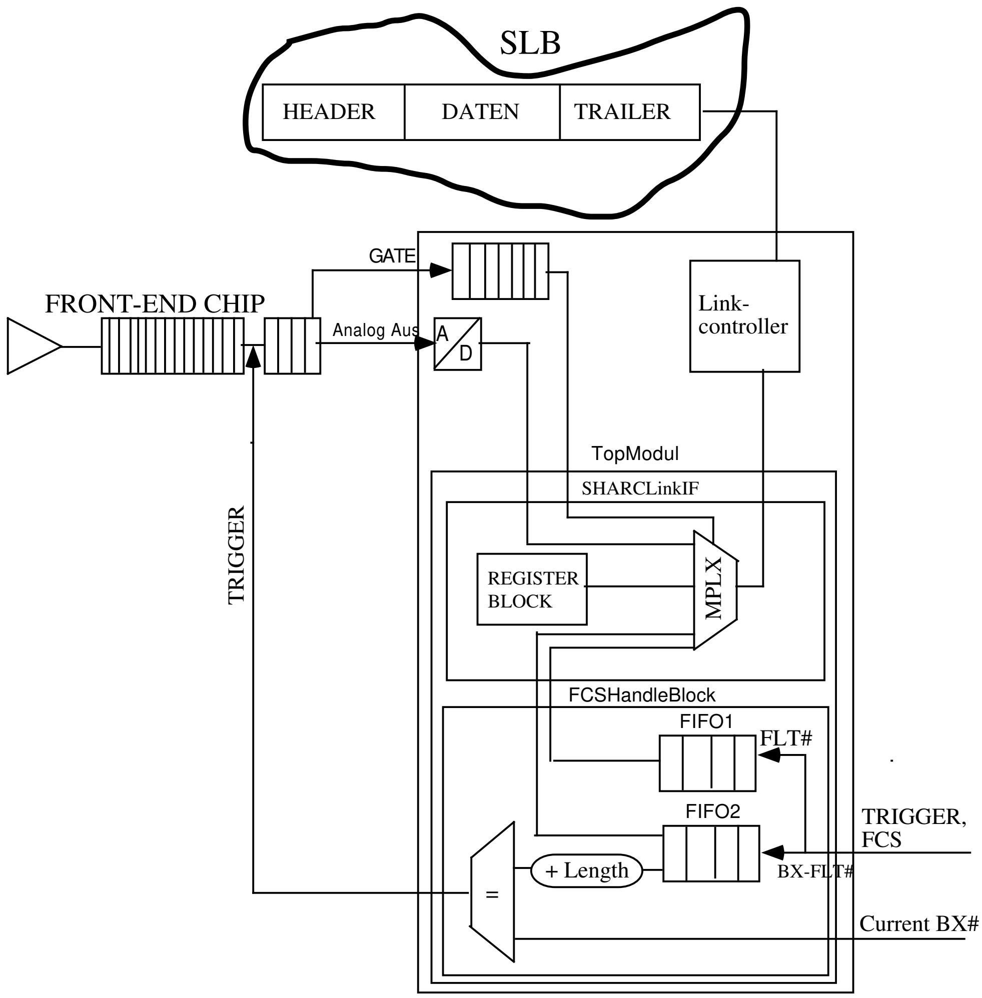

Abbildung 4.1: Ein prizipielles Strukturdiagramm des Controllerchips.

In der Abbildung 4.1, die ein grobes Strukturdiagramm des Controllers wiedergibt, sind die beiden Funktionsblöcke und ihre Wechselwirkung dargestellt. Man erkennt dort auch die im vorigen Abschnitt beschriebenen

Aufgaben wieder. Der FCS Handle Block handhabt, wie der Name sagt, die Signale, die von der Fast Control angeliefert werden, triggert den Auslesechip und stößt den SHARC-Link Interface Block (SHARCLinkIF) an. Dieser kontrolliert die korrekte Digitalisierung der Detektorinformationen und steuert die Übermittlung der Daten an den Linkcontrollerbaustein.

4.2.1 Das Topmodul

Wie man aus der obigen Abbildung 4.1 erkennt, bildet das Topmodul die äußere Schnittstelle des Controllers, in dessen Inneren sich die beiden Funktionsblöcke befinden. Desweiteren erwähnte ich bereits, daß bei der Initialisierung dieses Chips notwendige Informationen für den Betrieb der Karte geladen werden müssen. Diese Register sind zweckmäßigerweise im TopModul untergebracht, da beide Submodule selbständig darauf zugreifen. Dieser Registerblock läßt sich in verschiedene Bereiche aufteilen, entsprechend den jeweiligen Informationen, die sie beherbergen. Das genaue Registerbild mit den Zugriffsadressen und der genauen Beschreibung des jeweiligen Eintrages ist im Anhang wiedergegeben. Dort ist auch ein Pascal-Beispielprogramm für die Initialisierung des Controllers aufgeführt.

Diskussion des Registerblocks

Der erste Block umfaßt die zur Markierung des Datensatzes notwendigen Informationen (z.B. ByteCount etc.), die ich weiter oben beschrieben habe. Da der Datenbus nur 8-Bit breit ist, müssen diese 16-Bit Werte mittels zweier Zugriffe in das jeweilige Register geschrieben werden.

Der nächste Block von insgesamt vier Adressen dient zur Adressierung der vier Delay-Lines. Wie schon in Kapitel 3 erwähnt, sind diese Bausteine parallel mit einem 8-Bit Wert programmierbar. Dieses geschieht über den gleichen Typ von VME-Bus Zugriff wie bei der Initialisierung des Controllers. Dem Controller kommt hierbei die Aufgabe zu, den Write Strobe an die Delay-Lines zu liefern, so daß diese den anliegenden 8-Bit Wert in ihren internen Speicher schreiben können.

Der nächste Block von insgesamt 6 Registereinträgen beherbergt Informationen, die für die Ablaufsteuerung des Controllers notwendig sind. Wie es oben in der Abbildung dargestellt ist, benötigt der Sequenzer für das korrekte Triggern des Auslesechips Informationen über die Tiefe des Level-1 Buffers; wartet er zu lange, gehen Daten verloren [1] - andersherum kann der Trigger zum falschen Zeitpunkt, d.h. zu früh abgesetzt werden. Desweiteren benötigt der Controller die Anzahl der Datenkanäle, die seriell übertragen und in einem Datensatz versendet werden. Erinnern wir uns an die Auflistung der verschiedenen Auslesechips in Kapitel 3, so hat der HELIX 32 nur

[1] z.B. bei Verwendung eines HELIX 32 mit der Pipelinetiefe von 64 Ereignissen in der passenden Konfiguration für den APV-6 mit 128 Stufen Tiefe.

32 Datenkanäle, der Datensatz wird entsprechend kürzer[2].

Desweiteren hat man sich überlegt, daß die Karte zu Testzwecken, aber auch für die Inbetriebnahme, in verschiedenen Auslesemodi betrieben werden kann. Aus diesem Grunde hat man ein weiteres Register vorgesehen, in welches der jeweilige Betriebsmodus einprogrammiert wird. Im Folgenden sind die Bezeichnungen der Modi, ihr Wert und eine Kurzbeschreibung des jeweiligen Betriebsmusters aufgelistet.

- **ROMODE_EXTSTROBE = 0:** Dieses ist der Modus für den Betrieb bei HERA-B. Die Auslese löst ein externer Strobe am "DATA-VALID"-Eingang, dessen Länge mindestens einem Takt der Auslesefrequenz entspricht, aus, worauf der Controller genau die Anzahl der einprogrammierten Datenkanäle digitalisiert und versendet.
- **ROMODE_TEST = 1:** Dergleiche Modus wie **ROMODE_EXTSTROBE**, allerdings werden statt der FADC-Daten Testdaten an den SHARC versendet.
- **ROMODE_ADCTRIGGER = 2:** Hier wird ein Trigger über einen VME-Bus Zugriff ausgelöst, im Gegensatz zu einem FCS angestoßenen Trigger in den beiden vorherigen Modi, der Chip getriggert und die FADC-Daten versendet[3]
- **ROMODE_TESTTRIGGER = 3:** Wie in **ROMODE_ADCTRIGGER**, allerdings werden hier Testdaten vom Controller an den Second Level Buffer versandt.

Die beiden letzten Betriebsmodi wurden intensiv bei der Inbetriebnahme und der Evaluierung der Funktionsfähigkeit der Karte genutzt. Dieser Block wird von einem 10-Bit Register abgeschlossen, das mit einem sog. "Testmuster" geladen werden kann, welches dann mit der Auslesefrequenz in einem Schieberegister rotiert, so daß man nachher bei der Auslese leicht die korrekte Übertragung anhand des periodischen Musters überprüfen kann.

Der letzte, große Block beinhaltet die Register, die für die Auslese in den Betriebsmodi "ROMODE_ADCTRIGGER" und "ROMODE_TESTTRIGGER" unentbehrlich sind. Da der Betrieb mit diesen Modi unabhängig von der Fast Control erfolgt, müssen die FLT# und BX-FLT# extern bereitgestellt werden, d.h. programmierbar sein. Desweiteren muß

[2]Insgesamt sind die in Abbildung 3.6 aufgelisteten Kanalzahlen vorgesehen:

- 32 Kanäle für den HELIX 32
- 128 Kanäle für den HELIX 128, APV-6/Felix und APC 128
- 256, 480 oder 640 Kanäle für das sog. "daisy-chaining von 2,4 oder5" "großen" Ausleseschips

[3]Später sollen zum Beispiel Testdaten in den Auslesechip eingespeist werden können, um die Verbindung Auslesechip $\longleftrightarrow$ FED zu testen.

man, nach Validierung der Daten, den Ausleseablauf mittels eines "Strobe" anstoßen.

Es stehen noch zwei einzelne Register zur Verfügung:

1. **SoftwareReset:** Mit diesem Register kann der Controller und damit die gesamte Karte unabhängig von irgendwelchen Detektorsteuersignalen, z.B zu Testzwecken in den definierten Anfangszustand zurückgesetzt werden, was die Auslese in einem Testaufbau erleichtert.

2. **ERROR:** Um ein einfaches On-Line Protokoll der auftretenden Fehlerzustände führen zu können, ist noch ein 3-Bit Register, in dem jedes Bit einen bestimmten Fehler signalisiert, vorgesehen.

Alle diese Register sind schreib- und lesbar, d.h. man kann jederzeit per Software die Korrektheit der Einträge überprüfen. Als letzte Aufgabe koordiniert das TopModul die verschiedenen "Reset"-Signale, die das FED mithört.

4.2.2 Der FCS Handle Block

Wie in der Abbildung 4.1 zu sehen ist, nimmmt der FCS-Handle-Block Trigger von der Fast Control (FCS) entgegen und speichert die FLT# und BX-FLT# zwischen. Insgesamt haben wir in diesem Block drei verschiedene Zustandsautomaten, die jeweils autark, unter Rückgriff auf die im Zwischenspeicher enthaltenen Informationen, ihre Aufgaben ausführen. Hierfür ist ein vier Stufen tiefer FIFO vorgesehen[4], dessen Tiefe genau der des Multieventbuffers des Auslesechips entspricht.

Man benötigt solch einen Zwischenspeicher, da, wie oben erwähnt, die Datenaufnahme asynchron verläuft. Ein weiterer First Level Trigger könnte also schon im nächsten Bunchcrossing generiert werden, während noch das vorherige Ereignis eingelesen wird. Dieses Ereignis will man nicht verlieren, also muß die DAQ die Informationen auch zwischenspeichern. Ein Modell für diesen Zwischenspeicher ist in Abbildung 4.2 wiedergegeben. Was passiert nun, nachdem ein FLT-Accept generiert wurde und FLT# und BX-FLT# im Speicher abgelegt sind?

Im FCS-FIFO werden die beiden Bits EVTRIGGERPENDING und EVDATAPENDING gesetzt, d.h. dieses Ereignis ist eingetragen.

Im Level-1 Buffer werden die Detektorinformationen genau soviele Takte lang zwischengespeichert (z.B. 128), wie dieser Zwischenspeicher tief ist. Genau zu dem Zeitpunkt, bevor die Ereignisse aus der Pipeline herausfallen, will man sie auslesen. Der Controller weiß jetzt die Bunchcrossingnummer, zu der die interessierenden Daten in die Pipeline hineingeschrieben wurden,

[4]Aus Detektorsimulationen ableitend rechnet man damit, daß mit vier Stufen FIFO-Tiefe bei einer FLT-Rate von 50kHz die Schwankungen des Triggerzeitpunktes ohne Datenverlust ausgeglichen werden können.

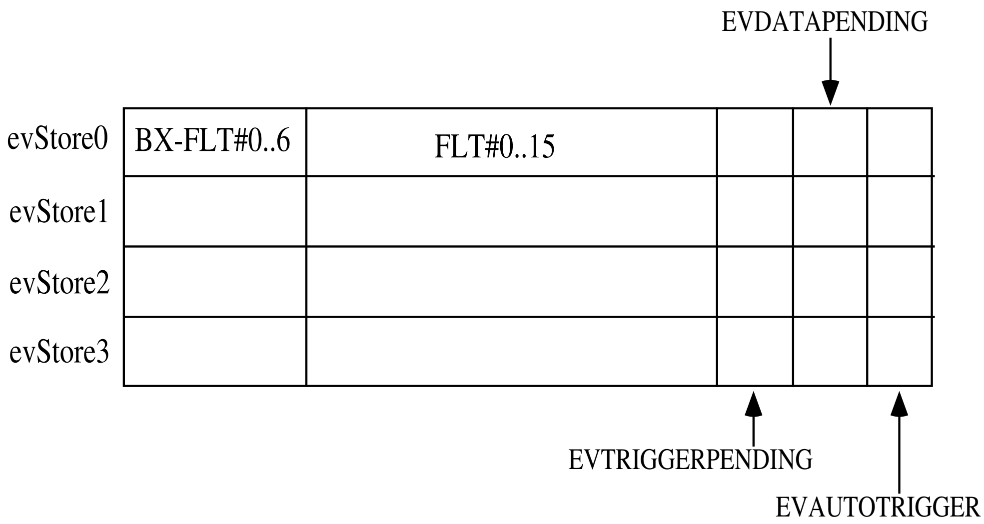

Abbildung 4.2: Das Modell des FCS FIFO's, der im Controller implementiert ist.

die BX-FLT-Nummer. Jetzt muß der Controller nur noch feststellen, wann diese Informationen aus der Pipeline fallen: genau zu dem Zeitpunkt, wenn **BX-FLT# + Pipelinetiefe = BX#** (s.Abb. 4.3), die ständig aktualisiert wird. Wenn dieser Vergleich stimmt, triggert er den Auslesechip, der diese Detektordaten dann seriell aus seinem Speicher ausliest und verschickt. Im FCS-FIFO wird EVTRIGGERPENDING gelöscht, da für dieses Ereignis der Trigger versendet ist.

Wenn der Controller die Trigger abgesandt hat, darf er den Zwischenspeicher noch nicht freigeben, der korrespondierende Multieventbuffereintrag mag ja noch nicht vollständig ausgelesen worden sein. Dieser Zustand wird durch das weiterhin gesetze EVDATAPENDING Bit angezeigt. Ist der Datensatz versendet, wird es gelöscht werden.

Der Autotrigger Modus

Sollte man einen der beiden "AUTOTRIGGER"-Auslesemodi benutzen, die über einen VME-Bus Zugriff ausgelöst werden, so muß man diesen Eintrag entsprechend markieren, da in diesem Fall der Vergleich der augenblicklichen Bunchcrossingnummer unterbleibt und der Datensatz unverzüglich abgeschickt werden kann. Hierzu wird EVAUTOTRIGGER Bit gesetzt.

Fehlerbehandlung

Im Betrieb des FED kann im FCS Handle Block der Fehler auftreten, daß mehr als vier Trigger vom First Level Trigger ausgelöst werden, ohne daß eine Speicherstelle im FCS-FIFO wieder freigegeben ist. Dieser Fehler muß erkannt und behandelt werden, da mindestens ein akzeptierter Trigger

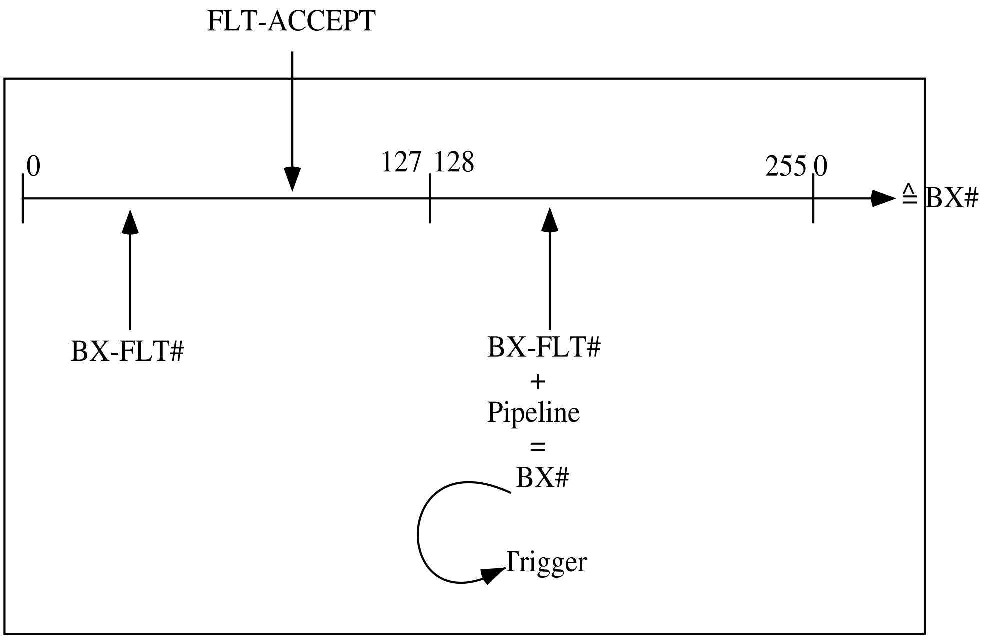

Abbildung 4.3: Die Abbildung verdeutlicht die Triggerlatency und den korrekten Zeitpunkt der Auslese. [FEUERSTACK]

verloren geht. Hierzu wird das unterste Bit im Fehlerregister des Controller gesetzt. Im Second Level Buffer kann man diesen Zustand aus dem jeweiligen Datensatz erkennen.

4.2.3 Der SHARCLink Interface Block

Dieser Funktionsblock wickelt die Versendung des Datensatzes im korrekten Format über den SHARC Link an den Second Level Buffer ab. Dieses wird von einem Zustandsautomaten geleistet, dessen Zustandsdiagramm in der nachfolgenden Abbildung 4.4 aufgezeichnet ist.

Aus der Abbildung 4.4 entnimmt man, daß der Zustandsautomat, nachdem er vom FCS Handle Block angestoßen wurde (StartTransmission = 1), den kompletten Header verschickt, Zustände TXBYTECOUNTHI bis TXCOMPIDLO. Im Zustand TXFEDIDHI überprüft der Controller, ob der "DATA-VALID"-Eingang schon aktiviert worden ist. Hier ist jetzt der Zeitpunkt, wo wir uns das Pipelining der FADC's wieder ins Gedächtnis rufen, das ja genau 12 Taktzyklen dauert. Der Controller muß also das "DATA-VALID"-Signal, welches zeitgleich mit den Analog Daten anliegt, um genau diese 12 Taktzyklen verzögern, um nicht zu früh angebliche FADC-Daten zu versenden, obwohl noch nicht die korrekten Daten an den Ausgängen der FADC's anliegen. Ist der "DATA-VALID"-Eingang noch nicht aktiviert wor-

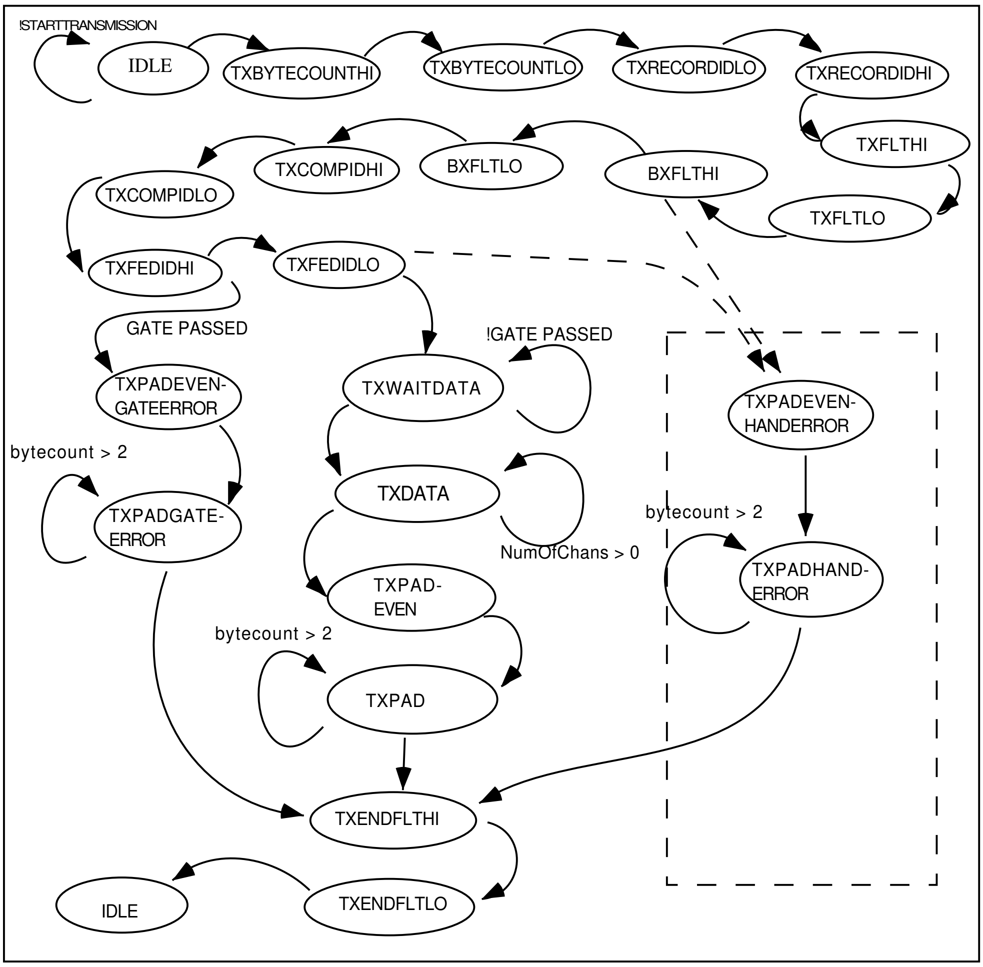

Abbildung 4.4: Das Zustandsdiagramm für den SHARC-Link Interface Block.

den[5], so geht die Zustandsmaschine in einen Wartezustand über, so lange, bis dieses geschehen ist[6] (TXWAITDATA).

Ist dieses erfolgt, versendet der Controller genau die Anzahl an Datenkanälen, die im Register **RoNumOfChans** angegeben ist (TXDATA).

Danach füllt er den Datensatz bis zur nächsten Bytegrenze auf und versendet anschließend solange Nullen, bis die Anzahl der versendeten Byte genau $ByteCount - 2$ entspricht (TXPAD). Zum Abschluß werden die letzten 2 Byte Trailerdaten versendet und die Zustandsmaschine kehrt in den IDLE-Zustand zurück und signalisiert die erneute Bereitschaft, einen Datensatz zu versenden.

[5] Dieses Modul überprüft den Eingang nach Passieren des Schieberegisters.

[6] Im Fall eines sog. AutoTrigger Modus überspringt der Controller diesen Zustand und beginnt sofort Daten zu versenden.

Fehlerbehandlung

Ein verfrühtes "DATA-VALID"-Signal: Es wurden noch Headerdaten versendet, während schon digitalisierte Daten aus der FADC-Pipeline herausgefallen sind. In diesem Fall setzt der Controller das entsprechende Fehlerbit, das EARLYGATEERROR-Bit, und versendet den Fehlercode solange (TXPADGATEERROR), bis inklusive der angehängten Trailerdaten der gesamte Datensatz aufgefüllt ist.

Datenverlust bei der Übertragung zum Second Level Buffer: Um sicherzustellen, daß die Daten auch korrekt an den SLB übermittelt wurden, muß das "LACK"-signal der SHARC-Prozessoren überwacht werden. Ein Fehlerzustand wird vom Linkcontroller erkannt und mit dem "WACK"-Kontrollsignal an den Controller signalisiert.

Da die Versendung der Analogdaten nicht angehalten werden kann, hat ein verspätetes "LACK" unweigerlich den Verlust von Daten zur Folge. Da dieser Fehler zu jedem Zeitpunkt der Übermittlung auftreten, muß er auch zu jedem Zeitpunkt abzufangen sein. Ist der Fehler signalisiert, springt die Zustandsmaschine in einen anderen Fehlerzustand (TXPADHAND), setzt das entsprechende Fehlerbit LATEHANDSHAKEERROR und beginnt, den Fehlercode so lange zu versenden, bis wiederum, inklusive der Trailerdaten, der gesamte Datensatz versendet ist. Hierbei wartet der Controller natürlich bei jedem weiteren Auftreten eines verspäteten "Handshake" solange, bis der SHARC-Prozessor weitere Daten empfangen kann.

Aus Gründen der Übersichtlichkeit sind diese Zustandswechsel nicht im Bubblediagramm vollständig eingezeichnet.

4.3 Implementierung des Bausteines

Die gestiegenen Anforderungen an die Ausleselektronik für heutige Experimente in der Hochenergiephysik (s.Kapitel 2) bedingen immer aufwendigere Entwicklungen. Vielfach sind die hierbei angestrebten Entwicklungen kommerziell nicht oder nur sehr schwer erhältlich, was zumeist mit hohen finanziellen Aufwendungen verbunden ist. Überdies will man auch in der physikalischen Grundlagenforschung weiterhin Tuchfühlung zu den neuesten Entwicklungen der Informationstechnologie halten. Aus diesem Grunde wurde in Heidelberg im Herbst 1994 gemeinsam vom Physikalischen Institut, dem Institut für Hochenergiephysik und dem Max-Planck Institut für Kernphysik ein Entwicklungs- und Testlabor für anwendungsspezifische Halbleiterbausteine[7] gegründet, das sog. ASIC-Labor. Die Aufgabe dieser Einrichtung besteht im Entwurf hochintegrierter Schaltkreise, insbesondere für Experimente in der Hochenergiephysik, unter Verwendung neuester Entwurfswerkzeuge und -methodologien. Gleichzeitig entschied man für das physikalische Institut, dort die gleiche Infrastruktur aufzubauen, um insbesondere auch das

[7] **A**pplication **S**pecific **I**ntegrated **C**ircuits

neuerworbene Know-How des ASIC-Labors hausintern nutzbar zu machen. Die Entwicklung des Front-End Drivers und die der verschiedenen HELIX-Versionen sind die beiden ersten Projekte dieser Zusammenarbeit. Bei der Einrichtung der notwendigen Infrastruktur hat man sich für die neuesten, auf dem Markt erhältlichen Technologien entschieden, die ich im Folgenden kurz vorstellen möchte.

4.3.1 Die Hardwarebeschreibungssprache Verilog

Mitte der 80er Jahre war abzusehen, daß der Trend zu immer höherer Systemintegration im Bereich der Mikroelektronik rasant voranschreiten würde. Insbesondere die Auswirkungen dieser Entwicklung auf die Test- und Wartbarkeit zukünftiger Systeme bereiteten den Experten Kopfzerbrechen. Man erkannte, daß die Entwicklung einer gemeinsamen Test- und Entwicklungsplattform, die gleichzeitig die Möglichkeit eröffnet, ein höheres Abstraktionsniveau bei der Beschreibung elektronischer Systeme zu erreichen, unabdingbar war. Aus dieser Erkenntnis heraus begann die Entwicklung sogenannter Hardwarebeschreibungssprachen. Diesen kommt die Aufgabe zu, die Funktionalität der Schaltung in einer algorithmischen Verhaltensbeschreibung darzustellen. Bei der Entwicklung lehnte man sich bewußt an die Hochsprachen für die Softwareentwicklung an, auch um den Einstieg in die neuen Konzepte zu erleichtern. In den folgenden Jahren setzten sich für Digitalschaltungen zwei Hauptprotagonisten dieser sogenannten HDL's[8] durch: VHDL und VERILOG. Die Entwicklung von VHDL war Mitte der 80er Jahre vom amerikanischen Verteidigungsministerium aufgrund der oben erwähnten Probleme ins Leben gerufen worden, VERILOG HDL hingegen ist eng mit der Firma CADENCE, dem Marktführer für sog. EDA[9]-Systeme, verbunden.

Am Anfang der Entwicklung stand das Ziel, das Verhalten zukünftiger Bausteine zu beschreiben und nachfolgend die Korrektheit des Entwurfes anhand von Simulationen zu überprüfen. Anschließend teilte der Entwickler das Gesamtsystem in Subkomponenten auf und implementierte diese überschaubaren Einheiten per Hand (d.h. in einem Schaltplanentwurf).

Mit der Zeit aber verbesserten sich die Algorithmen und die Rechenleistung der Computer derart, daß die Automatisierung der Entwicklungstufen immer weiter fortschritt.

Wie läßt sich ein Chip mit Hilfe solch einer Hardwaresprache beschreiben?

Für die Entwicklung des Front-End Driver am Physikalischen Institut stand nur Verilog HDL zur Verfügung, so daß ich mich bei meinen Erklärungen hierauf beschränken werde. Die Ausführungen sind aber ohne Weiteres

[8]**H**ardware **D**escription **L**anguage

[9]**E**lectronic **D**esign **A**utomation: Ausdruck für ganze Pakete von Entwicklungswerkzeugen

auf eine VHDL-Implementierung übertragbar.

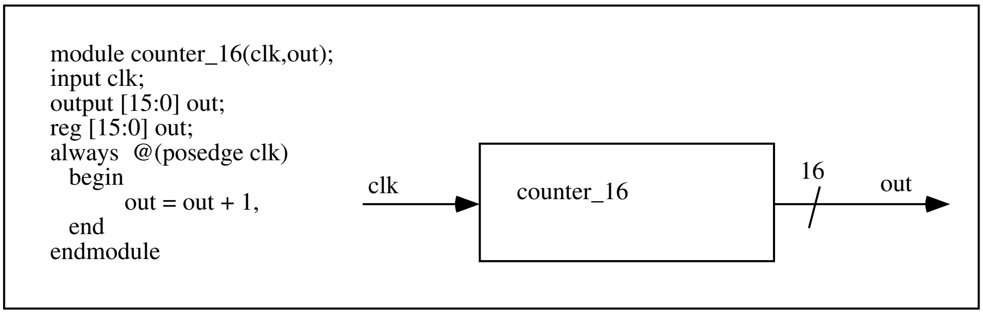

Abbildung 4.5: Ein einfacher 16-Bit Zähler in VERILOG implementiert.

Betrachten wir einen 16-Bit Zähler (s.Abb. 4.5). Früher baute man diese Schaltung aus mehreren diskreten IC's auf. Diese Vorgehensweise mag für einen 16-Bit Zähler praktikabel sein, für einen Mikroprozessor der 32-Bit Klasse ist es schlichtweg unmöglich. Dort wird man versuchen, das System zu modularisieren und diese Subkomponenten zu implementieren.

Wie steht es aber dann mit der Funktionsfähigkeit des Gesamtsystems? Hier wäre eine kompakte Formulierung und die passende Testumgebung vonnöten. Kommen wir zurück zu unserem 16-Bit Zähler. Abbildung 4.5 zeigt die Formulierung des Bausteines in Verilog. Auf drei Tatsachen möchte ich hier aufmerksam machen:

1. **Wiederverwendbarkeit:** Ein einmal geschriebenes Modul, wie z.B. der Zähler, kann, ähnlich den Funktionsbibliotheken herkömmlicher Programmiersprachen, wiederverwendet werden.

2. **Anpassung:** Will man statt des 16-Bit einen 24-Bit Zähler benutzen, so ändert man nur die Breite des Registers und des Ausgangsbusses und schon ist die Anpassung vorgenommen.

3. **Übersichtlichkeit:** Die Zuweisung $out = out + 1$ kommt der mathematischen Abstraktion eines Zählers viel näher als ein komplexes Schaltbild; diese Abstraktion unterstützt die Entwicklung komplexer Systeme.

Hat man jetzt seinen Baustein in der Beschreibungssprache formuliert, so kann man in derselben Sprache die Simulationsumgebung für den Baustein schreiben (s.Abb. 4.6. Man erkennt, daß die Testumgebung sämtliche Signale für den zu beschreibenden Baustein liefert bzw. empfängt. Mit Hilfe dieser sogenannten "Testbank" läßt sich dann das korrekte Verhalten überprüfen.

Abbildung 4.7 zeigt die Testumgebung für die Entwicklung des Controllers. Diese Umgebung besteht aus mehreren Modulen, die die wichtigen Interaktionspartner des zu prüfenden Chips beschreiben. Man erkennt den

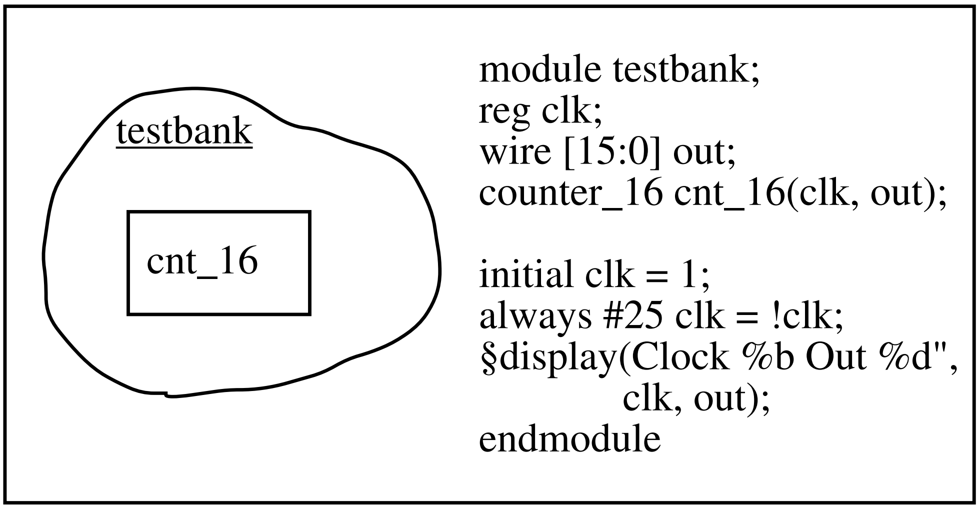

Abbildung 4.6: Das Bild zeigt die Verilog Simulationsumgebung für den 16-Bit Zähler.

Controller **(DL526Ctrl.v)**, den Linkcontroller **(SHARCIf.v)**, den FADC, den Auslesechip und die Fast Control. Alles zusammen ergibt die komplette Testbank für den Chip. Alle Verhaltensimulationen während der Entwurfsphase sind hiermit durchgeführt worden.

4.3.2 Die digitale Synthese eines Entwurfes

Hat man sich in den Anfängen mit der Simulation eines in einer HDL beschriebenen Entwurfes begnügt und die Implementierung per Hand erledigt, so hat sich in den letzten Jahren der Trend zur automatischen Schaltplanumsetzung verstärkt. Heutzutage kann für bestimmte Klassen von Schaltungen der gesamte Weg vom Entwurf bis zum fertigen Layout des Chips automatisch durchgeführt werden. Diese Vorgehensweise gliedert sich in drei Abschnitte:

1. Entwurf des Chips mittels einer HDL und Simulation der Funktionalität in einer adäquaten Testumgebung.

2. Synthese des zugehörigen Schaltplanes unter Verwendung einer Zieltechnologiespezifischen Teilebibliothek und des entsprechenden Syntheseprogrammes.

3. Auslegen ("Layout") des endgültigen Chips unter Berücksichtigung der verfügbaren Ressourcen ("Placing and Routing").

Diese drei Abschnitte sind in der folgenden Abbildung 4.8 visualisiert. Durch die Einführung der Hardwarebeschreibungssprache ist es jetzt aber auch

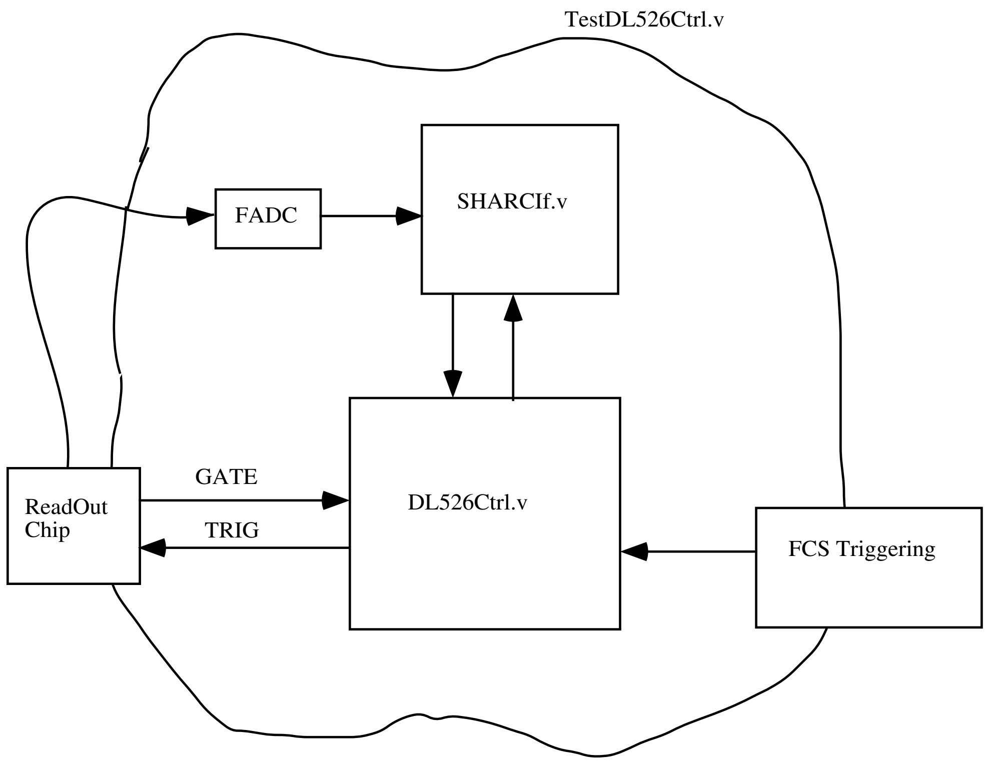

Abbildung 4.7: Das Bild gibt die Verilog Simulationsumgebung für den Entwurf des Controllerchips wieder.

möglich, jede dieser Entwicklungsstufen unter der gleichen Testumgebung (TESTBANK) zu simulieren, was die Wahrscheinlichkeit von endgültigen Entwurfsfehlern erheblich unterdrückt. Auf der letzten Stufe, der sog. Layout-Ebene, stehen sogar die internen Laufzeiten des Chips für die Simulationen zur Verfügung, so daß eine sehr wirklichkeitsnahe Simulation des Bausteines möglich ist. Die Vorteile dieser Entwurfsmethodologie liegen auf der Hand:

- **Testbarkeit:** Durch sorgfältiges Testen läßt sich ein Großteil aller möglichen Fehler schon vor der Implementierung vermeiden.
- **Flexibilität:** Entwurfsänderungen können sehr schnell implementiert und auch getestet werden (insbesondere programmierbare Bausteine sind hierfür geeignet).
- **Abstraktion:** Ein hoher Abstraktionsgrad erleichtert den Übergang zu den notwendigen, höchsten Integrationsdichten.

Nachteile

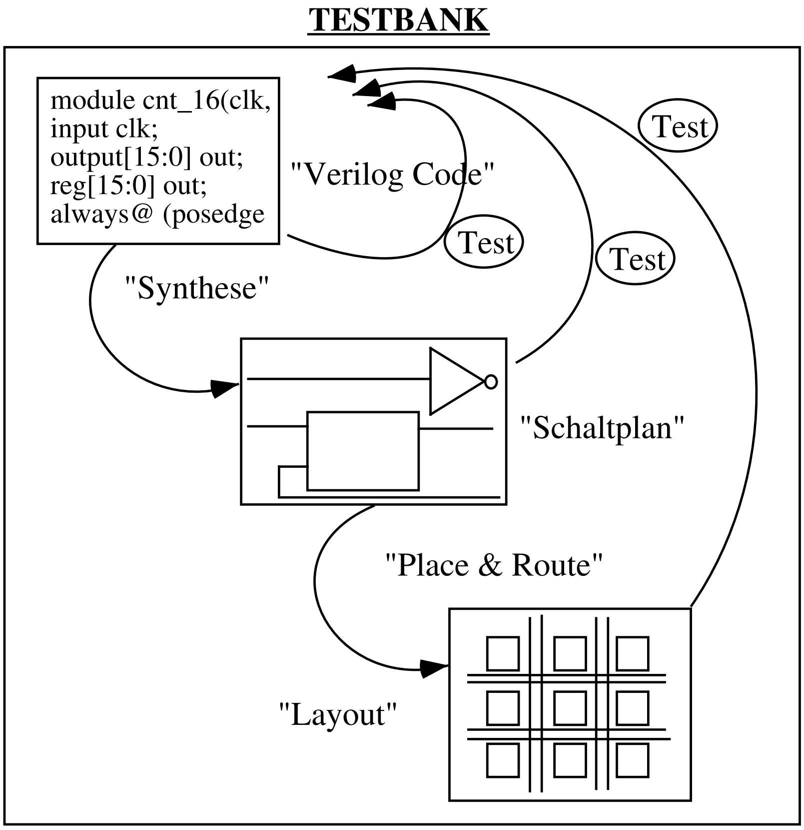

Abbildung 4.8: Vorgehen bei der Implementation eines integrierten Bausteines.

Die Qualität des endgültigen Bausteins steht und fällt mit den Synthese- und Layoutwerkzeugen und den verwendeten Teilebibliotheken. Gerade Letzteres hatte während dieser Arbeit erhebliche Probleme bereitet. Ein dreiviertel Jahr stand keine funktionstüchtige Teilebibliothek zur Verfügung, wir mußten sie selbst entwickeln, sonst hätte der Controller nie in diesem Zeitrahmen fertiggestellt werden können.

Die heutzutage erhältlichen Synthesewerkzeuge können nur eine Untermenge des Sprachumfanges der HDL's effektiv in funktionstüchtige Schaltkreise umsetzen. All dieses kostet Zeit, d.h. man braucht gut ausgebildete Schaltungsdesigner, um zu einer schnellen Entwurfsumsetzung zu gelangen.

Kapitel 5

Test zur Inbetriebnahme der Karte

Das FED-Board wurde sukzessive in Betrieb genommen. Diese Tests gliederten sich in 3 Phasen:

1. Allgemeine Tests des Controllers; Initialisierung der Bausteine, Reset in den Anfangszustand.

2. Überprüfung der Verbindung FED $\longleftrightarrow$ SHARC unter Verwendung des Betriebsmodus **ROMODE_TESTTRIGGER** .

3. Inbetriebnahme der Schnittstelle zum Auslesechip und Test der FADC's im Betriebsmodus **ROMODE_ADCTRIGGER**.

4. Test der Schnittstelle zur Fast Control und der Korrektheit der Triggersequenz für den Auslesechip.

Für den gesamten Zeitraum der Inbetriebnahme konnten wir denselben Testaufbau benutzen. Er ist in Abbildung5.1 dargestellt. Die zentrale Steuereinheit dieses Aufbaus ist ein Personal Computer. Dieser PC ist mit einer Erweiterungskarte, die mit zwei SHARC-Prozessoren (ADSP-21062) bestückt ist, ausgerüstet. Desweiteren umfaßt die vorhandene Software das komplette Entwicklungssystem für diese digitalen Signalprozessoren. Der PC bedient folgende Aufgaben:

1. Laden der Konfigurationsdatei für den Xilinx-Baustein nach einem Neustart des Gesamtsystems.

2. Initialisierung des Controller-Bausteines.

3. Steuerung der Auslesesequenzen, entweder über den direkten Zugriff auf die Karte im Autotriggermodus, oder über externe Trigger.

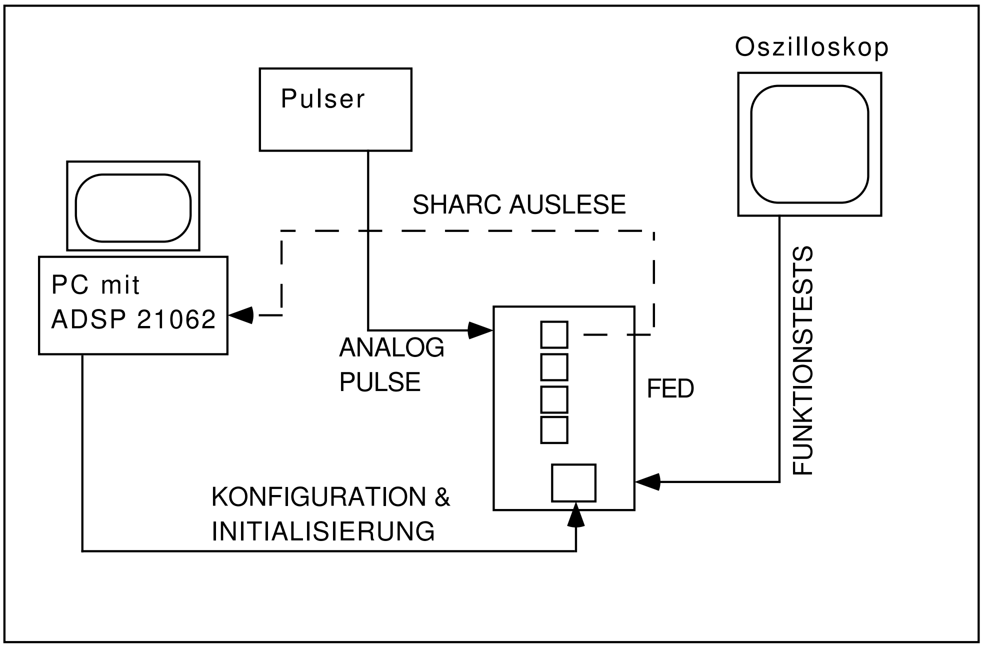

Abbildung 5.1: Das ist ein schematisches Abbild des Testaufbaus zur Inbetriebnahme.

4. Emulation des Second Level Buffer mit Hilfe der SHARC-Prozessoren auf der PC-Einsteckkarte der Firma Perimos(PM60 Erweiterungskarte).

5.1 Allgemeine Funktionstests

Nachdem die Karte mit den Hauptkomponenten bestückt war, begannen die ersten Funktionstests. Diese umfaßten rein elektrische Tests, aber auch die Überprüfung der wichtigen Systemsignale anhand des folgenden Fragenkataloges war durchzuführen.

- Ist die Phasenlage der verschiedenen Systemtakte korrekt?
- Verharrt die Karte nach einem Reset in ihrem Anfangszustand?

Nach dem ersten Einschalten der Karte stellte sich heraus, daß die benutzte PLL-Schaltung nicht korrekt funktionierte. Seit der Überarbeitung der Schaltung ist dieses Problem behoben.

Die Klärung der zweiten Frage nahm weitaus mehr Zeit in Anspruch. Der erste Entwurf der Controllerschaltung war funktionsuntüchtig, da die Karte ohne Trigger anfing, Datensätze zu versenden. Der Entwurf für den Controller wurde in der Folgezeit noch zweimal grundlegend geändert, bis eine lauffähige Version gefunden war. Hierbei stellte sich heraus, daß funktional identische Beschreibungen, die aber unterschiedlich formuliert waren, zu unterschiedlichen Schaltungen synthetisiert wurden, die sich auch in ihrem Zeitverhalten unterschieden. Für die aufwendige Zustandsmaschine des SHARCLink Interface Modules (s. Abschnitt 4.2.2) hat sich die Formulierung mittels einer expliziten Zustandsmaschine auf Register-Transfer Niveau (RTL: Register-Transfer-Level) als die Zweckmäßige herausgestellt. Diese Entwurfsversion ist seitdem im Betrieb.

5.2 Test der Verbindung FED $\longleftrightarrow$ Second Level Buffer

Funktionstests der Verbindung

Für diese Tests war insbesondere die Beantwortung der folgenden Fragen gefordert:

- Sendet der Front-End Driver Datensätze korrekter Länge ab?
- Ist der Wechsel zwischen 8- und 10-Bit Daten korrekt?
- Wie verhält sich der SHARC-Link im realen Betrieb? Kann man den Link mit der maximalen Bandbreite betreiben?

Für die Funktionstests sind verschiedene Signale auf dem Front-End Driver interessant:

- Die Auslesefrequenz, Readoutclock
- Der Systemtakt des Linkcontrollers, die Linkclock
- Das "Handshake"-Signal des SHARC-Prozessors, "LACK"
- Das Startsignal des Controllers für die Versendung von Daten, "notLnkTrippleStart" (s.Anhang: Source Code des Controllers)

Um die Länge des Datensatzes zu ermitteln, haben wir auf dem Oszilloskopbildschirm das "notLnkTrippleStart"-Signal zusammen mit der Auslesefrequenz dargestellt. Die Anzahl der Taktzyklen der Auslesefrequenz während des Anliegens des "notLnkTrippleStart"-Signals gibt die Zahl der versendeten Byte wieder. Hat man z.B. 36 Taktzyklen gezählt, so sind genau 36 Byte in einem Datensatz versendet worden. In Abbildung 5.2 sind oben das "notLnkTrippleStart"-Signal und unten die Auslesefrequenz dargestellt. An dem unteren Signal lassen sich genau 36 Taktzyklen abzählen, wie es für diesen Test erwartet war.

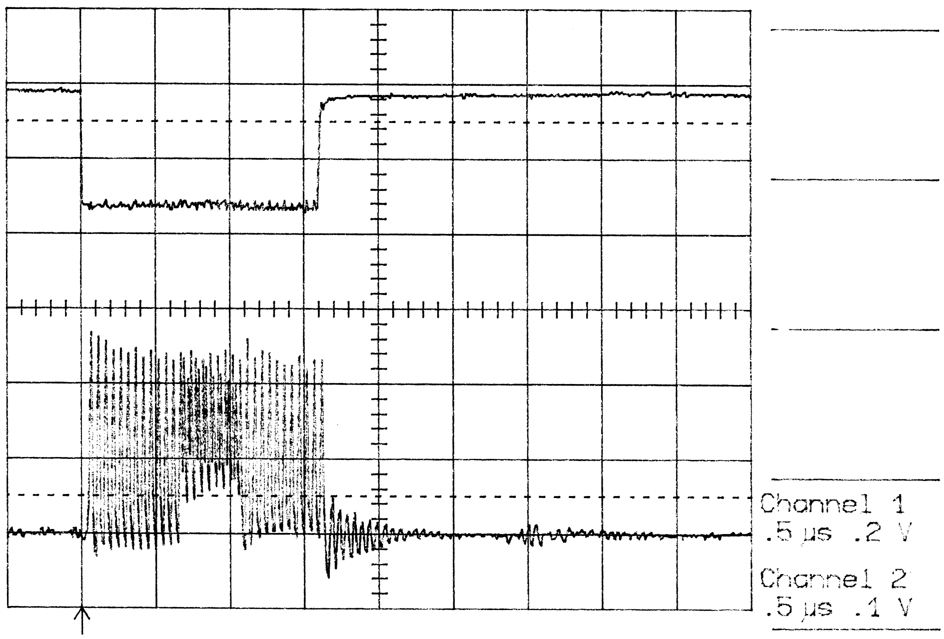

Abbildung 5.2: Dieses Bild zeigt die Übertragung von 36 Byte über den SHARC Link. [FEUERSTACK]

Will man hingegen den Wechsel zwischen 8- und 10-Bit Daten überprüfen, so muß die Linkclock hinzugenommen werden. Erinnern wir uns: der Linkcontroller durchläuft einen 3-taktigen Zyklus zur Versendung der Daten. Bei 10-Bit Daten treibt die Linkclock alle drei Takte den Link, bei 8-Bit Daten hingegen bleibt im letzten Zustand eines jeden Zyklus' die Linkclock inaktiviert. Dieses sollte an einem "Aussetzer" der Linkclock alle drei Takte zu erkennen sein. Ein Beispiel hierfür ist in Abbildung 5.3 bei dem obersten Signal dargestellt. Man erkennt das Aussetzen der Linkclock alle 3 Takte, d.h die Versendung von 8-Bit Daten, und den Wechsel zu 10-Bit Daten, wenn alle 3 Takte der Link getrieben wird.

Bei der Inbetriebnahme der SHARC Links gab es Überraschungen. Die Frage, die uns am brennendsten interessierte war, unter welchen Bedingungen das LACK-Signal des SHARC-Prozessors zu spät bestätigt wird. Oder anders formuliert:

Wo liegt die Grenze für die Bandbreite der Links und wie häufig sind auftretende Fehler?

Da wir den SHARC-Prozessor in C programmieren wollten, die andere Option wäre SHARC-Assembler gewesen, und der C-Compiler von Analog Devices nur 32-Bit Worte verarbeiten kann, mußten wir den SHARC-Link im 32-Bit Modus betreiben, d.h. wir übertragen immer acht Nibble in einem 32-Bit Wort. In Abbilddung5.3 sind die LinkClock (LCLK), das Handsha-

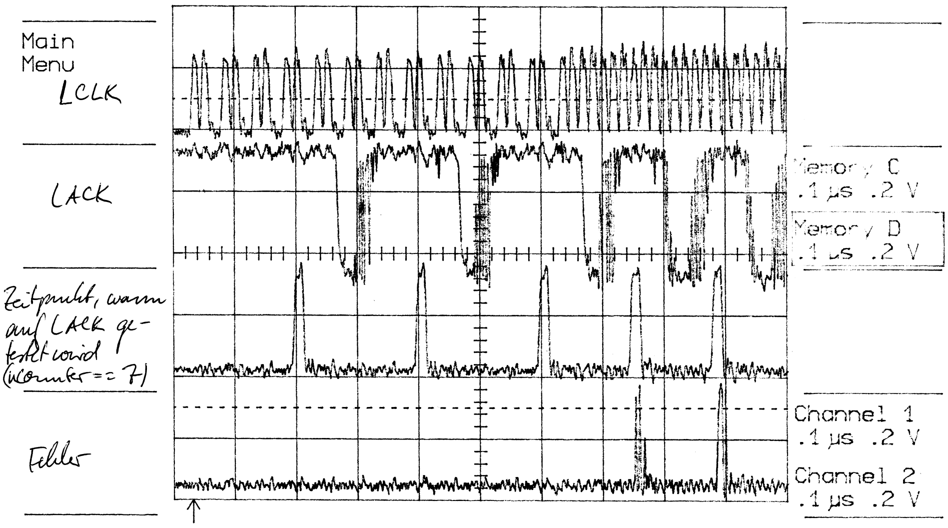

Abbildung 5.3: In diesem erkennt man eine fehlerhafte Übertragung der Daten. [FEUERSTACK]

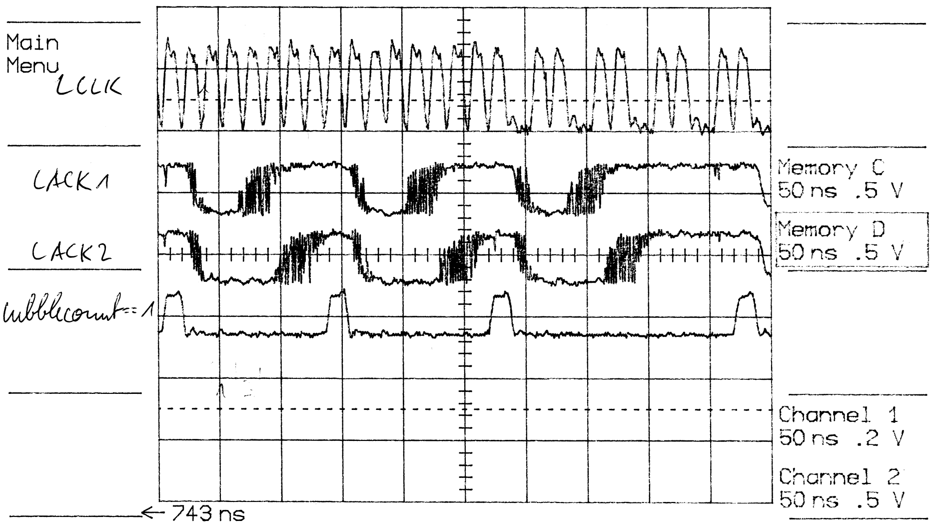

Abbildung 5.4: Der SHARC Prozessor arbitriert das LACK Signal bei der Benutzung mehrerer Link Ports. [FEUERSTACK]

kesignal (LACK), der Zeitpunkt der Versendung des 8.Nibbles (nibblecount == 7)[1] auf den LACK getestet wird und die Verletzung der Bedingung (nibblecount == 7 && LACK == 1) wiedergegeben. Es verwundert, daß Fehler auftreten, obwohl augenscheinlich die Bedingung hierfür nicht verletzt zu sein scheint, da zu dem Zeitpunkt, an dem nibblecount==7 ist, LACK==1 gilt.

[1]nibblecount läuft von 0 bis 7, d.h es ist das letzte der 8 Nibbles

Wir betreiben nicht nur einen einzigen SHARC Link, sondern mehrere. Sobald aber mehr als ein Link eines Prozessors genutzt wird, muß der Prozessor die Entleerung der verschiedenen Linkbuffer zeitlich hintereinanderreihen. Damit wird aber auch das "LACK"-Signal für den zweiten und alle weiteren Links später auf "1" gezogen als bei dem 1.Link. Die Abfrage auf das "LACK"-Signal muß also genau auf den Zeitpunkt abgestimmt werden, an dem alle benutzten Links wieder frei sind (s.Abb.5.4). Man sieht aber auch, daß der Bereich für den Zeitpunkt der Abfrage sehr schmal ist, was vermuten läßt, daß man wohl mit 2 Links die volle Bandbreite wird nutzen können, die Verwendung weiterer Links aber zu Performanceeinbußen führen kann.

Auslese von Daten

Bei diesen Tests wurde von der Möglichkeit Gebrauch gemacht, ein bestimmtes Testmuster an den Second Level Buffer zu versenden. Bei der Initialisierung des Controller, aber auch zu jedem beliebigen, späteren Zeitpunkt, kann dieses Testmuster geladen werden, welches dann in einem 10-Bit Schieberegister rotiert.

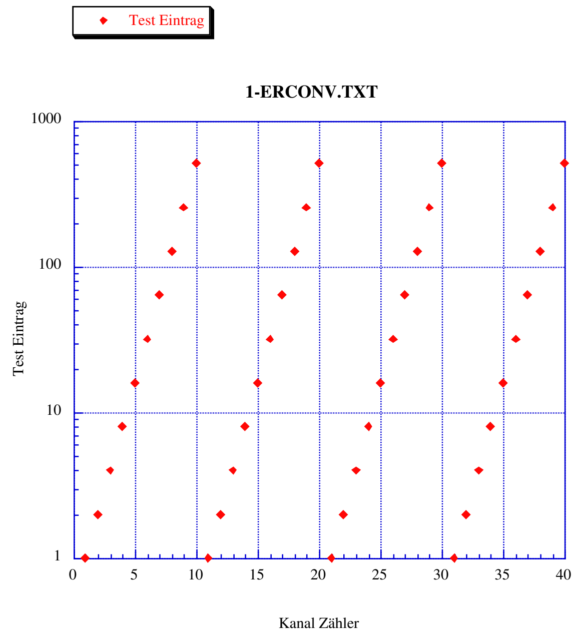

Abbildung 5.5: Versendung einer Eins als Testmuster.

Nach der Übermittlung wurden die Daten vom SHARC-Prozessor in einer Datei abgelegt. In der Datei wurde ein gesamter Second Level Buffer Datensatz als ein String abgelegt. Dieser String besteht aus 32-Bit Wörtern, dem Inhalt eines Linkbuffers, die durch Leerzeichen getrennt sind. Die 32-Bit Worte sind als hexadezimale Zahlen dargestellt. Im Anhang ist ein Pro-

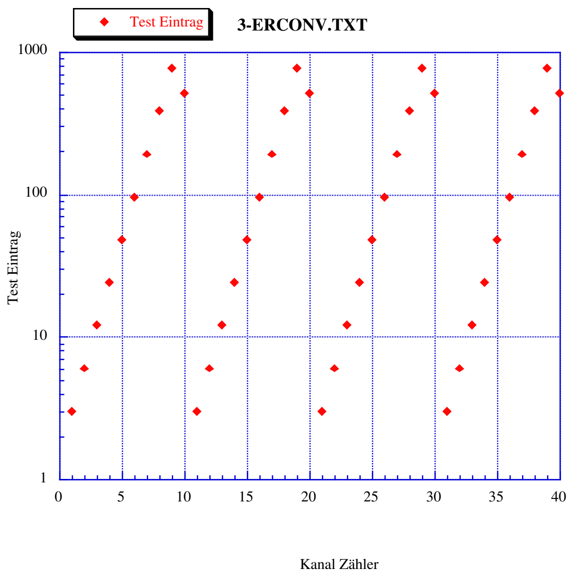

Abbildung 5.6: Versendung einer Drei als Testmuster.

gramm wiedergegeben, das aus diesem String die Headerinformationen und die übertragenen Test- bzw. Kanaldaten extrahiert. Hierbei geht das Programm in den folgenden Schitten vor:

- Trenne Headerinformationen von dem restlichen Datensatz.
- Extrahiere jeweils 16-Bit Worte und konvertiere sie zu dezimalen Werten, den Headerdaten.
- Lese jeweils 12-Bit Worte ein, die Kanaldaten, und konvertiere sie zu dezimalen Werten.
- Lege jeden extrahierten Wert mit einer Markierung (Kanalzähler) in einer gesonderten Datei ab.

Das Ergebnis läßt sich dann in dieser Formatierung leicht mit einem Graphikprogramm, z.B. Kaleidagraph, in einem Diagramm darstellen. Zuerst wurde das Register mit dem Wert 1 beschrieben (die übermittelten Werte sind dann: 1,2,4,...,256,512,1 etc.). Das rotierende Muster zeigt dann eine für das jeweilige Muster charakteristische Kennlinie. In dem Diagramm (s.Abb. 5.5) erkennt man, daß das Testmuster korrekt übertragen wurde, die Werte lassen sich an der Y-Achse ablesen.

Um die Tests der korrekten Übertragung abzusichern, wurden noch eine Reihe weiterer Testmuster (z.B. die Zahlen 3, s.Abb.5.6) übermittelt, die alle korrekt extrahiert werden konnten. Das ließ die Schlußfolgerung zu, daß in der Tat die Datenübertragung und die Auslese der empfangenen Daten korrekt ist.

5.3 Aufnahme von analogen Testpulsen

Der nächste Test bestand in der Inbetriebnahme der FADC's. Hierzu wurde der Testaufbau dahingehend erweitert, daß ein Pulser die Analogpulse an den Analogeingang des FED lieferte (s.Abb. 5.1. Der Pulser wurde über den "Trigger"-Ausgang des FED getriggert, um synchronisierte Testpulse an die Analogeingänge des FED abzugeben.

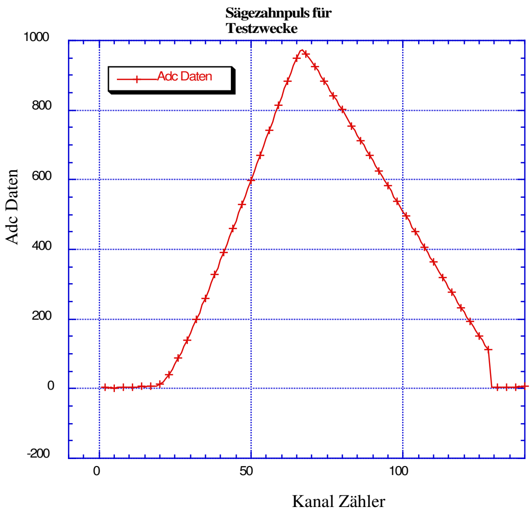

Abbildung 5.7: Ein sauber digitalisierter Sägezahn.

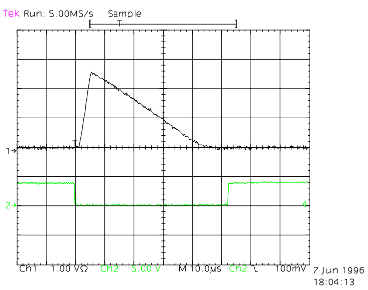

Abbildung 5.8: Das Einganssignal der fallenden Flanke als Testpuls.

Es wurden zwei verschiedene Tests durchgeführt:

- Vom Pulser wurde ein Sägezahnpuls erzeugt (s.Abb. 5.7), der die Länge eines Datensatzes andauern und den dynamischen Bereich der FADC's ausnutzen sollte.
- Eine lineare, fallende Flanke wurde vom Pulser erzeugt (s.Abb. 5.8), die den dynamischen Bereich der FADC's überschreiten sollte. In diesem Fall solltedie Linearitätder FADC's geprüft werden.

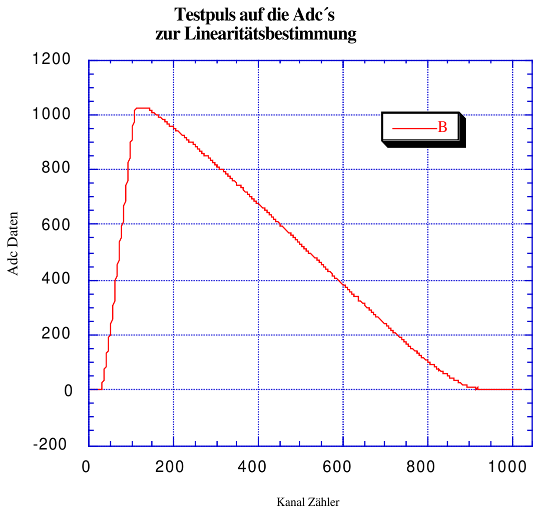

Abbildung 5.9: Dieser Plot zeigt den ausgelesenen Testpuls zur Überprüfung der Linearität.

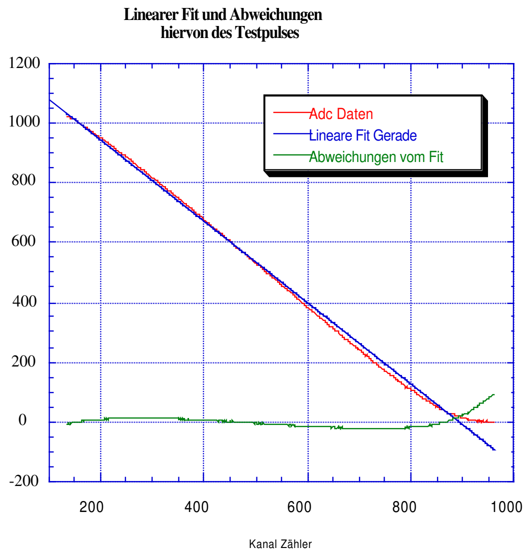

Abbildung 5.10: Diese Kurven zeigen einen linearen Fit an die fallende Flanke des obigen Testpulses unde die Abweichungen davon.

In der Abbildung 5.7 ist das Diagramm für den ausgelesenen Sägezahn wiedergegeben, der allerdings bei Kanal 128 noch nicht auf Null gefallen war, so daß hier, mit Beginn des nächsten Datensatzes, der Puls in der graphischen Wiedergabe abgeschnitten ist. Beruhigend ist die Tatsache, daß die Pulsform keine besonderen Sprünge aufweist.

Auf den Eingang des FADC haben wir eine lineare Flanke als Testpuls gegeben (s.Abb. 5.8). Nach dem Auslesen der Daten aus dem SHARC-Prozessor und der Extraktion der Kanaldaten stellt sich der Puls wie in Abbildung 5.9 dar. Man sieht ganz deutlich, daß der dynamische Bereich ausgenutzt wurde, die Kurve ist nach oben hin abgeschnitten, wie es zu erwarten war.

Zur Überprüfung der Linearität der FADC's wurde in die fallende Flanke des Pulses eine Gerade hineingelegt. Durch Subtraktion der Geraden von der ausgelesenen Kurve (s.untere Kurve) kann die Qualität der Digitalisierung überprüft werden.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde der Prototyp des Analogen Front End Driver Boards für die MSGC's und den Silizium Vertex Detektor bei HERA-B entwickelt und in Betrieb genommen. Der Hauptgegenstand der Arbeit war die Entwicklung des zentralen Controller Bausteines und dessen Implementierung in einem programmierbaren Chip (FPGA).

Zu Anfang wurde ein Konzept für den Analogen Front End Driver entwickelt. Dieses geschah im Rahmen der HERA-B Kollaboration und in Zusammenarbeit mit der Elektronikwerkstatt im Physikalischen Institut.

Anhand diese Konzeptes wurden dann die Aufgaben für den Controller definiert. Anschließend ist die Funktion des Controllers in der Hardwarebeschreibungssprache VERILOG HDL beschrieben worden. Einen wesentlichen Teil der Arbeit nahm die gründliche Simulation des Controllers ein, um ein späteres Fehlverhalten zu vermeiden.

Im nächsten Schritt sollte der Controller in einem programmierbaren Chip der Firma XILINX implementiert werden. Hierbei sollte das CADENCE-Entwicklungssystem und das XACT-Programm der Firma XILINX verwendet werden. Da zu Beginn der Arbeit keine funktionierende Teilebibliothek für diese Chips für das CADENCE-System zur Verfügung stand, mußte eine entsprechende Bibliothek entworfen werden, mit der dann sämtliche weiteren Entwicklungen im Laufe dieser Arbeit durchgeführt wurden.

Der letzte Teil der Arbeit bestand dann in der Inbetriebnahme des Analogen Front End Driver Boards in einem geeigneten Testaufbau. Mit diesen Tests konnte die Funktionstüchtigkeit des Boards unter Beweis gestellt werden. Die ersten Prototypen sollen im Sommer 1996 für die Tests der beiden Subdetektoren verwendet werden.

Für den endgültigen Entwurf des Analogen Front End Drivers, der in der zweiten Hälfte des Jahres 1996 geplant ist, sind einige Änderungen vorzunehmen:

- Korrektur der Fehler im Layout der Karte
- Benutzung von 8-Bit FADC's: Das Protokoll des Linkcontrollerchips wird hiervon beeinflußt. Außerdem führt der Einsatz solcher Bausteine zu einer wesentlichen Kostenreduktion.
- Kostengünstigere Implementierung des Controllers: Zur Zeit beträgt der Preis für den Controllerchip ungefähr die Hälfte des Preises der gesamten Karte. Für die endgültige Lösung wird man eine kostengünstigere Alternative suchen.

Anhang A

Schaltpläne

In diesem Anhang sind alle Schalt- und Bestückungspläne [RAUSCH] & [RUSNYAK] für das FED Board zusammengefaßt:

- Der Schaltplan für die PLL-schaltung zur Synchronisation der verschiedenen Taktfrequenzen (Seiten 1 - 2)
- Der Schaltplan für die Platine auf der die Steuersignale zum Auslesechip an die Frontplatte geführt sind (Seite 3)
- Die Schaltpläne für das Front End Driver Board (Seiten 4 - 8)
- Die Bestückungspläne für die Front- und Rückseite des Front End Driver Boards (Seiten 9 - 10)

Anhang B

Die Quellcodes für die einzelnen Bausteine der Karte

B.1 Der Source Code für den Linkcontroller Chip

```
module Dl526_36
title ' h1 link_nibbler A. Rausch';

"Edition history:
" 96/04/03 MF       Nibble counter reduced to 3bit counter because we use
"                   32bit = 8nibble mode of SHARC links.
" 96/04/04 MF       Nibble counter clock is identical to link clock out,
"                   thus it is always incremented when the link is clocked
" 96/04/05 MF       Test design that ignores LNKACK and always sets WACK
"                   to one (i. e. it never jumps into an error state).
"                   The time when it samples LNKACK is output to TESTOUT1
"                   The result of testing LNKACK when is output to TESTOUT2
" 96/04/11 CP & AR  State machine is changed to error handling within all
"                   of the three nibble states
" 96/04/12 CP & MF  LSB swapping removed
" 96/04/14 AR & CP  TESTCOUNTER zur Verlngerung von LNKACK eingebaut.
" 96/05/01 CP & AR  In state lowelownibble the expression badlnkack
"                   substituted with _LNKACK

"      DL526_36 device 'i1032j08';

      PLSI property 'EXTENDED_ROUTE on';
      PLSI property 'STRATEGY delay';
      PLSI property 'PULLUP on';

      declarations
```

```
"inputs
LNKCLK                    pin 12;
PLSI property 'CLK LNKCLK FASTCLK';

ADCA9..ADCA0              pin 45,46,47,48,49,50,51,52,53,54;
ADCB9..ADCB0              pin 55,56,57,58,59,60,68,69,70,71;
ADCC9..ADCC0              pin 72,73,74,75,76,77,78,79,80,81;
HEAD9..HEAD0              pin 82,83,3,4,5,6,7,8,9,10;
_LNKSTART                 pin 39;
_ADCOE,_HEADOE            pin 67,11;
LNKACKA,LNKACKB,LNKACKC   pin 19,2,84;

PLSI property 'PULLUP  LNKACKA';
PLSI property 'PULLUP  LNKACKB';
PLSI property 'PULLUP  LNKACKC';

"outputs
nibA3..nibA0              pin 33,32,31,30 istype 'reg_d,buffer';
nibB3..nibB0              pin 29,28,27,26 istype 'reg_d,buffer';
nibC3..nibC0              pin 18,17,16,15 istype 'reg_d,buffer';

WACKOUT                   pin 38;
LNKCLKA,LNKCLKB,LNKCLKC   pin 34,14,13;
TESTOUT2                  pin 41 istype 'com';

"internal
IOCLK,_LNKACK             node    istype 'com';
badlnkack                 node    istype 'com';

PLSI property 'CLK IOCLK IOCLK0';
LNKCLK1,LNKCLK2,LNKCLK3   node    istype 'com';

PLSI property 'PRESERVE LNKCLK1';
PLSI property 'PRESERVE LNKCLK2';
PLSI property 'PRESERVE LNKCLK3';

nibcnt0,nibcnt1,nibcnt2                  node    istype 'reg_d,buffer';
error4,error3,error2,error1,gate,wack    node    istype 'reg_d,buffer';

qadcA9..qadcA0              node      istype 'reg_d,buffer';
qadcB9..qadcB0              node      istype 'reg_d,buffer';
qadcC9..qadcC0              node      istype 'reg_d,buffer';
qhead9..qhead0              node      istype 'reg_d,buffer';
_QADCOE,_QHEADOE            node      istype 'reg_d,buffer';
cnt2_dly,cnt0_dly,cnt1_dly  node      istype 'reg_d,buffer';
"       testcnt2,testcnt1,testcnt0      node    istype 'reg_d,buffer';
"       TESTWAIT                        node    istype 'com';

H,L,X,C = 1,0,.X.,.C.;
```

```
      nibblecount  =  [nibcnt2,nibcnt1,nibcnt0];
      mode    =  [_QADCOE,_QHEADOE];

      test    =  [0,0];            "nibble header 3 * 4bit
      adc     =  [0,1];            "nibble adc    3 * 4bit
      head    =  [1,0];            "nibble header 2 * 4bit

"             TESTCNT = [testcnt2,testcnt1,testcnt0];

      QAL      = [qadcA1.q,qadcA0.q,0,0];
      QAM      = [qadcA5..qadcA2].q;
      QAH      = [qadcA9..qadcA6].q;

      QBL      = [qadcB1.q,qadcB0.q,0,0];
      QBM      = [qadcB5..qadcB2].q;
      QBH      = [qadcB9..qadcB6].q;

      QCL      = [qadcC1.q,qadcC0.q,0,0];
      QCM      = [qadcC5..qadcC2].q;
      QCH      = [qadcC9..qadcC6].q;

      QHL      = [qhead1.q,qhead0.q,0,0];
      QHM      = [qhead5..qhead2].q;
      QHH      = [qhead9..qhead6].q;

      NIBA     = [nibA3.. nibA0];
      NIBB     = [nibB3.. nibB0];
      NIBC     = [nibC3.. nibC0];

"             outset_dly = [cnt2_dly,cnt1_dly,cnt0_dly];

      inset            = [LNKCLK,_LNKSTART];
      outset           = [error4,error3,error2,error1,gate,wack];
      null             = [0       ,0      ,0     ,0     ,0   ,0   ];
      nonibble         = [0       ,0      ,0     ,0     ,0   ,1   ];
      hinibble         = [0       ,0      ,0     ,1     ,0   ,1   ];
      midnibble        = [0       ,0      ,1     ,0     ,0   ,1   ];
      lownibble        = [0       ,0      ,1     ,1     ,0   ,1   ];

      hiehinibble      = [0       ,0      ,0     ,1     ,1   ,0   ];
      hiemidnibble     = [0       ,0      ,1     ,0     ,1   ,0   ];
      hielownibble     = [0       ,0      ,1     ,1     ,1   ,0   ];

      midehinibble     = [0       ,1      ,0     ,1     ,1   ,0   ];
      midemidnibble    = [0       ,1      ,1     ,0     ,1   ,0   ];
      midelownibble    = [0       ,1      ,1     ,1     ,1   ,0   ];

      lowehinibble     = [1       ,0      ,0     ,1     ,1   ,0   ];
```

```
      lowemidnibble    = [1        ,0       ,1      ,0      ,1    ,0    ];
      lowelownibble    = [1        ,0       ,1      ,1      ,1    ,0    ];


      state_diagram outset;
      state null:      goto                   nonibble;
      state nonibble:  case _LNKSTART:        nonibble;
                           !_LNKSTART:        hinibble;
                       endcase;
      state hinibble:  case badlnkack:        hiehinibble;
                           !badlnkack:        midnibble;
                       endcase
      state midnibble: case badlnkack:        midehinibble;
                           !badlnkack:        lownibble;
                       endcase
      state lownibble: case badlnkack & !_LNKSTART:    lowehinibble;
                           badlnkack & _LNKSTART:     nonibble;
                           !badlnkack & !_LNKSTART:   hinibble;
                           !badlnkack & _LNKSTART:    nonibble;
                       endcase
      state hiehinibble:  goto                hiemidnibble;
      state hiemidnibble: goto                hielownibble;
      state hielownibble: case !_LNKACK:      hiehinibble;
                                _LNKACK :     midnibble;
                          endcase;
      state midehinibble:  goto                 midemidnibble;
      state midemidnibble: goto                 midelownibble;
      state midelownibble: case !_LNKACK:       midehinibble;
                                _LNKACK:        lownibble;
                           endcase;
      state lowehinibble:  goto                          lowemidnibble;
      state lowemidnibble: goto                          lowelownibble;
      state lowelownibble: case _LNKACK & !_LNKSTART:  lowehinibble;
                                _LNKACK & _LNKSTART:   lowehinibble;
                                !_LNKACK & !_LNKSTART: hinibble;
                                !_LNKACK & _LNKSTART: nonibble;
                           endcase

      equations

"MF: Increment counter only when link is clocked
      nibblecount.d      =   (nibblecount.q + 1);
      nibblecount.clk    =   LNKCLK2;

"             TESTOUT1            = !TESTWAIT;
              TESTOUT2            = (nibblecount == 1);

      WACKOUT      = 0;
      WACKOUT.oe   = !wack;
```

```
badlnkack    = (nibblecount == 1) & !_LNKACK;

outset.clk   = LNKCLK;
IOCLK        =   LNKCLK & (outset == nonibble)
               # LNKCLK & (outset == lownibble)
               # LNKCLK & (outset == lowelownibble);

[qadcA9..qadcA0].d         = [ADCA9..ADCA0];
[qadcA9..qadcA0].clk       = IOCLK;

[qadcB9..qadcB0].d         = [ADCB9..ADCB0];
[qadcB9..qadcB0].clk       = IOCLK;

[qadcC9..qadcC0].d         = [ADCC9..ADCC0];
[qadcC9..qadcC0].clk       = IOCLK;

[qhead9..qhead0].d         = [HEAD9..HEAD0];
[qhead9..qhead0].clk       = IOCLK;

_QADCOE.d              = _ADCOE;
_QADCOE.clk            = IOCLK;
_QHEADOE.d             = _HEADOE;
_QHEADOE.clk           = IOCLK;


NIBA.d =  QAH & (mode == adc)  & (outset == hinibble)
        # QAM & (mode == adc)  & (outset == midnibble)
        # QAL & (mode == adc)  & (outset == lownibble)

        # QHH & (mode == test) & (outset == hinibble)
        # QHM & (mode == test) & (outset == midnibble)
        # QHL & (mode == test) & (outset == lownibble)

        # QHH & (mode == head) & (outset == hinibble)
        # QHM & (mode == head) & (outset == midnibble)
        # QHL & (mode == head) & (outset == lownibble);
        NIBA.clk = LNKCLK;

NIBB.d =  QBH & (mode == adc)  & (outset == hinibble)
        # QBM & (mode == adc)  & (outset == midnibble)
        # QBL & (mode == adc)  & (outset == lownibble)

        # QHH & (mode == test) & (outset == hinibble)
        # QHM & (mode == test) & (outset == midnibble)
        # QHL & (mode == test) & (outset == lownibble)

        # QHH & (mode == head) & (outset == hinibble)
        # QHM & (mode == head) & (outset == midnibble)
        # QHL & (mode == head) & (outset == lownibble);
```

```
        NIBB.clk = LNKCLK;

NIBC.d =  QCH & (mode == adc)  & (outset == hinibble)
        # QCM & (mode == adc)  & (outset == midnibble)
        # QCL & (mode == adc)  & (outset == lownibble)

        # QHH & (mode == test) & (outset == hinibble)
        # QHM & (mode == test) & (outset == midnibble)
        # QHL & (mode == test) & (outset == lownibble)

        # QHH & (mode == head) & (outset == hinibble)
        # QHM & (mode == head) & (outset == midnibble)
        # QHL & (mode == head) & (outset == lownibble);
        NIBC.clk = LNKCLK;

        !_LNKACK    =  !LNKACKA # !LNKACKB # !LNKACKC;

LNKCLK1 =     LNKCLK & (mode == test) & (outset == lownibble)
            # LNKCLK & (mode == adc)  & (outset == lownibble)
            # LNKCLK & (outset == midnibble)
            # LNKCLK & (outset == hinibble)
            # gate;

LNKCLK2          = LNKCLK1;  " 1. dummy for 8ns delay (no funktion)!!!
!LNKCLK3         = LNKCLK2;  " 2. dummy for 8ns delay (O.K.)!
LNKCLKA          = LNKCLK2;
LNKCLKB          = LNKCLK2;
LNKCLKC         = LNKCLK2;
```

Anhang C

Registerbild des Controllers

Für den Betrieb des Front-End Driver muß der Controller korrekt initialisiert werden. Zur Ablage der Daten ist ein Registerblock von 23 Registern vorgesehen. Sie können über einen VME-Bus Zugriff beschrieben und gelesen werden. Das gesamte egisterbild läßt sich in vier funktional unterschiedliche Abschnitte aufteilen:

- Register, die für die Ereignismarkierung mittels Headerinformationen benötigt werden (s.Abb.C.1)
- 4 Register, mit denen man die Delay-Lines beschreiben kann (s.Abb.C.2)
- 4 Register, mit denen der Betrieb der Karte gesteuert wird, z.B. der Betriebsmodus des Front-End Driver(s. Abb.C.3)
- 4 Register, die notwendige Informationen für den Betrieb der Karte in den Auto-Trigger Modi (e.g. ROMODE_ADCTRIGGER oder ROMODE_TESTTRIGGER) beinhalten. Zusätzlich sind noch jeweils ein Register für Fehlermonitoring und die Möglichkeit eines Softwarereset vorgesehen (s.Abb.C.4).

Register Name	*Adresse*	*Kurzbeschreibung*
FEDIDLO	0	Untere Hälfte 7..0 der FED Identifikation
FEDIDHI	1	Obere Hälfte 15..8 der FED Identifikation
RECORDIDLO	2	Untere Hälfte 7..0 der Record ID
RECORDIDHI	3	Obere Hälfte 15..8 der Record ID
COMPONENTIDLO	4	Untere Hälfte 7..0 der Component ID
COMPONENTIDHI	5	Obere Hälfte 15..8 der Component ID
BYTECOUNTLO	6	Untere Hälfte 7..0 der Anzahl der Bytes
BYTECOUNTHI	7	Obere Hälfte 15..8 der Anzahl der Bytes

Abbildung C.1: Die Headerregister

Register Name	*Adresse*	*Kurzbeschreibung*
WRITEDLY1	8	Zugriff zum Auslösen eines Strobe für die Delay-Line Initialisierung
WRITEDLY2	9	"
WRITEDLY3	10	"
WRITEDLY4	11	"

Abbildung C.2: Die Delay-line Zugriffe.

Register Name	*Adresse*	*Kurzbeschreibung*
ROPIPEDEPTH	12	Gibt die Tiefe der Level-1 Pipeline an
ROMODE	13	Dieses ist das Register für den Modus
RONUMOFCHANSLO	14	Die unteren 8 Bit der Kanalanzahl
RONUMOFCHANSHI	15	Die 2 höchsten Bits für die Kanalanzahl
ROTESTDATALO	16	Die unteren 8 Bit des Test musters
ROTESTDATAHI	17	Die obersten 2 Bit für das Testmuster

Abbildung C.3: Die Betriebsregister, die im Controller implementiert sind.

Register Name	*Adresse*	*Kurzbeschreibung*
ERRORS	18	3-Bit Fehler Register
AUTOTRIGGERFLTLO	19	Untere Hälfte der FLT Nummer 7..0 für die Stand-Alone Modi
AUTOTRIGGERFLTHI	20	Obere Hälfte der FLT Nummer 15..8 für die Stand-Alone Modi
AUTOTRIGGERBXFLT	21	7 Bit BX-FLT Nummer
AUTOTRIGGERSTART	22	Der Start Strobe für die Auto-Trigger modi
SOFTWARERESET	23	Reset über den VME-Bus-Zugriff

Abbildung C.4: Die Autotrigger Register.

Anhang D

Der Controller FPGA

D.1 Das Topmodul des Controllers

```
// Top module of the DL526 controller chip.
//
// - Handles access to the registers via the VMEBus interface
// - Instantiates the FCS interface and the SHARC link interface
//

// Comment the following line out if you do not like the $display statement
// within these modules.
`define DEBUG
`timescale 1 ns / 10 ps

module DL526Ctrl(BxClk, ReadClk, notReset, notCS,        // General system functions
                notFLTAccept, BxFLT, FLTNum, CurrBx,     // FCS interface
                Address, Data, notWr, notRd,             // VME Interface
                LnkData, notLnkTrippleStart, LnkAck,     // SHARC link interface
                notADCOE, notDataOE,
                WrDly,                                   // Write strobes for progr. delay lines
                Trigger,                                 // Trigger output to readout chip
                Gate,                                    // Gate from readout chip
                Error,                                   // Error condition present on chip.
                GlobalReset                              // Output of global reset (active high)
                );


//
// Width of various registers within the controller
//
`define BXFLTWIDTH            7
`define FLTNUMWIDTH           16
`define CURRBXWIDTH           8
`define ADDRESSWIDTH          6
`define DATAWIDTH             8
`define LNKDATAWIDTH          10
`define WRDLYWIDTH            4
`define FCSFEDIDWIDTH         16
```

```
`define FCSRECORDIDWIDTH        16
`define FCSCOMPONENTIDWIDTH     16
`define FCSBYTECOUNTWIDTH       16
`define ROPIPEDEPTHWIDTH        7
`define ROMODEWIDTH             2
`define RONUMOFCHANSWIDTH       10
`define ROTESTDATAWIDTH         10

`define EVWRITEPTRWIDTH         2
`define EVREADTRIGPTRWIDTH      `EVWRITEPTRWIDTH
`define EVREADDATAPTRWIDTH      `EVWRITEPTRWIDTH

`define HANDLEFCSERRORSWIDTH    1
`define SHARCIFERRORSWIDTH      3
`define ERRORSWIDTH             (`HANDLEFCSERRORSWIDTH+`SHARCIFERRORSWIDTH)

//
// Possible errors that are detected within the chip
// (Numbers are the bit position within the <errors> register)
//
`define EVSTOREOVERFLOW         0

input                           BxClk;          // Bunch crossing clock
input                           ReadClk;        // Readout clock (same as FADC clock, besides
                                                // phase shift)
input                           notReset;       // Asynchronous system reset
input                           notCS;          // Chip select signal
input                           notFLTAccept;   // Indicates arrival of first level trigger
input [`BXFLTWIDTH-1:0]         BxFLT;          // Bunch crossing number for which flt was
                                                // generated,
                                                // valid at !notFLTAccept
input [`FLTNUMWIDTH-1:0]        FLTNum;         // First level trigger number, valid at
                                                // !notFLTAccept
input [`CURRBXWIDTH-1:0]        CurrBx;         // Current BX

input [`ADDRESSWIDTH-1:0]       Address;        // Address lines form VMEbus interface
inout [`DATAWIDTH-1:0]          Data;           // Data bus from VMEbus interface
input                           notWr;          // Falling edge indicates that data must be
                                                // latched into the chip
input                           notRd;          // Data must be asserted to <Data> during low
                                                // phase of this signal.

output [`LNKDATAWIDTH-1:0]      LnkData;        // Header/Trailer/Testdata output to SHARC
                                                // interface chips
output                          notLnkTrippleStart;// Indicates beginning of a new link nibble
                                                   // tripple.
input                           LnkAck;         // Link acknowledge signal. Transmission is
                                                // permitted for 12 nibbles if <LnkAck> is high.
output                          notADCOE;       // If low, either ADC data must be sent via
                                                // the SHARClinks or 10bit test data from
                                                //<LnkData>
output                          notDataOE;      // If low either 8bit Header/Trailer data or
                                                // 10bit ADC testdata must be sent via <LnkData>
output [`WRDLYWIDTH-1:0]        WrDly;          // Strobes for programming the external delay
```

```
                                                // lines, when corresponding address is asserted
                                                // from the VMEbus interface.
output                          Trigger;        // Trigger output to the readout chip(s)
output                          GlobalReset;

input                           Gate;           // Gate from readout chip. If high, data is
                                                // available at input of FADCs.
output                          Error;          // Error condition present on chip
reg                             Error;
reg [`WRDLYWIDTH-1:0]           WrDly;
reg [`WRDLYWIDTH-1:0]           wrDelay;        // Ungated write strobes for external
                                                // delay lines
reg [`DATAWIDTH-1:0]            dataReg;        // Register used to make assignements to the
                                                // <Data> bus.
wire[`DATAWIDTH-1:0]            Data = dataReg;
wire[`BXFLTWIDTH-1:0]           txBxFLT;        // Wires the BxFLT to transmit via SHARC link
wire[`FLTNUMWIDTH-1:0]          txFLTNum;       // Wires the FLT number to transmit via
                                                // SHARC link
wire                            txAutoTrigger;  // Wires wether the event to read out is an auto
                                                // trigger event.
wire                            linkIdle;       // Wires state of SHARC links.

//
// Register set of the DL526 controller chip. Accessible via VMEBus interface
//

// Addresses of the registers as seen by the VMEbus interface
`define FCSFEDIDLO              0
`define FCSFEDIDHI              1
`define FCSRECORDIDLO           2
`define FCSRECORDIDHI           3
`define FCSCOMPONENTIDLO        4
`define FCSCOMPONENTIDHI        5
`define FCSBYTECOUNTLO          6
`define FCSBYTECOUNTHI          7

`define WRITEDLY0               8       // Addresses of programmable delay lines
`define WRITEDLY1               9
`define WRITEDLY2               10
`define WRITEDLY3               11

`define ROPIPEDEPTH             12
`define ROMODE                  13
`define RONUMOFCHANSLO          14
`define RONUMOFCHANSHI          15
`define ROTESTDATALO            16
`define ROTESTDATAHI            17
`define ERRORS                  18
`define AUTOTRIGGERFLTNUMLO     19
`define AUTOTRIGGERFLTNUMHI     20
`define AUTOTRIGGERBXFLT        21
`define SOFTWARERESET           22
`define AUTOTRIGGERSTART        23
```

```
// Data necessary to assemble a FCS message
reg [`FCSFEDIDWIDTH-1:0]        fcsFedID;       // Front end driver id
reg [`FCSRECORDIDWIDTH-1:0]     fcsRecordID;    // Record number of message
reg [`FCSCOMPONENTIDWIDTH-1:0]  fcsComponentID; // Component id
reg [`FCSBYTECOUNTWIDTH-1:0]    fcsByteCount;   // Byte count of message to Second Level
                                                // trigger Buffer

// Parameters of the readout system
reg [`ROPIPEDEPTHWIDTH-1:0]     roPipeDepth;    // Pipeline depth (0..127)
reg [`ROMODEWIDTH-1:0]          roMode;         // Readout mode as defined by RO_MODExxxxx
reg [`RONUMOFCHANSWIDTH-1:0]    roNumOfChans;   // Number of channels of the readout chip
                                                //(system) -1 (1..1024)
reg [`ROTESTDATAWIDTH-1:0]      roTestData;     // Test data pattern register.

reg [`ERRORSWIDTH-1:0]          errors;         // The error accumulator which is only cleared
                                                // at system reset and by a read from VMEbus
wire                            handleFCSErrors;// Error lines of handleFCS submodule
wire [`SHARCIFERRORSWIDTH-1:0]  sharcIFErrors;
reg                             resetHandleFCSErrors;// Reset line for the internal error
                                                     // registers
reg                             resetSharcIFErrors;

reg                             autoTrigger;    // Becomes one for one BxClk period, when
                                                // autoTriggerStart is one.
reg                             autoTriggerStart;// Set to one by VMEbus write on
                                                 // <autoTriggerBxFLT>
reg [`FLTNUMWIDTH-1:0]          autoTriggerFLTNum;
reg [`BXFLTWIDTH-1:0]           autoTriggerBxFLT;
wire [`FLTNUMWIDTH-1:0]         muxFLTNum;
wire [`BXFLTWIDTH-1:0]          muxBxFLT;
reg                             notSoftReset;
wire                            intReset;       // Internal reset
wire                            notdoWr;
`       //
        // Submodule instances
        //
        HandleFCS handleFCS (BxClk, intReset,                       // General system functions
                        notFLTAccept, muxBxFLT, muxFLTNum, CurrBx,  // FCS interface
                        roMode, autoTrigger,
                        linkIdle, startTransmission, txBxFLT, txFLTNum, txAutoTrigger,
                        roPipeDepth, Trigger, handleFCSErrors, resetHandleFCSErrors);


        defparam
                handleFCS.BXFLTWIDTH            = `BXFLTWIDTH,
                handleFCS.FLTNUMWIDTH           = `FLTNUMWIDTH,
                handleFCS.ROMODEWIDTH           = `ROMODEWIDTH,
                handleFCS.CURRBXWIDTH           = `CURRBXWIDTH,
                handleFCS.ROPIPEDEPTHWIDTH      = `ROPIPEDEPTHWIDTH,
                handleFCS.ERRORSWIDTH           = `ERRORSWIDTH,
                handleFCS.EVWRITEPTRWIDTH       = `EVWRITEPTRWIDTH,
                handleFCS.EVREADTRIGPTRWIDTH    = `EVREADTRIGPTRWIDTH,
                handleFCS.EVREADDATAPTRWIDTH    = `EVREADDATAPTRWIDTH,
                handleFCS.EVSTOREOVERFLOW       = `EVSTOREOVERFLOW;
```

```
        SHARCLinkIF sharcLinkIF (ReadClk, intReset,
                        linkIdle, startTransmission,
                        roMode, roNumOfChans,
                        txBxFLT, txFLTNum, txAutoTrigger,
                        fcsFedID, fcsComponentID, fcsRecordID, fcsByteCount,
                        roTestData,
                        LnkData,notLnkTrippleStart,LnkAck, notADCOE,notDataOE,
                        Gate,
                        sharcIFErrors, resetSharcIFErrors);

        defparam
                sharcLinkIF.BXFLTWIDTH          = `BXFLTWIDTH,
                sharcLinkIF.FLTNUMWIDTH         = `FLTNUMWIDTH,
                sharcLinkIF.ROMODEWIDTH         = `ROMODEWIDTH,
                sharcLinkIF.RONUMOFCHANSWIDTH   = `RONUMOFCHANSWIDTH,
                sharcLinkIF.FCSFEDIDWIDTH       = `FCSFEDIDWIDTH,
                sharcLinkIF.FCSCOMPONENTIDWIDTH = `FCSCOMPONENTIDWIDTH,
                sharcLinkIF.FCSRECORDIDWIDTH    = `FCSRECORDIDWIDTH,
                sharcLinkIF.FCSBYTECOUNTWIDTH   = `FCSBYTECOUNTWIDTH,
                sharcLinkIF.ROTESTDATAWIDTH     = `ROTESTDATAWIDTH,
                sharcLinkIF.LNKDATAWIDTH        = `LNKDATAWIDTH,
                sharcLinkIF.ERRORSWIDTH         = `SHARCIFERRORSWIDTH;

`ifdef DEBUG
        always @(notReset)
                $display("**** notReset = ", notReset);
`endif

        always @ (notReset or notCS or notWr or Address)
        begin
                if (!notReset || (!notCS && !notWr && (Address == `SOFTWARERESET)))
                        assign notSoftReset = 0;
                else
                        assign notSoftReset = 1;
`ifdef DEBUG
                $display("**** notSoftReset = ", notSoftReset);
`endif
        end
        assign intReset = !(notSoftReset && notReset);
        assign GlobalReset = intReset;

`ifdef DEBUG
        always @ (intReset)
                if (intReset)
                        $display("**** intReset = 1");
                else
                        $display("**** intReset = 0");
`endif

        //
        // Handle asynchronous system reset.
        //
        always @ (intReset)
```

```
begin
        if (intReset)
        begin
                assign autoTrigger = 0;
                assign autoTriggerStart = 0;
        end
        else
        begin
                deassign autoTrigger;
                deassign autoTriggerStart;
        end
end

//
// Handle VMEBus interface accesses to internal registers
//
assign notdoWr = notCS || notWr;
always @ (posedge (notdoWr || !(Address == `FCSFEDIDLO)))
        fcsFedID[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSFEDIDHI)))
        fcsFedID[15:8] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSRECORDIDLO)))
        fcsRecordID[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSRECORDIDHI)))
        fcsRecordID[15:8] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSCOMPONENTIDLO)))
        fcsComponentID[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSCOMPONENTIDHI)))
        fcsComponentID[15:8] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSBYTECOUNTLO)))
        fcsByteCount[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `FCSBYTECOUNTHI)))
        fcsByteCount[15:8] <= Data;

always @ (posedge (notdoWr || !(Address == `ROPIPEDEPTH)))
        roPipeDepth <= Data[6:0];
always @ (posedge (notdoWr || !(Address == `ROMODE)))
        roMode <= Data[1:0];
always @ (posedge (notdoWr || !(Address == `RONUMOFCHANSLO)))
        roNumOfChans[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `RONUMOFCHANSHI)))
        roNumOfChans[9:8] <= Data;
always @ (posedge (notdoWr || !(Address == `ROTESTDATALO)))
        roTestData[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `ROTESTDATAHI)))
        roTestData[9:8] <= Data;

always @ (posedge (notdoWr || !(Address == `AUTOTRIGGERFLTNUMLO)))
        autoTriggerFLTNum[7:0] <= Data;
always @ (posedge (notdoWr || !(Address == `AUTOTRIGGERFLTNUMHI)))
        autoTriggerFLTNum[15:8] <= Data;
always @ (posedge (notdoWr || !(Address == `AUTOTRIGGERBXFLT)))
        autoTriggerBxFLT[6:0] <= Data;
always @ (posedge (notdoWr || !(Address == `AUTOTRIGGERSTART)))
```

```
        autoTriggerStart <= Data;

//
// Generate the "reset error register" signals for the submodules.
//
always @ (notWr or Address or intReset)
        if ((!notWr && (Address == `ERRORS)) || intReset)
        begin
                assign errors = 0;
                resetHandleFCSErrors <= 1;
                resetSharcIFErrors <= 1;
        end
        else
        begin
                deassign errors;
                resetHandleFCSErrors <= 0;
                resetSharcIFErrors <= 0;
        end

always @ (notCS or notRd or Address or roPipeDepth or roMode or
          fcsFedID or fcsRecordID or fcsComponentID or
          fcsByteCount or roNumOfChans or roTestData or errors)
        vmebusRead(Address);

//
// Generate the write strobes for the externally programmable delay lines registers.
//
always @(Address or notCS or notWr)
begin
        case (Address)
                `WRITEDLY0      :       wrDelay = 4'b0001;
                `WRITEDLY1      :       wrDelay = 4'b0010;
                `WRITEDLY2      :       wrDelay = 4'b0100;
                `WRITEDLY3      :       wrDelay = 4'b1000;
                default         :       wrDelay = 0;
        endcase
end
always @ (notWr or wrDelay or notCS)
begin
        WrDly = {!notWr && !notCS, !notWr && !notCS, !notWr && !notCS, !notWr
                 && !notCS} & wrDelay;
end

//
// Store all error conditions from the HandleFCS module
//
always @ (posedge BxClk)
begin: storeHandleFCSErrors
        errors[0] = errors[0] | handleFCSErrors;
end

//
// Store all error conditions of the SHARCLinkIf module
//
```

```
        always @ (posedge ReadClk)
        begin: storeSharcIFErrors
                errors[`ERRORSWIDTH-1:1] = errors[`ERRORSWIDTH-1:1] | sharcIFErrors;
        end

        //
        // Signal wether an error condition is present on the chip to the outer world.
        //
        always @ (errors)
        begin
                Error = |errors;
`ifdef DEBUG
                $display("**** Error condition changed. <errors> = %b", errors);
`endif
        end

        //
        // Look if there is an auto trigger pending. If yes, submit it synchronous to the
        // bunch crossing clock.
        //
        always @ (posedge BxClk)
        begin: startAutoTriggerBlk
                if (intReset)
                        disable startAutoTriggerBlk;
                else
                begin
                        if (autoTriggerStart)
                        begin
                                @ (posedge BxClk)
                                        autoTrigger = 1;     // Make sure the trigger is only
                                                             // asserted for one bunch crossing
                                                             // interval
                                while (autoTriggerStart)
                                        @ (posedge BxClk)
                                                autoTrigger = 0;
                        end
                end
        end

`ifdef DEBUG
        always @ (autoTrigger)
                $display("**** autoTrigger = %b", autoTrigger);
`endif

        //
        // Multiplex the first level trigger number and the buch crossing of the first level
        // trigger number depending on wether there is an auto trigger event or a normal one.
        //
        assign muxFLTNum = autoTrigger ? autoTriggerFLTNum : FLTNum;
        assign muxBxFLT = autoTrigger ? autoTriggerBxFLT : BxFLT;

        //
        // Satisfy read request from VMEbus interface
        //
```

```
        task vmebusRead;
                input [5:0] address;

                begin
                        if (!notRd && !notCS)
                                case (address)
                                    `FCSFEDIDLO         :       dataReg <= fcsFedID[7:0];
                                    `FCSFEDIDHI         :       dataReg <= fcsFedID[15:8];
                                    `FCSRECORDIDLO      :       dataReg <= fcsRecordID[7:0];
                                    `FCSRECORDIDHI      :       dataReg <= fcsRecordID[15:8];
                                    `FCSCOMPONENTIDLO   :       dataReg <= fcsComponentID[7:0];
                                    `FCSCOMPONENTIDHI   :       dataReg <= fcsComponentID[15:8];
                                    `FCSBYTECOUNTLO     :       dataReg <= fcsByteCount[7:0];
                                    `FCSBYTECOUNTHI     :       dataReg <= fcsByteCount[15:8];
                                    `ROPIPEDEPTH        :       dataReg <= roPipeDepth;
                                    `ROMODE             :       dataReg <= roMode;
                                    `RONUMOFCHANSLO     :       dataReg <= roNumOfChans[7:0];
                                    `RONUMOFCHANSHI     :       dataReg <= roNumOfChans[9:8];
                                    `ROTESTDATALO       :       dataReg <= roTestData[7:0];
                                    `ROTESTDATAHI       :       dataReg <= roTestData[9:8];
                                    `ERRORS             :       dataReg <= errors;
                                    default             :       dataReg <= 8'bzzzzzzzz;
                                endcase
                        else
                                dataReg <= 8'bzzzzzzzz;
                end
        endtask
endmodule
```

D.2 Das FCSHandle Modul

```
//*********************************************************************************
// This module handles the Fast Control System signals.
// Upon reception of a first level trigger it stores all valid information
// into a fifo and marks that the fifo is not empty any longer.
//
module HandleFCS(BxClk, Reset,                          // General system functions
                 notFLTAccept, BxFLT, FLTNum, CurrBx,  // FCS interface
                 RoMode, AutoTrigger,
                 LinkIdle, StartTransmission, TxBxFLT,
                 TxFLTNum, TxAutoTrigger,              // SHARC link interface
                 RoPipeDepth, Trigger, Errors, ResetErrors
    );

parameter
    BXFLTWIDTH    = 7,
     FLTNUMWIDTH    = 16,
    CURRBXWIDTH     = 8,
    ROMODEWIDTH    = 2,
    ROPIPEDEPTHWIDTH   = 7,
    ERRORSWIDTH     = 1,
    EVWRITEPTRWIDTH   = 2,
    EVREADTRIGPTRWIDTH   = 2,
```

```
    EVREADDATAPTRWIDTH   = 2,
    EVSTOREOVERFLOW   = 0;

`define EVSTOREWIDTH     (BXFLTWIDTH+FLTNUMWIDTH+3)
`define EVTRIGGERPENDING  (BXFLTWIDTH+FLTNUMWIDTH)
`define EVDATAPENDING    (BXFLTWIDTH+FLTNUMWIDTH+1)
`define EVAUTOTRIGGER    (BXFLTWIDTH+FLTNUMWIDTH+2)


input                    BxClk;    // Bunch crossing clock
input                    Reset;    // Asynchronous system reset

input                    notFLTAccept;  // Indicates arrival of first level trigger
input [BXFLTWIDTH-1:0]   BxFLT;    // Bunch crossing number for which flt was generated,
                                   // valid at !notFLTAccept
input [FLTNUMWIDTH-1:0]  FLTNum;   // First level trigger number, valid at !notFLTAccept
input [CURRBXWIDTH-1:0]  CurrBx;   // Current BX
input [ROMODEWIDTH-1:0]  RoMode;   // Readout mode.
input                    AutoTrigger;  // Positive edge triggers a readout sequence with
                                   // current <BxFLT> and <FLTNum>.
input                    LinkIdle; // 1 indicates that SHARC link is ready for transmission
output                   StartTransmission;// Initiates transmission of next event
output [FLTNUMWIDTH-1:0] TxFLTNum; // FLT number of event to transmit
output [BXFLTWIDTH-1:0]  TxBxFLT;  // BX-FLT number of event to transmit
output                   TxAutoTrigger;  // Wether the event to transmit is an auto
                                         // triggered or not.
input [ROPIPEDEPTHWIDTH-1:0] RoPipeDepth; // Lenght of the First Level Pipeline
output                   Trigger;  // Trigger output to the readout chip.
output                   Errors;   // Error output of this module. An error is only
                                   // asserted at occurence for one clock cycle and must
                                   // be saved externally.
reg                      StartTransmission;// Initiates transmission of next event
reg [FLTNUMWIDTH-1:0]    TxFLTNum;  // FLT number of event to transmit
reg [BXFLTWIDTH-1:0]     TxBxFLT;  // BX-FLT number of event to transmit
reg                      TxAutoTrigger;  // Wether the event to transmit is an auto
                                         // triggered or not.

reg                      Trigger;
reg                      Errors;
input                    ResetErrors;

//
// Internal storage of the controller
//
reg [`EVSTOREWIDTH-1:0]  evStore0,  // Event storage.
                         evStore1,  // Bits [BXFLTWIDTH-1:0] store BxFLT of event
                         evStore2,  // Bits [BXFLTWIDTH+FLTNUMWIDTH-1:BXFLTWIDTH] store FLTNum
                         evStore3,  // Bit  [`EVSTOREWIDTH-3] stores wether a trigger has
                                    // been sent to the chip for this event
                                    // Bit  [`EVSTOREWIDTH-2] stores wether the complete data
                                    // has already been sent to the SHARC for this event.
                                    // Bit [`EVSTOREWIDTH-1] stores wether the event was
                                    // generated as an auto trigger event.
                         evStore2Trig;  // event storage to trigger (without flags)
```

```
reg [EVWRITEPTRWIDTH-1:0] evWritePtr;  // Points to the evStoreX where the next event
                                       // will be stored.
reg [EVREADTRIGPTRWIDTH-1:0] evReadTrigPtr;// Points to the evStoreX for which the next
                                           // trigger must
                                       // be sent to the readout chip.
reg [EVREADDATAPTRWIDTH-1:0] evReadDataPtr;// Points to the evStoreX for which data
                                           // must be sent to the SHARC link
wire                     evStoreFull;  // 1 if there is no space left in the event storage
wire                     eventTriggerPending;
wire                     storeEvent;
wire                     triggerThere;
wire                     triggerError;
wire                     sendData;
wire                     clearStore0TrigFlag,
                         clearStore0DataFlag,
                         clearStore1TrigFlag,
                         clearStore1DataFlag,
                         clearStore2TrigFlag,
                         clearStore2DataFlag,
                         clearStore3TrigFlag,
                         clearStore3DataFlag;
`ifdef DEBUG
reg [`EVSTOREWIDTH-1:0]  dbgEvStore;  // Event store for debugging
`endif

  //
  // Generate the global system reset.
  //
  always @ (Reset)
  begin
`ifdef DEBUG
    $display("**** HandleFCS.Reset = ", Reset);
`endif

    if (Reset)
    begin
      assign evStore0 = 0;
      assign evStore1 = 0;
      assign evStore2 = 0;
      assign evStore3 = 0;
      assign evStore2Trig = 0;
      assign evWritePtr = 0;
      assign evReadTrigPtr = 0;
      assign evReadDataPtr = 0;
      assign StartTransmission = 0;
      assign TxAutoTrigger = 0;
      assign Trigger = 0;
    end
    else
    begin
      deassign evStore0;
      deassign evStore1;
      deassign evStore2;
```

```
      deassign evStore3;
      deassign evStore2Trig;
      deassign evWritePtr;
      deassign evReadTrigPtr;
      deassign evReadDataPtr;
      deassign StartTransmission;
      deassign TxAutoTrigger;
      deassign Trigger;
    end
  end

  //
  // The <Errors> register may be reset by the global reset and a separate one.
  //
  always @ (Reset or ResetErrors)
    if (Reset || ResetErrors)
      assign Errors = 0;
    else
      deassign Errors;

  assign eventTriggerPending = ((evStore2Trig[`EVTRIGGERPENDING] &&
        (evStore2Trig[BXFLTWIDTH-1:0] + RoPipeDepth == CurrBx))
         ||  evStore2Trig[`EVAUTOTRIGGER]);
  //
  // Calculate wether there is space left in the event storage.
  // Since the write pointer writes the four storages consecutively and the read pointer
  // does so, too, it is sufficient to look wether all slots are occupied or not.
  //
  assign evStoreFull =  (evStore0[`EVTRIGGERPENDING] || evStore0[`EVDATAPENDING]) &&
        (evStore1[`EVTRIGGERPENDING] || evStore1[`EVDATAPENDING]) &&
        (evStore2[`EVTRIGGERPENDING] || evStore2[`EVDATAPENDING]) &&
        (evStore3[`EVTRIGGERPENDING] || evStore3[`EVDATAPENDING]);

`ifdef DEBUG
  always @ (evStoreFull)
    $display("**** evStoreFull = ", evStoreFull);
`endif

  assign triggerThere = ((RoMode == 3 || RoMode == 2) ? AutoTrigger : !notFLTAccept);
  assign storeEvent = triggerThere && !evStoreFull;
  assign triggerError = triggerThere && evStoreFull;

  assign sendData = (!StartTransmission && LinkIdle) &&
      (evStore0[`EVDATAPENDING] ||
      evStore1[`EVDATAPENDING] ||
      evStore2[`EVDATAPENDING] ||
      evStore3[`EVDATAPENDING]);

  always @ (posedge BxClk)
  begin: getNextEventBlk
    if (Reset)
      disable getNextEventBlk;
    else
    begin
```

```
      //
      // When a first level trigger occurs, store the event information into
      // the fifo and increase the write pointer
      // If no space is left, mark error.
      //
      if (storeEvent)
      begin
`ifdef DEBUG
  if (AutoTrigger)
  $display("**** HandleFCS.storeEvent(%d, %d) AutoTrigger at location %d", FLTNum, BxFLT, evWrite
  else
  $display("**** HandleFCS.storeEvent(%d, %d) at location %d", FLTNum, BxFLT, evWritePtr);
`endif
        case (evWritePtr)
          0 : evStore0[BXFLTWIDTH+FLTNUMWIDTH-1:0] <= {FLTNum, BxFLT};
          1 : evStore1[BXFLTWIDTH+FLTNUMWIDTH-1:0] <= {FLTNum, BxFLT};
          2 : evStore2[BXFLTWIDTH+FLTNUMWIDTH-1:0] <= {FLTNum, BxFLT};
          3 : evStore3[BXFLTWIDTH+FLTNUMWIDTH-1:0] <= {FLTNum, BxFLT};
        endcase
        evWritePtr <= evWritePtr + 1;
      end
      else if (triggerError)
        Errors <= 1;
    end
  end

`ifdef DEBUG
  always @ (evStore0 or evStore1 or evStore2 or evStore3)
    $display("**** Event storage = %x %x %x %x", evStore0, evStore1, evStore2, evStore3);
`endif

  assign clearStore0TrigFlag = (evReadTrigPtr == 0) && eventTriggerPending;
  assign clearStore0DataFlag = (evReadDataPtr == 0) && sendData;
  always @ (posedge BxClk)
  begin: setStore0FlagsBlk
    if (Reset)
      disable setStore0FlagsBlk;
    else
    begin
      if (clearStore0TrigFlag || clearStore0DataFlag)
      begin
        if (clearStore0TrigFlag)
          evStore0[`EVTRIGGERPENDING] <= 0;
        if (clearStore0DataFlag)
          evStore0[`EVAUTOTRIGGER:`EVDATAPENDING] <= 2'b0;
      end
      else if ((evWritePtr == 0) && storeEvent)
        evStore0[`EVAUTOTRIGGER:`EVTRIGGERPENDING] <= {AutoTrigger,2'b11};
    end
  end

  assign clearStore1TrigFlag = (evReadTrigPtr == 1) && eventTriggerPending;
  assign clearStore1DataFlag = (evReadDataPtr == 1) && sendData;
  always @ (posedge BxClk)
```

```
begin: setStore1FlagsBlk
  if (Reset)
    disable setStore1FlagsBlk;
  else
  begin
    if (clearStore1TrigFlag || clearStore1DataFlag)
    begin
      if (clearStore1TrigFlag)
        evStore1[`EVTRIGGERPENDING] <= 0;
      if (clearStore1DataFlag)
        evStore1[`EVAUTOTRIGGER:`EVDATAPENDING] <= 2'b0;
    end
    else if ((evWritePtr == 1) && storeEvent)
      evStore1[`EVAUTOTRIGGER:`EVTRIGGERPENDING] <= {AutoTrigger,2'b11};
  end
end

assign clearStore2TrigFlag = (evReadTrigPtr == 2) && eventTriggerPending;
assign clearStore2DataFlag = (evReadDataPtr == 2) && sendData;
always @ (posedge BxClk)
begin: setStore2FlagsBlk
  if (Reset)
    disable setStore2FlagsBlk;
  else
  begin
    if (clearStore2TrigFlag || clearStore2DataFlag)
    begin
      if (clearStore2TrigFlag)
        evStore2[`EVTRIGGERPENDING] <= 0;
      if (clearStore2DataFlag)
        evStore2[`EVAUTOTRIGGER:`EVDATAPENDING] <= 2'b0;
    end
    else if ((evWritePtr == 2) && storeEvent)
      evStore2[`EVAUTOTRIGGER:`EVTRIGGERPENDING] <= {AutoTrigger,2'b11};
  end
end

assign clearStore3TrigFlag = (evReadTrigPtr == 3) && eventTriggerPending;
assign clearStore3DataFlag = (evReadDataPtr == 3) && sendData;
always @ (posedge BxClk)
begin: setStore3FlagsBlk
  if (Reset)
    disable setStore3FlagsBlk;
  else
  begin
    if (clearStore3TrigFlag || clearStore3DataFlag)
    begin
      if (clearStore3TrigFlag)
        evStore3[`EVTRIGGERPENDING] <= 0;
      if (clearStore3DataFlag)
        evStore3[`EVAUTOTRIGGER:`EVDATAPENDING] <= 2'b0;
    end
    else if ((evWritePtr == 3) && storeEvent)
      evStore3[`EVAUTOTRIGGER:`EVTRIGGERPENDING] <= {AutoTrigger,2'b11};
```

```
    end
  end

  always @ (posedge BxClk)
  begin: sendDataBlk
    if (Reset)
      disable sendDataBlk;
    else
    begin
      //
      // If <EvReadDataPtr> points to a storage location which has still data pending,
      // initiate data transmission via the SHARC links
      //
      if (sendData)
      begin
        case (evReadDataPtr)
          0 : {TxAutoTrigger,TxFLTNum,TxBxFLT} <= {evStore0[`EVAUTOTRIGGER],
                                                  evStore0[BXFLTWIDTH+FLTNUMWIDTH-1:0]};
          1 : {TxAutoTrigger,TxFLTNum,TxBxFLT} <= {evStore1[`EVAUTOTRIGGER],
                                                  evStore1[BXFLTWIDTH+FLTNUMWIDTH-1:0]};
          2 : {TxAutoTrigger,TxFLTNum,TxBxFLT} <= {evStore2[`EVAUTOTRIGGER],
                                                  evStore2[BXFLTWIDTH+FLTNUMWIDTH-1:0]};
          3 : {TxAutoTrigger,TxFLTNum,TxBxFLT} <= {evStore3[`EVAUTOTRIGGER],
                                                  evStore3[BXFLTWIDTH+FLTNUMWIDTH-1:0]};
        endcase
        StartTransmission <= 1;
`ifdef DEBUG
  $display("**** HandleFCS: start transmission");
`endif
        evReadDataPtr <= evReadDataPtr + 1;
      end
      else
        StartTransmission <= 0;
    end
  end

  always @ (posedge BxClk)
  begin: sendTriggerBlk
    if (Reset)
      disable sendTriggerBlk;
    else
    begin
      if (eventTriggerPending)
      begin
        Trigger <= 1;
        evReadTrigPtr <= evReadTrigPtr + 1;
      end
      else
        Trigger <= 0;
    end
  end

  //
  // Select the event the read pointer is pointing to.
```

```
  //
  always @ (posedge BxClk)
  begin: selectEventBlk
    if (Reset)
      disable selectEventBlk;
    else
    begin
      case (evReadTrigPtr)
        0 : evStore2Trig <= evStore0;
        1 : evStore2Trig <= evStore1;
        2 : evStore2Trig <= evStore2;
        3 : evStore2Trig <= evStore3;
      endcase
    end
  end

endmodule
```

D.3 Das Sharc Link Interface Modul

```
//*****************************************************************************************
// This module handles transportation of data via the SHARC links.
//

module SHARCLinkIF (Clk, Reset,                      // System interface
                    LinkIdle, StartTransmission,     // Handshake signals
                    RoMode, RoNumOfChans,            // Data parameters
                    TxBxFLT, TxFLTNum, TxAutoTrigger,// FLT event parameters
                    FCSFedID, FCSComponentID,
                    FCSRecordID, FCSByteCount,       // FCS parameters
                    RoTestData,                      // Test data pattern
                    LnkData, notLnkTrippleStart,wack,
                    notADCOE,notDataOE,
                    Gate,                            // Gate from readout chip.
                    Errors, ResetErrors);

parameter
    BXFLTWIDTH    = 7,
    FLTNUMWIDTH    = 16,
    ROMODEWIDTH    = 2,
    RONUMOFCHANSWIDTH  = 10,
    FCSFEDIDWIDTH    = 16,
    FCSCOMPONENTIDWIDTH  = 16,
    FCSRECORDIDWIDTH  = 16,
    FCSBYTECOUNTWIDTH  = 16,
    ROTESTDATAWIDTH    = 10,
    LNKDATAWIDTH    = 10,
    ERRORSWIDTH    = 3;

// Content of <roMode> register
`define ROMODE_EXTSTROBE   0  // Readout is inited by a strobe on the external
                              // gate input of at least one Readout clock duration
```

```
                             // and performed for as many channels as are
                             // programmed in register <RoNumOfChans>
`define ROMODE_TEST        1  // Same as <`ROMODE_EXTSTROBE>, but test data is sent.
`define ROMODE_ADCTRIGGER  2  // An Autotrigger is generated when a VMEbus-accessible
                             // register for autotigger start is written to
`define ROMODE_TESTTRIGGER 3  // Same as <`ROMODE_TEST>, but an auto trigger is
                             // generated, when the VMEbus-accessible <AutoTriggerBxFLTHi>
                             // register is written

`define GATEDELAY    12       // Gate delay to take care of FADC`s internal pipelining

input                           Clk;
input                           Reset;
output                          LinkIdle;
input                           StartTransmission;
input [ROMODEWIDTH-1:0]         RoMode;
input [RONUMOFCHANSWIDTH-1:0]   RoNumOfChans;
input [BXFLTWIDTH-1:0]          TxBxFLT;
input [FLTNUMWIDTH-1:0]         TxFLTNum;
input                           TxAutoTrigger;
input [FCSFEDIDWIDTH-1:0]       FCSFedID;
input [FCSCOMPONENTIDWIDTH-1:0] FCSComponentID;
input [FCSRECORDIDWIDTH-1:0]    FCSRecordID;
input [FCSBYTECOUNTWIDTH-1:0]   FCSByteCount;
input [ROTESTDATAWIDTH-1:0]     RoTestData;
output[LNKDATAWIDTH-1:0]        LnkData;
output                          notLnkTrippleStart;
input                           wack;  // Link acknowledge signal. Transmission is
                                       // permitted for 8 nibbles if <LnkAck> is high.
output                          notADCOE;
output                          notDataOE;
input                           Gate;  // Gate from readout chip. If high, data is valid at
                                       // input form FADCs.
output[ERRORSWIDTH-1:0]         Errors;
input                           ResetErrors;  // Asynchronous reset for error register.


//
// State definitions
//
`define IDLE      0
`define TXBYTECOUNTHI    1
`define TXBYTECOUNTLO    2
`define TXFCSRECORDIDHI    3
`define TXFCSRECORDIDLO    4
`define TXFLTNUMHI    5
`define TXFLTNUMLO    6
`define TXBXFLTNUMHI    7
`define TXBXFLTNUMLO    8
`define TXFCSCOMPONENTIDHI  9
`define TXFCSCOMPONENTIDLO  10
`define TXFCSFEDIDHI    11
`define TXFCSFEDIDLO    12
`define TXWAITDATA    13
```

```
`define TXDATA      14
`define TXPADEVEN    15
`define TXPAD_4_6MINUS2    16
`define TXENDFLTNUMHI    17
`define TXENDFLTNUMLO    18
`define TXPADEVENGATEERROR  19
`define TXPADGATEERROR    20
`define TXPADEVENHANDSHAKEERROR  21
`define TXPADHANDSHAKEERROR  22


`define LINKIFSTATEWIDTH  5


`define TXNIBBLESWIDTH    4


//
// States of {notADCOE,notDataOE} during transmission
//
`define TXNOTHING    2'b11
`define TXHEADER    2'b10
`define TXADCDATA    2'b01
`define TXTESTDATA    2'b00

//
// Bit positions of possible errors.
//
`define LATEHANDSHAKEERR  0  // The SHARC has sent the handshake signal too late.
`define EARLYGATEERR      1  // Data was valid before header was
                             // completely transmitted.
`define GATELENGTHERR     2  // If <`ROMODE_EXTGATE> is used, this error is issued
                             // when the external gate length does not aggree with
                             // the content of <RoNumOfChans>.
//
// Internal registers of the module
//
reg [ERRORSWIDTH-1:0]       Errors;  // The error register
reg                         notLnkTrippleStart;// If 0, the SHARC link interface
                                               // must send data, if 1 it must not.
reg                         notADCOE;
reg                         notDataOE;
reg [LNKDATAWIDTH-1:0]      LnkData;
reg                         LinkIdle;
reg [`LINKIFSTATEWIDTH-1:0] linkIFstate;
reg [FCSBYTECOUNTWIDTH-1:0] byteCount;  // Counts the number of bytes yet to transmit.
reg [`GATEDELAY-2:0]        gateDelayLine;  // Delay line for gate to obey the FADC`s
                                            // internal pipelining.
wire                        delayedGate;    // Gate delayed by `GATEDELAY clock cycles.
reg                         delayedGatePassed;// Storage which memorizes wether a gate
                                              // already passed or not.
reg [RONUMOFCHANSWIDTH-1:0] chans2Send;  // Contains the remaining number of channels to send.
reg [ROTESTDATAWIDTH-1:0]   testData;    // register to rotate test data
reg [ERRORSWIDTH-1:0]       intErrors;   // Internal error flags used to send error
```

```
                                          // code via SHARC link.
reg [3:0]                   txNibbles;   // Number of nibbles already sent modulus 12
wire[3:0]                   txNibblesPlus2;    // = ((txNibbles + 2) modulus 12)
wire[3:0]                   txNibblesPlus3;    // = ((txNibbles + 3) modulus 12)

  //
  // Instantiate modules that calculate the next nibble values.
  // During transmission one of the sums is selected an assigned
  // to txNibbles.
  //
  Inc2Mod12  nibblesPlus2(txNibbles, txNibblesPlus2);
  defparam  nibblesPlus2.WIDTH = 4;

  Inc3Mod12  nibblesPlus3(txNibbles, txNibblesPlus3);
  defparam  nibblesPlus3.WIDTH = 4;

  `ifdef DEBUG
    always @(wack)
    $display("**** SHARCLinkIF:wack = %b", wack," **** t = %8.2fns ", $realtime);
  `endif

  //
  // Generate asynchronous system reset.
  //
  always @ (Reset)
    if (Reset)
    begin
      assign linkIFstate = `IDLE;
      assign byteCount = 0;
      assign gateDelayLine = 0;
      assign chans2Send = 0;
      assign testData = 0;
      assign notLnkTrippleStart = 1;
      assign LinkIdle = 0;
      assign notADCOE = 1;
      assign notDataOE = 1;
      assign LnkData = 0;
      assign intErrors = 0;
      assign txNibbles = 0;
    end
    else
    begin
      deassign linkIFstate;
      deassign byteCount;
      deassign gateDelayLine;
      deassign chans2Send;
      deassign testData;
      deassign notLnkTrippleStart;
      deassign LinkIdle;
      deassign notADCOE;
      deassign notDataOE;
      deassign LnkData;
      deassign intErrors;
      deassign txNibbles;
```

```
  end

//
// The <Errors> and the <intErrors> register may be reset by the global reset
// and a separate one.
//
always @ (Reset or ResetErrors)
  begin
  if (Reset || ResetErrors)
  begin
    assign Errors = 0;
    assign intErrors = 0;
  end
  else
  begin
    deassign Errors;
    deassign intErrors;
  end
end

//
// Generate delayed gate by a shift register and latch
      // if gate passes
//
assign delayedGate = gateDelayLine[`GATEDELAY-2];
always @ (posedge Clk)
begin
  gateDelayLine = {gateDelayLine[`GATEDELAY-3:0], !Gate};
  if (delayedGate)
    delayedGatePassed = 1;
end

//
// Generate reset signal for <delayedGatePassed>
//
always @ (Reset)
  if (Reset)
    assign delayedGatePassed = 0;
  else
    deassign delayedGatePassed;

//
// Next state logic
//
always @ (posedge Clk)
begin
  case (linkIFstate)

    `IDLE  :  begin
        `ifdef DEBUG
          $display("**** SHARCLinkIF is in idle state");
        `endif
        LinkIdle <= 1;
        notLnkTrippleStart <= 1;
```

```
    {notADCOE,notDataOE} <= `TXNOTHING;
    LnkData <= 0;
    Errors[`GATELENGTHERR] <= 0;
    intErrors <= 0;
    byteCount <= FCSByteCount - 2;
    if (!StartTransmission)
      linkIFstate = `IDLE;
    else
      linkIFstate = `TXBYTECOUNTHI;
  end

`TXBYTECOUNTHI  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in bytecount high state");
    `endif
    if (wack)
    begin
      chans2Send <= RoNumOfChans;
      LinkIdle <= 0;
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSByteCount[15:8];
      testData <= RoTestData;
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
    end
    linkIFstate = `TXBYTECOUNTLO;
  end

`TXBYTECOUNTLO  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in bytecount low state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSByteCount[7:0];
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXFCSRECORDIDHI;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
```

```
      {notADCOE,notDataOE} = `TXNOTHING;
      `ifdef DEBUG
        $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
      `endif
      linkIFstate = `TXBYTECOUNTLO;
    end
  end

`TXFCSRECORDIDHI  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in record id high state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSRecordID[15:8];
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXFCSRECORDIDLO;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      `ifdef DEBUG
        $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
      `endif
      linkIFstate = `TXFCSRECORDIDHI;
    end
  end

`TXFCSRECORDIDLO  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in record id low state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSRecordID[7:0];
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXFLTNUMHI;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      `ifdef DEBUG
```

```
          $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
        `endif
        linkIFstate = `TXFCSRECORDIDLO;
      end
    end

  `TXFLTNUMHI  :  begin
      if (wack)
      begin
        `ifdef DEBUG
          $display("**** SHARCLinkIF is in FLT num high state");
        `endif
        notLnkTrippleStart <= 0;
        {notADCOE,notDataOE} <= `TXHEADER;
        LnkData[9:2] <= TxFLTNum[15:8];
        byteCount <= byteCount - 1;
        txNibbles <= txNibblesPlus2;
        linkIFstate = `TXFLTNUMLO;
      end
      else
      begin
        intErrors[`LATEHANDSHAKEERR] = 1;
        Errors[`LATEHANDSHAKEERR] = 1;
        notLnkTrippleStart = 1;
        {notADCOE,notDataOE} = `TXNOTHING;
        `ifdef DEBUG
          $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
        `endif
        linkIFstate = `TXFLTNUMHI;
      end
    end

  `TXFLTNUMLO  :  begin
      if (wack)
      begin
        `ifdef DEBUG
          $display("**** SHARCLinkIF is in FLT num low state");
        `endif
        notLnkTrippleStart <= 0;
        {notADCOE,notDataOE} <= `TXHEADER;
        LnkData[9:2] <= TxFLTNum[7:0];
        byteCount <= byteCount - 1;
        txNibbles <= txNibblesPlus2;
        linkIFstate = `TXBXFLTNUMHI;
      end
      else
      begin
        intErrors[`LATEHANDSHAKEERR] = 1;
        Errors[`LATEHANDSHAKEERR] = 1;
        notLnkTrippleStart = 1;
        {notADCOE,notDataOE} = `TXNOTHING;
        `ifdef DEBUG
          $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
        `endif
```

```
      linkIFstate = `TXFLTNUMLO;
    end
  end

`TXBXFLTNUMHI  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in BxFLT high state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= 0;
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXBXFLTNUMLO;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      `ifdef DEBUG
        $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
      `endif
      linkIFstate = `TXBXFLTNUMHI;
    end
  end

`TXBXFLTNUMLO  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in BxFLT low state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= {1'b0,TxBxFLT[6:0]};
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXFCSCOMPONENTIDHI;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      `ifdef DEBUG
        $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
      `endif
      linkIFstate = `TXBXFLTNUMLO;
    end
```

```
  end

`TXFCSCOMPONENTIDHI  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in component id high state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSComponentID[15:8];
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXFCSCOMPONENTIDLO;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      `ifdef DEBUG
        $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
      `endif
      linkIFstate = `TXFCSCOMPONENTIDHI;
    end
  end

`TXFCSCOMPONENTIDLO  :  begin
    if (wack)
    begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in component id low state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSComponentID[7:0];
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      linkIFstate = `TXFCSFEDIDHI;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      notLnkTrippleStart = 1;
      `ifdef DEBUG
        $display("**** LateHandshake  **** t = %8.2fns ", $realtime);
      `endif
      linkIFstate = `TXFCSCOMPONENTIDLO;
    end
  end
```

```
`TXFCSFEDIDHI  :  begin
      `ifdef DEBUG
        $display("**** SHARCLinkIF is in FED id high state");
      `endif
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= FCSFedID[15:8];
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      if (!delayedGatePassed)
        linkIFstate = `TXFCSFEDIDLO;
      else
        linkIFstate = `TXPADEVENGATEERROR;
    end

`TXFCSFEDIDLO  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in FED id low state");
    `endif
    notLnkTrippleStart <= 0;
    {notADCOE,notDataOE} <= `TXHEADER;
    LnkData[9:2] <= FCSFedID[7:0];
    byteCount <= byteCount - 1;
  //  chans2Send <= chans2Send - 1;
    txNibbles <= txNibblesPlus2;
    if (intErrors[`LATEHANDSHAKEERR])
      linkIFstate = `TXPADEVENHANDSHAKEERROR;
    else if ((RoMode !=`ROMODE_TESTTRIGGER) && (RoMode !=`ROMODE_ADCTRIGGER)
          && !delayedGatePassed)
         linkIFstate = `TXWAITDATA;
       else
        linkIFstate = `TXDATA;
  end

`TXWAITDATA  :  begin
  // Wait to see the external gate rising when we are not in test mode.
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in wait gate state");
    `endif
    notLnkTrippleStart = 1;
    {notADCOE,notDataOE} = `TXNOTHING;
    LnkData = 0;
    `ifdef DEBUG
      $display("**** chans2Send = ", chans2Send);
    `endif
    if(!delayedGatePassed)
      linkIFstate = `TXWAITDATA;
    else
      linkIFstate = `TXDATA;
  end

`TXDATA  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in data state. chans2Send = ", chans2Send);
```

```
    `endif
    `ifdef DEBUG
      $display("**** SHARCLinkIF: byteCount = %d", byteCount);
    `endif
    delayedGatePassed = 0;
    if ((RoMode == `ROMODE_TEST) || (RoMode == `ROMODE_TESTTRIGGER))
      {notADCOE,notDataOE} = `TXTESTDATA;
    else
      {notADCOE,notDataOE} = `TXADCDATA;
    // While sending data (either from the FADCs or test data)
    // we always present the test data register content on the
    // data bus. This may be useful for debugging during ADC data
    // transmission (by expense of power).
    if (wack)
    begin
      chans2Send = chans2Send - 1;
      notLnkTrippleStart = 0;
      LnkData = testData;
      testData = {testData[ROTESTDATAWIDTH-2:0],testData[ROTESTDATAWIDTH-1]};
      if(txNibbles[0] == 1)
        byteCount = byteCount - 3;

      txNibbles = txNibblesPlus3;
      if (chans2Send)
        linkIFstate = `TXDATA;
      else
        linkIFstate = `TXPADEVEN;    // This is preliminary!!!!!!!!
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      linkIFstate = `TXPADEVENHANDSHAKEERROR;
    end
  end

`TXPADEVEN  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in pad_even state.");
    `endif
    `ifdef DEBUG
      $display("**** SHARCLinkIF: byteCount = %d", byteCount);
    `endif
    if (wack)
    begin
      notLnkTrippleStart = 0;
      if (txNibbles[0] == 1)
      begin
        {notADCOE,notDataOE} = `TXTESTDATA;
        LnkData = 0;
        testData = {testData[ROTESTDATAWIDTH-2:0],testData[ROTESTDATAWIDTH-1]};
        byteCount = byteCount - 3;
```

```
        txNibbles = txNibblesPlus3;
      end
      else
      begin
        {notADCOE,notDataOE} = `TXHEADER;
        LnkData = 0;
        testData = {testData[ROTESTDATAWIDTH-2:0],testData[ROTESTDATAWIDTH-1]};
        byteCount = byteCount - 1;
        txNibbles = txNibblesPlus2;
      end
      linkIFstate = `TXPAD_4_6MINUS2;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      linkIFstate = `TXPADEVEN;
    end
  end
`TXPAD_4_6MINUS2  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in padding 2 byteCount minus 2 state");
    `endif
    `ifdef DEBUG
      $display("**** SHARCLinkIF: byteCount = %d", byteCount);
    `endif
    if(wack)
    begin
      notLnkTrippleStart = 0;
      {notADCOE,notDataOE} = `TXHEADER;
      LnkData = 0;
      byteCount = byteCount - 1;
      txNibbles = txNibblesPlus2;
      if (byteCount)
        linkIFstate = `TXPAD_4_6MINUS2;
      else
        linkIFstate = `TXENDFLTNUMHI;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      linkIFstate = `TXPADEVENHANDSHAKEERROR;
    end
  end


`TXPADEVENGATEERROR  :  begin
    `ifdef DEBUG
      $display("**** Error in SHARCLinkIF: EARLYGATEERR!!");
```

```
    `endif
  if(wack)
    begin
    Errors[`EARLYGATEERR] = 1;
    intErrors[`EARLYGATEERR] = 1;
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in PadEvenGateError state");
    `endif
    notLnkTrippleStart = 0;
    {notADCOE,notDataOE} = `TXHEADER;
    LnkData[9:2] <= FCSFedID[7:0];
    byteCount = byteCount - 1;
    txNibbles = txNibblesPlus2;
    linkIFstate = `TXPADGATEERROR;
    end
  else
    begin
    intErrors[`LATEHANDSHAKEERR] = 1;
    Errors[`LATEHANDSHAKEERR] = 1;
    notLnkTrippleStart = 1;
    {notADCOE,notDataOE} = `TXNOTHING;
    linkIFstate = `TXPADEVENHANDSHAKEERROR;
    end
  end

`TXPADGATEERROR  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in PadGateError state");
    `endif
    if(wack)
    begin
      notLnkTrippleStart = 0;
      {notADCOE,notDataOE} = `TXHEADER;
      LnkData[9:2] = Errors;
      byteCount = byteCount - 1;
      txNibbles = txNibblesPlus2;
      if(byteCount)
        linkIFstate = `TXPADGATEERROR;
      else
        linkIFstate = `TXENDFLTNUMHI;
    end
    else
    begin
      intErrors[`LATEHANDSHAKEERR] = 1;
      Errors[`LATEHANDSHAKEERR] = 1;
      notLnkTrippleStart = 1;
      {notADCOE,notDataOE} = `TXNOTHING;
      linkIFstate = `TXPADHANDSHAKEERROR;
    end
  end

`TXPADEVENHANDSHAKEERROR  :  begin
    `ifdef DEBUG
    $display("**** SHARCLinkIF is in padding even LateHSHK error state");
```

```
    `endif
    `ifdef DEBUG
      $display("**** SHARCLinkIF: byteCount = %d", byteCount);
    `endif
  if(wack)
    begin
      notLnkTrippleStart = 0;
    if (txNibbles[0] == 1)
    begin
      {notADCOE,notDataOE} = `TXTESTDATA;
      LnkData = Errors;
      byteCount = byteCount - 3;
      txNibbles = txNibblesPlus3;
    end
    else
    begin
      {notADCOE,notDataOE} = `TXHEADER;
      LnkData[9:2] = Errors;
      byteCount = byteCount - 1;
      txNibbles = txNibblesPlus2;
    end
    linkIFstate = `TXPADHANDSHAKEERROR;
    end
  else
    begin
    intErrors[`LATEHANDSHAKEERR] = 1;
    Errors[`LATEHANDSHAKEERR] = 1;
    notLnkTrippleStart = 1;
    {notADCOE,notDataOE} = `TXNOTHING;
    linkIFstate = `TXPADEVENHANDSHAKEERROR;
    end
  end

`TXPADHANDSHAKEERROR  :  begin
    `ifdef DEBUG
      $display("**** SHARCLinkIF is in padding error state");
    `endif
    `ifdef DEBUG
      $display("**** SHARCLinkIF: byteCount = %d", byteCount);
    `endif
    if (wack)
    begin
      notLnkTrippleStart <= 0;
      {notADCOE,notDataOE} <= `TXHEADER;
      LnkData[9:2] <= Errors;
      byteCount <= byteCount - 1;
      txNibbles <= txNibblesPlus2;
      if(byteCount)
        linkIFstate = `TXPADHANDSHAKEERROR;
      else
        linkIFstate = `TXENDFLTNUMHI;
    end
    else
    begin
```

```
            intErrors[`LATEHANDSHAKEERR] = 1;
            Errors[`LATEHANDSHAKEERR] = 1;
            notLnkTrippleStart <= 1;
            {notADCOE,notDataOE} <= `TXNOTHING;
            linkIFstate = `TXPADHANDSHAKEERROR;
          end
        end

      `TXENDFLTNUMHI  :  begin
          `ifdef DEBUG
            $display("**** SHARCLinkIF is in trailer high state");
          `endif
          notLnkTrippleStart <= 0;
          {notADCOE,notDataOE} <= `TXHEADER;
          LnkData[9:2] <= TxFLTNum[15:8];
          txNibbles <= txNibblesPlus2;
          linkIFstate = `TXENDFLTNUMLO;
        end

      `TXENDFLTNUMLO  :  begin
          `ifdef DEBUG
            $display("**** SHARCLinkIF is in trailer low state");
          `endif
          notLnkTrippleStart <= 0;
          {notADCOE,notDataOE} <= `TXHEADER;
          LnkData[9:2] <= TxFLTNum[7:0];
          txNibbles <= txNibblesPlus2;
          linkIFstate = `IDLE;
        end

      default  :  begin
          notLnkTrippleStart <= 1;
          {notADCOE,notDataOE} <= `TXNOTHING;
          linkIFstate = `IDLE;
        end

    endcase

  end

endmodule

//
// Increment <In> by 2 modulus 12
//
module Inc2Mod12(In, Out);
parameter
  WIDTH = 4;

input  [WIDTH-1:0] In;
output  [WIDTH-1:0]  Out;
reg    [WIDTH-1:0]  Out;

  always @ (In)
```

```
    case (In)
      0  :  Out <= 2;
      1  :  Out <= 3;
      2  :  Out <= 4;
      3  :  Out <= 5;
      4  :  Out <= 6;
      5  :  Out <= 7;
      6  :  Out <= 8;
      7  :  Out <= 9;
      8  :  Out <= 10;
      9  :  Out <= 11;
      10  :  Out <= 0;
      11  :  Out <= 1;
      default  :   Out <= 0;
    endcase
endmodule
//
// Increment <In> by 3 modulus 12
//
module Inc3Mod12(In, Out);
parameter
  WIDTH = 4;

input  [WIDTH-1:0] In;
output  [WIDTH-1:0]  Out;
reg    [WIDTH-1:0]  Out;

  always @ (In)
    case (In)
      0  :  Out <= 3;
      1  :  Out <= 4;
      2  :  Out <= 5;
      3  :  Out <= 6;
      4  :  Out <= 7;
      5  :  Out <= 8;
      6  :  Out <= 9;
      7  :  Out <= 10;
      8  :  Out <= 11;
      9  :  Out <= 0;
      10  :  Out <= 1;
      11  :  Out <= 2;
      default  :   Out <= 0;
    endcase
endmodule
```

Anhang E

TestProgramme

E.1 Ein Testprogramm zur Initialisierung des Controller Chips

Vor der Benutzung des Front-End Driver muß nach Booten des Controllers dieser noch initialisiert werden. Das folgende Beipielprogramm stellt exemplarisch diesen Vorgang dar. Die einzelnen Schritte des Programms gliedern sich wie folgt:

1. Initialisierung des VME-Bus Zugriffes

2. Zurücksetzen der Karte per Softwarezugriff (Zugriffsadresse # 23)

3. Initialisierung der notwendigen Register (z.B. RoMode, Pipelenght,etc.)

Diese Schritte sind, mit Ausnahme eines expliziten Softwarereset, bei jedem Neustart des Systems durchzuführen, da diese Informationen unerläßlich für den Betrieb sind. Fast alle dieser Schritte sind in der Prozedur "**initcntrl**" gekapselt. Desweiteren haben wir während der Phase der Inbetriebnahme die Datenaufnahme unabhängig von der Fast Control über den VME-Bus gesteuert.

Der letzte Schritt, der sich bis zum Programmabbruch wiederholt, beinhaltet das Triggern des Front-End Driver im "Autotrigger"-Modus.

```
{ This program builds up a testenvironment for the Xilinx FED Controller.
  The sequence goes as follows:
      1. The controller is initialized via a VME Bus write access.
      2. After this start-up sequence the controller is switched into
         the autotrigger mode immediately starting the trigger sequence.
}

program XilinxTest;
```

```
USES DOS,CRT,_IOAVME,_HEXNUM;

var    zeichen   : char;
       datar     :   longint;
       numOfChans : longint;
       fcsbytecount : longint;

const  shortadr  = $ff00;
       xlx       = shortadr;
       status    = shortadr + $80;
       am      = $29;

       {Addresses of DL526 registers. }
       FCSFEDIDLO = 0;
       FCSFEDIDHI = 2;
       FCSRECORDIDLO = 4;
       FCSRECORDIDHI = 6;
       FCSCOMPONENTIDLO = 8;
       FCSCOMPONENTIDHI = 10;
       FCSBYTECOUNTLO = 12;
       FCSBYTECOUNTHI = 14;
       WRITEDLY0 = 16;
       WRITEDLY1 = 18;
       WRITEDLY2 = 20;
       WRITEDLY3 = 22;
       ROPIPEDEPTH = 24;
       ROMODE = 26;
       RONUMOFCHANSLO = 28;
       RONUMOFCHANSHI = 30;
       ROTESTDATALO = 32;
       ROTESTDATAHI = 34;
       ERRORS = 36;
       AUTOTRIGGERFLTNUMLO = 38;
       AUTOTRIGGERFLTNUMHI = 40;
       AUTOTRIGGERBXFLT = 42;
       SOFTWARERESET = 44;
       AUTOTRIGGERSTART = 46;

       { Other constants}
       PIPELENGTH = 127;
       NUMCHAN = 6;

procedure initcntrl;
begin
     writeln('--> Softwarereset');
     vme_wr(0,xlx+SOFTWARERESET,am,vme16);
     delay(10);
     writeln('<-- Softwarereset');
```

```
     writeln('--> initialize Controller');
     vme_wr(5,xlx+FCSFEDIDLO,am,vme16);
     vme_wr(0,xlx+FCSFEDIDHI,am,vme16);
     vme_wr(1,xlx+FCSRECORDIDLO,am,vme16);
     vme_wr(0,xlx+FCSRECORDIDHI,am,vme16);
     vme_wr(4,xlx+FCSCOMPONENTIDLO,am,vme16);
     vme_wr(0,xlx+FCSCOMPONENTIDHI,am,vme16);
     vme_wr(0,xlx+WRITEDLY0,am,vme16);
     vme_wr(0,xlx+WRITEDLY1,am,vme16);
     vme_wr(0,xlx+WRITEDLY2,am,vme16);
     vme_wr(0,xlx+WRITEDLY3,am,vme16);
     vme_wr(PIPELENGTH and $ff,xlx+ROPIPEDEPTH,am,vme16);
     vme_wr(2,xlx+ROMODE,am,vme16);
     vme_wr(0, xlx+ROTESTDATAHI,am,vme16);
     vme_wr(1, xlx+ROTESTDATALO,am,vme16);
end;

{ return the byte count number for a fcs message
  that is divisible by 6 and 4 for a given number of channels.
}
function CalcFCSByteCount(numChans : longint) : longint;

var
   retval : longint;
begin
     retval := 14 + (numOfChans * 3) div 2;
     while ((retval mod 12) <> 0) do
            retval := retval + 1;
     CalcFCSByteCount := retval;
end;

begin   { The core program starts at this point }
     vme_init;
     clrscr;
     initcntrl;

     {... generate autotrigger events...}
     vme_wr(2,xlx+AUTOTRIGGERFLTNUMLO,am,vme16);
     vme_wr(0,xlx+AUTOTRIGGERFLTNUMHI,am,vme16);
     vme_wr(3,xlx+AUTOTRIGGERBXFLT,am,vme16);
     writeln('**** generating auto trigger event....', numOfChans);
     numOfChans := 128;
     fcsbytecount := CalcFCSByteCount(numOfChans);
     repeat
           writeln('numOfChans = ', numOfChans,
                   ' fcsBytecount = ', fcsbytecount);
           vme_wr((numOfChans mod 256) and $ff,xlx+RONUMOFCHANSLO,am,vme16);
           vme_wr((numOfChans div 256) and $ff,xlx+RONUMOFCHANSHI,am,vme16);
           vme_wr((fcsbytecount mod 256),xlx+FCSBYTECOUNTLO,am,vme16);
```

```
            vme_wr((fcsbytecount div 256),xlx+FCSBYTECOUNTHI,am,vme16);
            vme_wr(1 and $ff,xlx+AUTOTRIGGERSTART,am,vme16);
            vme_wr(0 and $ff,xlx+AUTOTRIGGERSTART,am,vme16);
            delay(500);
      until keypressed;
end.
```

E.2 Das Programm zur Auslese des FED Ausgabe Datenformates

Während der Datennahme hat der emulierte Second Level Buffer die Informationen in einer Datei abgelegt. Das hierbei verwendete Datenformat entspricht entspricht dem Front-End Driver Ausgabe Format (s.Abbildung 3.6). Das nachfolgende Programm führt die Schritte zur Datenextraktion durch. Die Vorgehensweise gliedert sich in die folgenden Schritte:

1. Einlesen eines kompletten Ausgabedatensatzes
2. Extraktion der Headerinformationen und Ausgabe auf dem Terminal
3. Zusammensetzen und Konvertierung der 10-Bit Daten und Ausgabe in eine Datei zur späteren, graphischen Weiterverarbeitung

```
/*  This is a small routine for extracting the ADC data from
                a complete FED Output data set.

                Author : Carsten Polenz , Phys. Institut Univ. Heidelberg

                                                                         */

#include <String.h>
#include <Stdio.h>

#define NUMOFNIBBLES    3
#define NUMOFCHANS      128
#define NUMOFDATA       10


int CalcNumOFNibbles(int numofchans)
{
        int numofnibbles;
        numofnibbles = (numofchans*3) + ((numofchans * 3) / 8);
//      printf("NumofNibbles %d \n",numofnibbles);
        return numofnibbles;
        }


unsigned int ConvertHexToInt(char *phead)
```

```
{
        unsigned int nibbleValue;
        if((*phead == 'a') || (*phead == 'b') || (*phead == 'c')
         || (*phead == 'd') || (*phead == 'e') || (*phead == 'f'))
          nibbleValue = *phead - 87;
        else
          nibbleValue = *phead - 48;
//      printf("%d  \n",nibbleValue);
        return nibbleValue;
        }


void main()
{
        FILE *link_ptr;                 /* Definition der Dateizeiger  */
        FILE *adc_ptr;

        int numofbytes = 500;
        int count1,count2,cnt1,stringlength;
        unsigned int  nib1,nib2,nib3,nib4;
        long bindata,binarydata[6];

        char linebuffer[1000];
        char newlinebuffer[1000];

        char    *pheader,*pdata,*ptrailer;
        char c;

        char *link_datei="Linkdata";
        char *adc_datei="Adcdata";


        link_ptr=fopen(link_datei,"r");
        adc_ptr=fopen(adc_datei,"w");


        count2 = 1;
        if((link_ptr==NULL) || (adc_ptr==NULL))
        {
                printf("Fehler beim Oeffnen der datei\n");
        }
        while(fgets(linebuffer,numofbytes,link_ptr) != NULL)
        {
                stringlength = strlen(linebuffer);
                printf("\n\nNeuer Datensatz:");
                printf("Laenge des Strings = %d \n",stringlength);
                pheader = linebuffer;
                pdata = pheader + 26;
                nib1 = 0;
```

```
		cnt1 = 0;

//
// Hier werden die Headerinformationen extrahiert
		while(pheader != pdata)
		{
			// Konvertiere das 1.Nibble
			// Ueberspringe Leerzeichen
			if (*pheader == ' ')
				pheader++;
			// Lese das Nibble in einen Integerwert
			nib1 = ConvertHexToInt(pheader);
			// Springe eine Stelle weiter
			pheader++;

			// Konvertiere das 2.Nibble
			if (*pheader == ' ')
				pheader++;
			nib2 = ConvertHexToInt(pheader);
			pheader++;

			// Konvertiere das 3.Nibble
			if (*pheader == ' ')
				pheader++;
			nib3 = ConvertHexToInt(pheader);
			pheader++;

			// Konvertiere das 4.Nibble
			if (*pheader == ' ')
				pheader++;
			nib4 = ConvertHexToInt(pheader);
			pheader++;

			// Schiebe die einzelnen Nibbles so, dass
			// das 16-Bit Datum wiedergegeben ist
			binarydata[cnt1] = nib4 + (nib3 << 4)
					   + (nib2 << 8) + (nib1 << 12);
			// Erhoehe den Schleifenzaehler
			cnt1++;
		}

		// Dieses ist die Ausgabe der HeaderDaten
		printf("ByteCount = %d \n",binarydata[0]);
		printf("RecordID = %d \n",binarydata[1]);
		printf("FLT Number = %d \n",binarydata[2]);
		printf("BX-FLT # = %d \n",binarydata[3]);
		printf("ComponentID = %d \n",binarydata[4]);
		printf("FEDID = %d \n",binarydata[5]);
```

```
                // Setze einen Zeiger an das Ende der Analog Daten
                ptrailer = pdata + CalcNumOFNibbles(NUMOFCHANS);

                // Schneide den String an dieser Stelle ab
                *ptrailer = '\0';

                // initialisiere den Schleifenzaehler fuer die Auslese
                // der Datenkanaele
                count1 = 0;
                while ((count1 < NUMOFCHANS) && (*pheader != '\0'))
                {
                        // Konvertiere das 1.Nibble des 12-Bit Wortes
                        if (*pheader == ' ')
                                pheader++;
                        nib1 = ConvertHexToInt(pheader);
                        pheader++;

                        if (*pheader == ' ')
                          pheader++;
                        nib2 = ConvertHexToInt(pheader);
                        pheader++;

                        if (*pheader == ' ')
                                pheader++;
                        nib3 = ConvertHexToInt(pheader);
                        pheader++;

                        // Schiebe die Nibbles korrekt zu einem 12-Bit
                        // Wort zuammen
                        bindata = (nib3 >> 2) + (nib2 << 2)
                                    + (nib1 << 6);

                        // Speicher die einzelnen Kanaele
                        // in einem File ab
                        fprintf(adc_ptr,"%d %d\n",count2,bindata);

                        // Ausgabe der Daten auf dem Terminal
                        printf("%d ",bindata);

                        // inkrementiere den Schleifenzeahler
                        count1++;
                        count2++;
                }
        }
        // Schliesse die Dateien
        fclose(link_ptr);
        fclose(adc_ptr);
}
```

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[BAUER] C.Bauer: “APC Readout Chip”;
HERA-B note 95-100

[COMMON] M.Feuerstack, W.Hofmann, D.Ressing: “Common DAQ Architecture”;
HERA-B note

[DESIGN] HERA-B Kollaboration: “HERA-B
An Experiment to study CP-Violation in the B system
using an internal target at the HERA Proton Ring”
Design Report
January 1995 **DESY-PRC 95/01**

[FCSSPEC] D.Ressing: “FCS Specifications”;
HERA-B note

[FEUERSTACK] M.Feuerstack: persönliche Mitteilung

[FORMAT] D.Ressing: “DAQ Architecture & Protocols at HERA-B”;
HERA-B note 95-051

[MEDINNIS] M.Medinnis: “HERA-B Triggering”;
HERA-B note 96-006

[RAUSCH] A.Rausch: persönliche Mitteilung

[RUSNYAK] R.Rusnyak: persönliche Mitteilung

[SHARCDATA] Analog Devices: “ADSP 2106x SHARC User Manual”;
Data Book

[SIGNAL] Signal Processing Technologies Data Book;
Data Book

[WURTH] R.Wurth: “Status of Sharc Board Development (Zeuthen, April 27,1995);
HERA-B note

Danksagung

Herrn Professor Dr. Franz Eisele danke ich für die Aufgabenstellung und die gute Betreuung meiner Arbeit und für die spannende Möglichkeit, ein Experiment der Grundlagenforschung in dieser Größe im Rahmen einer internationalen Kollaboration kennenzulernen.

Mein besonderer Dank gilt Herrn Dr.M.Feuerstack für die intensive Betreuung, die wesentlich zum Gelingen dieser Arbeit beigetragen hat.

Desweiteren möchte ich Herrn Albert Rausch und Herrn Rudolf Rusnyak danken. Sie hatten zu jederzeit ein offenes Ohr für meine Probleme, mochten sie auch noch so trivial sein.

Nicht zuletzt möchte ich mich bei meinen Eltern für die großherzige Unterstützung während meines Studiums bedanken, auch wenn sie sich nie haben vorstellen können, wie man "damit" jemals einen Lebensunterhalt verdienen kann.

Erklärung:

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe.

Heidelberg, den ....