# Die Systemsoftware für den First Level Trigger des HERA-B Experiments

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Thomas Wolf
aus Freiburg im Breisgau
August 1998

| | |
|---|---|
| Vorsitzender: | Prof. Dr. Peter de With, Universität Mannheim |
| Referent und Dekan: | Prof. Dr. Reinhard Männer, Universität Mannheim |
| Koreferent: | Prof. Dr. Ulrich Brüning, Universität Mannheim |

Tag der mündlichen Prüfung: 19. März 1999

**Zusammenfassung**

Für das HERA-B Experiment am HERA-Protonenring des DESY in Hamburg entwickelt der Lehrstuhl für Informatik V die Hard- und Software der ersten Triggerstufe. Ziel des Experiments ist der Nachweis der CP-Verletzung, einer Asymmetrie physikalischer Gesetze bezüglich Materie und Antimaterie, bei Zerfällen massereicher Teilchen, die ein b-Quark enthalten. Zur Kompensation der geringen Produktionswahrscheinlichkeit dieser Teilchen arbeitet das Experiment bei einer Ereignisrate von 10 Mhz. Dies erfordert ein mehrstufiges Datennahme- und Triggersystem, das hoch selektiv und effizient die interessanten Ereignisse herausfiltert. Für ihre Entscheidung verfügt die erste Stufe über 9 $\mu$s, in denen aus den Meßdaten Spuren rekonstruiert und Spurparameter berechnet werden müssen. Sie wird als Multiprozessorsystem mit massiv paralleler und gepipelineter Architektur aufgebaut. Es besteht aus ca. 80 diskret aufgebauten, asynchronen Spezialprozessoren, die durch einen speziellen Bus miteinander verbunden sind, über den sie Nachrichten austauschen. Die Detektordaten werden mit einer Rate von 1 Terabit/s über etwa 1600 optische Verbindungen zu dem System übertragen. Bei Vollast können damit 600 Millionen Spuren/s rekonstruiert werden, die mittlere Last liegt etwa eine Größenordnung darunter.

Es gibt drei verschiedene Prozessortypen, die jeweils auf Spurrekonstruktion, Parameterberechnung und Triggerentscheidung spezialisiert sind. Die Prozessorboards sind als VME-Einschübe ausgeführt, die zusätzlich einen Mikroprozessor zur Steuerung besitzen. Insgesamt besteht das heterogene Rechnersystem zur Überwachung und Steuerung des Triggers aus etwa 90 Rechnern, die über VME-Bus oder Ethernet miteinander kommunizieren. Während die zentrale Server-Workstation und die VME-Host-Rechner mit Unix-Systemen betrieben werden, besitzen die Board-Mikroprozessoren kein Betriebssystem und nur den VME-Bus als Schnittstelle nach außen. Im Rahmen dieser Arbeit wurde eine Entwicklungs- und Laufzeitumgebung implementiert, die das Programmieren dieser Rechner in C und, durch Kapselung der VME-Bus Kommunikation, die Verwendung der C-Standardbibliothek erlaubt.

Weiterhin wurde ein Prototyp für ein zentrales Trigger-Steuerprogramm, mit grafischer Benutzeroberfläche und Netzwerkkommunikation zu den einzelnen Board-Prozessen, in der Skriptsprache Tcl/Tk entwickelt.

Für Hardwaretests, die Inbetriebnahme und physikalische Fragestellungen wird eine Simulation der ersten Triggerstufe und der ihr vorgeschalteten drei Pretriggersysteme, die an anderen Instituten entwickelt werden, benötigt. Im Rahmen der Arbeit wurde ein objektorientiertes Framework als Basis eines gemeinsamen Simulationsprogramms entwickelt. Mit seiner Hilfe können logische Simulationen digitaler Schaltungen implementiert werden. Es wird von den verteilten Gruppen eingesetzt, um auf kohärente Weise ihre Subsystemsimulationen zu entwicklen und die einzelnen Teile anschließend in einer Gesamtsimulation zusammenfügen zu können.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Kapitel 1

# Einleitung

Neue Experimente in der Hochenergiephysik stellen immer höhere Anforderungen an die elektronische Datenverarbeitung. Die Experimente gehören zu den anspruchsvollsten Anwendungen überhaupt und bewegen sich an der Grenze des technisch Machbaren.

Die Eingangs-Datenraten der Datennahmesysteme liegen momentan in der Größenordnung von Terabytes pro Sekunde. Die Triggersysteme müssen unter Echtzeitbedingungen automatische Analysen dieser Daten vornehmen. Da dies die Rechenleistung einzelner Computer um viele Größenordnungen übersteigt, werden für diese Aufgaben parallele Rechnerarchitekturen eingesetzt.

Zukünftige Datennahmesysteme basieren daher auf mehrstufigen (in der Regel drei bis vier) Pipeline-Architekturen [59]. Dabei wird in jeder Stufe der gesamte Datensatz einer Messung zwischengespeichert, während die zugeordnete Triggerstufe an der Entscheidung über die Weitergabe der Daten zur nächsten Stufe arbeitet.

Entsprechend sind die Triggersysteme ebenfalls mehrstufig aufgebaut und operieren auf den vom Datennahmesystem zur Verfügung gestellten Informationen. Dies beginnt in der ersten Stufe, die ihre Mustererkennungsaufgaben aus den Meßdaten mit Bearbeitungszeiten in der Größenordnung von $\mu$s wahrnehmen muß. Solche Verarbeitungsgeschwindigkeiten lassen sich nur mit kommerziell nicht erhältlichen Speziallösungen, die in ihrer Hardwarearchitektur an die spezielle Aufgabe angepaßt sind, erreichen. Es endet in der letzten Stufe mit Rechner-Farmen, die aus kommerziell erhältlichen, mikroprozessorbasierten Systemen aufgebaut sind, unter normalen Unix-Versionen betrieben werden und für eine Analyse Zeiten in der Größenordnung von 100 ms benötigen.

Mit dem HERA-B Experiment versucht man, eine der wichtigsten offenen Fragen in der Hochenergiephysik zu klären: die Ursache der Verletzung der CP-Symmetrie, die bereits 1964 bei Zerfällen von K-Mesonen entdeckt wurde. Die CP-Verletzung besagt, daß es einen fundamentalen Unterschied in der Wirkung einer der Grundkräfte der Natur, vermutlich der schwachen Wechselwirkung, auf Materie und Antimaterie

gibt. Die CP-Verletzung ist beispielsweise eine mögliche Erklärung für das Fehlen der Antimaterie im Universum [43]. Bei Symmetrie zwischen Materie und Antimaterie hätte bei dem Urknall jeweils gleich viel von beiden Arten entstehen müssen. Da es aber nach heutigem Wissen keine größeren Vorkommen von Antimaterie im Universum gibt, muß bei dem Urknall ein Materieüberschuß entstanden sein. Die Symmetrie ist also nicht gegeben, möglicherweise aufgrund der CP-Verletzung.

Das HERA-B Experiment soll die von der Theorie vorhergesagte CP-Verletzung in Zerfällen von B-Mesonen entdecken und vermessen. Die CP-Verletzung äußert sich dadurch, daß bestimmte Zerfälle von B-Mesonen für ein Teilchen und sein Antiteilchen nicht gleichwahrscheinlich sind. Für die Erzeugung der B-Mesonen werden Protonen aus dem HERA-Protonenring auf ein feststehendes Target geschossen. Dies geschieht mit einer Periode von 96 ns, und die dabei entstehenden Reaktionsprodukte werden mit dem HERA-B Detektor nachgewiesen. Die Meßsysteme des gesamten Detektors erzeugen eine Datenrate von 10 Terabyte/s.

## 1.1 Motivation

An der HERA-B Kollaboration am DESY sind zur Zeit 32 Institute aus 14 Ländern mit insgesamt etwa 250 Wissenschaftlern beteiligt. Die Mannheimer Gruppe, in der diese Arbeit entstand, hat im Rahmen des Experiments die Entwicklung und den Bau der ersten Triggerstufe, des First Level Triggers (FLT), übernommen. Die physikalischen Simulationen und die daraus abgeleiteten Spezifikationen des FLT werden am DESY durchgeführt.

Ein großes Problem bei dem Experiment stellt das Verhältnis von Signal zu Untergrund von nur etwa $10^{-11}$ dar. Es wird daher ein sehr leistungsfähiges Triggersystem benötigt, das eine starke Reduktion des Untergrundes bei gleichzeitig hoher Nachweiseffizienz für die gesuchten Ereignisse ermöglicht. Die Ereignisrate soll in der ersten Stufe um den Faktor 200 reduziert werden, von 10 MHz auf 50 kHz. Die Trigger Latenzzeit der ersten Stufe darf dabei maximal 12,3 $\mu$s betragen (von denen dem FLT effektiv nur etwa 9 $\mu$s verbleiben).

Die Triggerentscheidung der ersten Stufe basiert auf der Identifizierung von Sekundärteilchen aus Zerfällen von neutralen B-Mesonen. Dazu muß eine Echtzeitrekonstruktion von Teilchenspuren aus den Detektordaten durchgeführt werden. Initiiert wird die Spurensuche des FLT grundsätzlich durch eines der drei Pretriggersysteme, indem es dem FLT Informationen über potentiell interessante Spurkandidaten übermittelt. Nach erfolgter Rekonstruktion werden Spurparameter errechnet und überprüft, die rekonstruierten Spuren paarweise kombiniert und mit der invarianten Masse des gesuchten Teilchens verglichen. Damit kann mit hoher Effizienz selektiv auf bestimmte Zerfallskanäle getriggert werden.

Diese Anforderungen können nur mit einem Spezialrechner erfüllt werden, dessen Hardware für diese Aufgabenstellung optimiert ist. Der FLT wird deshalb als Multi-

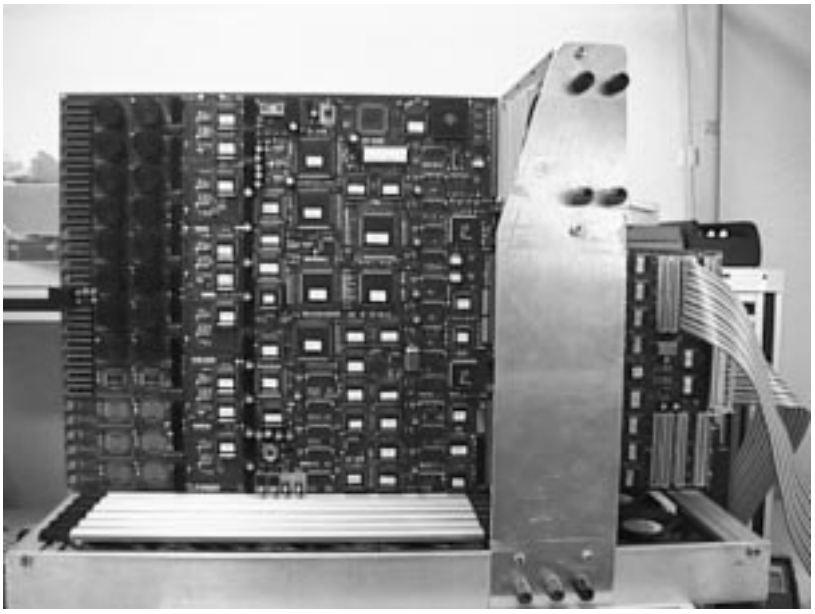

Abbildung 1.1: *Das Foto zeigt ein Prozessorboard für die Spurensuche in einem Experimentier-Crate. Von der Rückseite steckt ein Kommunikationsboard für Verbindungen zu anderen Prozessoren an dem VME-Steckplatz.*

prozessorsystem mit Pipeline- und massiver Parallel-Architektur realisiert, das aus etwa 80 VME-Bus-Boards der Größe 36 x 39 $cm^2$ besteht. Es werden drei verschiedene Typen von Spezialprozessoren eingesetzt: je einen für die Spurensuche, die Spurparameter-Berechnung und die Triggerentscheidung. Ein Prozessorboard zur Rekonstruktion von Teilchenspuren zeigt Abbildung 1.1.

Die für die erste Stufe relevanten Detektordaten werden mit 1600 Glasfaserverbindungen 60 m weit vom Detektor zu den Spursuche-Prozessoren des FLT-Systems übertragen. Insgesamt wird, zusammen mit den Kontrollinformationen, eine Datenrate von etwa 1,2 Terabit/s erreicht.

Das System ist modular und flexibel aufgebaut und kann bei Vollast in der geplanten Konfiguration bis zu 600 Millionen Spuren/s rekonstruieren. Die Prozessoren arbeiten asynchron und tauschen die Spurdaten über ein spezielles Bussystem aus. Sie sind diskret aus programmierbaren Logikbausteinen aufgebaut, für Berechnungen werden hauptsächlich Tabellen verwendet. In den vorberechneten Tabellen muß dann nur noch das Ergebnis nachgeschaut werden, was eine hohe Verarbeitungsgeschwindigkeiten und eine gewisse Flexibilität ermöglicht. Jedes Prozessorboard besitzt zusätzlich einen MC68020 Prozessor für Initialisierung, Kontroll- und Steueraufgaben.

Die Boards sind in neun VME-Crates untergebracht, die mit jeweils einem Host-Rechner ausgestattet sind, der unter einem Echtzeit-Unix-System (LynxOS) betrieben wird. Eine zentrale Unix-Workstation (Linux) dient als Boot- und Fileserver für

das ganze System. Die Steuerung des Triggers und die Entwicklungsumgebungen für die Software werden ebenfalls von der Workstation aus bedient.

Insgesamt bildet die Umgebung des FLT ein verteiltes vernetztes heterogenes Rechnersystem. Die Heterogenität ist durch die verschiedenen Prozessortypen (MC68020, MC68060, PowerPC604, Pentium), Betriebssysteme und die unterschiedlichen Schnittstellen (VME-Bus, Ethernet) begründet.

Für den Betrieb, den Test der Hardware und die Simulation dieses komplexen Systems sind umfangreiche Softwareentwicklungen notwendig.

## 1.2 Zielsetzung

Schwerpunkt dieser Arbeit ist die Konzeption der generellen Softwarearchitektur für den FLT nach dem Stand der Technik und die Implementierung der Teile, die in diesem Konzept der Systemsoftware zugeordnet werden. Die Systemsoftware setzt sich aus Basis-Softwarekomponenten zusammen, auf denen dann die einzelnen speziellen Applikationen, zum Beispiel Test- oder Betriebsprogramme aufbauen. Damit legt sie natürlich deren Struktur in weiten Teilen fest, entbindet aber den Entwickler davon, sich mit immer wiederkehrenden Dingen, wie zum Beispiel der VME-Bus Kommunikation, auseinandersetzen zu müssen. Dies reduziert den Entwicklungsaufwand und das benötigte Hintergrundwissen.

Die Systemsoftware beinhaltet eine Entwicklungs- und Laufzeitumgebung für C-Programme auf den 68K-Boards. Die Laufzeitumgebung enthält eine an die Triggerhardware angepaßte Version der C-Standardbibliothek, die einen Teil der Unix-Betriebssystemschnittstelle über den VME-Bus auf die Board-Rechner exportiert. Weiterhin gehören zu der Systemsoftware die Komponenten für die Kommunikation der verteilten Prozesse über Ethernet oder VME-Bus. Für die zentrale Steuerung und Überwachung des Gesamtsystems wird ein komfortabel zu bedienendes Programm mit grafischer Benutzeroberfläche benötigt. Hierfür wurde das Werkzeug zur Oberflächenentwicklung ausgewählt und ein Prototyp in der Skriptsprache Tcl/Tk geschrieben.

Bei allen Softwareentwicklungen gelten die Nebenbedingungen, daß sie während der gesamten Aufbau- und Betriebsphase des Experiments, insgesamt etwa 8 Jahre, benutzt und unter Umständen neuen Anforderungen angepaßt werden müssen. Weiterhin sollen nicht nur Experten der Triggerhardware in der Lage sein, das Multiprozessorsystem zu bedienen und die Trigger-Boards zu programmieren. Daher werden bei allen Entwicklungen Methoden eingesetzt, um zu guten Programmentwürfen zu kommen und die Bedienbarkeit, Wartbarkeit und die Dokumentation der Software sicherzustellen.

In besonderem Maße gilt dies für die Simulation des Triggersystems. Hier muß das gesamte System aus FLT und den drei Pretrigger-Systemen mit seinem zeitlichen Verhalten simuliert werden. Teilweise baut diese detaillierte Simulation auf Kompo-

nenten auf, die zuvor auch im Rahmen von Hardwaretests verwendet werden. Die Entwicklung der Simulation ist auf vier bis fünf Institute verteilt, die jeweils für ihre Subsysteme zuständig sind.

Um diesem umfangreichen Entwicklungsprojekt unter schwierigen Randbedingungen gerecht zu werden, werden objektorientierte Design Methoden und CASE-Werkzeuge eingesetzt. Zudem sind mit der Verwendung von Entwurfsmustern und der Entwicklung eines Frameworks neueste Entwicklungen aus dem Bereich der objektorientierten Softwaretechnik berücksichtigt.

Im Rahmen dieser Arbeit wurde ein objektorientiertes Framework für die logische Simulation digitaler Schaltungen entwickelt. Auf seiner Grundlage programmieren die Entwickler in den jeweiligen Instituten die Simulation ihrer speziellen Triggerkomponenten. Das Framework setzt die Basisabstraktionen des zugrundeliegenden Modells um und garantiert die Lauffähigkeit der getrennt enwickelten Simulationen der Subsysteme in einem gemeinsamen Programm.

Das folgende Kapitel gibt einen Überblick über die Fragestellungen des Experiments und das Umfeld, in dem der FLT eingesetzt wird. Nach einer kurzen Einführung in die physikalischen Grundlagen werden die Maschinen beschrieben, mit denen die Teilchen erzeugt und ihre Zerfallsprodukte nachgewiesen werden. Das Kapitel schließt mit der Beschreibung des Datennahme- und Triggersystems, in das der FLT direkt eingebunden ist.

Kapitel 3 geht detailliert auf den FLT ein. Zuerst werden die Anforderungen an den Trigger und der Triggeralgorithmus vorgestellt. Daran schließt sich eine Beschreibung des Multiprozessorsystems an, mit dem der Algorithmus unter den vorgegeben Randbedingungen umgesetzt wird. Es folgt der Aufbau der verschiedenen Prozessorboards und ihrer Kommunikationssysteme.

Kapitel 4 beginnt mit einem Überblick über die Hard- und Software-Umgebung, in der die benötigte Software für den FLT zum Einsatz kommen soll. Umfang, Entwicklungs- und Einsatzbedingungen der Software machen, in vernünftigem Rahmen, die Verwendung von Software Engineering Methoden sinnvoll. Als erster Schritt hierzu wurde ein Gesamtkonzept entworfen, das eine Analyse des Bedarfs an Software für den FLT vornimmt und eine klare Modularisierung der Software zum Ziel hat.

Kapitel 5 beschreibt die Implementierung einer Umgebung zum Erstellen und Ablaufenlassen von C-Applikationen für die Prozessorboards. Es beinhaltet auch eine Anbindung der Board-Prozesse an das Netzwerk. Die Netzwerkkommunikation erfolgt mit einem zentralen Steuerprozeß. Ein Prototyp für diesen Prozeß mit grafischer Benutzeroberfläche wird in Kapitel 6 vorgestellt.

Kapitel 7 befaßt sich mit dem objektorientierten Framework, das für die Implementierung der Subsystemsimulationen entwickelt wurde.

# Kapitel 2

# Das HERA-B Experiment

In diesem Kapitel wird gezeigt, wie die in der Einleitung angesprochene CP-Verletzung in das Standardmodell der Elementarteilchenphysik einzuordnen ist und warum man hofft, gerade bei Zerfällen von B-Mesonen neue Erkenntnisse darüber zu erhalten. Anschließend wird dargestellt, welcher Weg bei dem HERA-B Experiment gewählt wurde, um die benötigten B-Mesonen zu erzeugen und welche apparativen Möglichkeiten für ihre Erzeugung zur Verfügung stehen. Schließlich wird ein Überblick über den Detektor gegeben mit dem die B-Mesonen und ihre Zerfallsprodukte nachgewiesen werden. An den Detektor schließt sich ein äußerst leistungsfähiges System zur Datennahme und automatischen Analyse der Detektordaten an, zu dessen Bestandteilen auch der First Level Trigger gehört.

## 2.1 CP-Verletzung von B-Zerfällen im Standardmodell

Das Standardmodell der Elementarteilchenphysik beschreibt die elementaren Bausteine der Materie und ihre Wechselwirkungen in bisher nie gekannter Genauigkeit und Vollständigkeit.

Alle Materie und der gesamte, zum Beispiel in Beschleunigern erzeugbare, hadronische Teilchenzoo lassen sich auf zwei Gruppen von elementaren Teilchen zurückführen: sechs Quarks und sechs Leptonen. Zusammenen mit ihrem jeweiligen Antiteilchen ergibt dies 24 elementare Bausteine aus denen alle bekannten Teilchen aufgebaut sind.

Die Wechselwirkungen werden durch nichtabelsche Eichtheorien beschrieben: die Quantenfeldtheorien der starken- und elektroschwachen Wechselwirkung. In diesen Theorien sind die Ursache der Kräfte jeweils Ladungen und die Kraftübertragung erfolgt durch, für die Ladungsart spezifische, Austauschteilchen.

Nach heutigem Kenntnisstand gibt es keine gesicherten, ernstzunehmenden Abweichungen zwischen den experimentellen Daten und den Vorhersagen des Standardmodells [30]. Viele der Schlüsselkonstanten des Modells sind mit sehr großer Genauigkeit bekannt. Die Korrektur der Feinstrukturkonstanten auf Effekte der Vakuumpolarisation und die Masse des Higgs-Teilchens stellen nunmehr die größten Unsicherheiten für theoretische Vorhersagen dar. Momentan versucht man durch immer genauere Experimente die Vorhersagen zu überprüfen, um möglicherweise auf Effekte jenseits des Standardmodells zu stoßen. Dies führt zu den drei wichtigsten ungelösten Fragen im Zusammenhang mit dem Standardmodell:

1. Die Existenz des Higgs-Teilchens.
   Sie wird postuliert um die mathematische Konsistenz der Theorie zu ermöglichen. Die obere Massengrenze des Higgs-Teilchens kann mittlerweile mit 430 GeV angegeben werden. Damit wäre es, falls es existiert, mit dem geplanten Hadron-Collider LHC am CERN nachweisbar. [30]

2. Neutrinooszillationen.
   In der aktuellen Forschung häufen sich die Hinweise, daß sich die Neutrinosorten ineinander umwandeln können. Dies würde bedeuten, daß die Neutrinos Ruhemasse besitzen; ein einheitliches Bild ergeben die Experimente aber noch nicht. [30]

3. CP-Verletzung.
   Die CP-Verletzung läßt sich mathematisch in das Standardmodell einbauen, ohne aber ihre Ursache wirklich zu erklären.

Das HERA-B Experiment möchte dem Phänomen der CP-Verletzung auf den Grund gehen, indem es den Protonenstrahl von HERA für die Erzeugung von B-Mesonen nutzt. Die CP-Verletzung wurde 1964 bei Zerfällen neutraler K-Mesonen entdeckt. Der Effekt liegt in der Größenordnung von $10^{-3}$ und wurde bis heute nirgendwo sonst beobachtet. Die Ursache der CP-Verletzung ist noch ungeklärt und man versucht daher die experimentelle Basis zu erweitern um ein besseres Verständnis dieses Phänomens zu erreichen. Der Zerfall neutraler B-Mesonen ist einer der wenigen Prozesse in der Natur, bei dem man möglicherweise ebenfalls eine Verletzung der CP-Symmetrie beobachten kann. Außerdem kann man dabei die Vorhersagen des Standardmodells über den Mechanismus der CP-Verletzung überprüfen.

Wie in dem System für Kaonen wird für einige B-Zerfälle starke CP-Asymmetrie vorhergesagt. Besonders geeignet ist der Zerfall

$$B^0 \to J/\psi K_S^0, \tag{2.1}$$

weil er, neben der vorhergesagten starken CP-Asymmetrie, über die Sekundärzerfälle $J/\psi \to \mu^+\mu^-$ bzw. $J/\psi \to e^+e^-$ und $K_S^0 \to \pi^+\pi^-$ zu einer klaren experimentellen

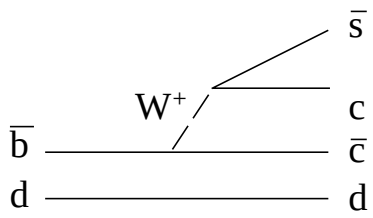


Abbildung 2.1: *Der goldene Zerfall eines $B^0$-Mesons durch schwache Wechselwirkung in ein $J/\psi$ $(c,\bar{c})$ und ein $K_S^0$ $(d,\bar{s})$.*

Signatur führt - einem Leptonpaar und einem $K_S^0$-Zerfall. Der Zerfall wird deswegen auch als „goldener Zerfall“ bezeichnet.

Abbildung 2.1 zeigt das Feynman Diagramm des Zerfalls. Das $B^0$-Meson zerfällt, indem sein $\bar{b}$-Quark sich unter Abstrahlung eines $W^+$ in ein $\bar{c}$-Quark umwandelt. Dabei entstehen ein $J/\psi$- $(c,\bar{c})$ und ein $K_S^0$-Meson $(d,\bar{s})$. Die CP-Verletzung äußert sich in geringfügig verschiedenen Verzweigungsverhältnissen der Zerfälle $B^0 \to J/\psi K_S^0$ und $\bar{B}^0 \to J/\psi K_S^0$ .

Eine zusätzlich Schwierigkeit ergibt sich aus der Tatsache, daß aus dem ladungssymmetrischen Endzustand nicht geschlossen werden kann, ob es sich ursprünglich um eine $B^0$ oder ein $\bar{B}^0$ gehandelt hat. Deshalb ist man gezwungen auch das zweite B-Meson, das aus dem ursprünglichen $b\bar{b}$-Paar entstanden ist, nachzuweisen (Abbildung 2.2). Ist man in der Lage aus dessen Zerfall zu entscheiden ob es sich um ein $\bar{b}$- oder ein $b$-Quark handelt [1] , so muß in dem goldenen Zerfall das jeweilige Antiteilchen enthalten sein. Das zweite B-Meson wird als „Tagging B“ bezeichnet, in dem gezeigten Beispiel ist es ein $B^-$.

In diesen Zerfällen der schwachen Wechselwirkung ist generell ein Quark der Ladung 2/3 und eines der Ladung -1/3 beteiligt, die ineinander umgewandelt werden. Bei insgesamt sechs Quarks ergeben sich daraus neun mögliche Quarkumwandlungen. Für jedes Quarkpaar $ik$ erhält man eine spezifische Amplitude, die Kopplungskonstante $V_{ik}$, die die jeweilige Umwandlungswahrscheinlichkeit ausdrückt.

Die Kopplungskonstanten für alle Quarkpaare lassen sich in einer $3 \times 3$ Matrix zusammenfassen, der CKM-Matrix [2] .

$$CKM = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td1} & V_{ts} & V_{tb} \end{pmatrix} \tag{2.2}$$

[1] Wenn der Nachweis gelingt ist dies praktisch immer der Fall, da es extrem unwahrscheinlich ist, das das zweite B-Meson ebenfalls ladungssymmetrisch zerfallen ist.

[2] Cabibbo-Kobayashi-Maskawa Quark Mischungs Matrix

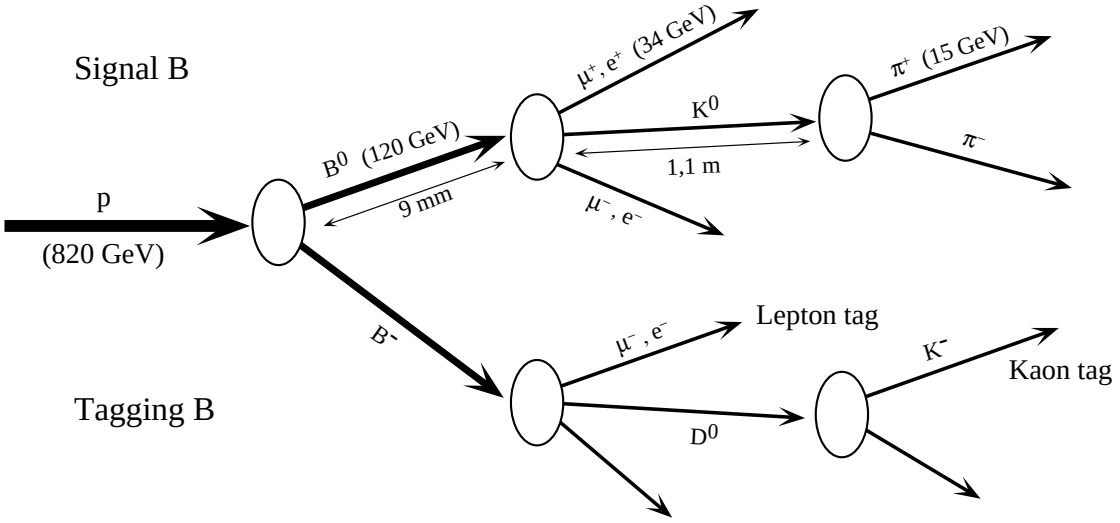


Abbildung 2.2: *Die experimentelle Signatur des goldenen Zerfalls $B^0 \to J/\psi K_S^0$ („Signal B“) zusammen mit Erwartungwerten für die Energie und Zerfallslänge der Sekundärteilchen. Aus den Reaktionsprodukten ist nicht erkennbar, ob es sich ursprünglich um ein $B^0$ oder $\bar{B}^0$ handelt. Hierzu muß das „Tagging B“ herangezogen werden. Das $B^-$ besitzt ein b-Quark, entsprechend muß dann das „Signal B“ ein $\bar{b}$ enthalten, es ist also ein $B^0$.*

Nach dem Standardmodell muß die CKM-Matrix unitär sein [49]. Die Unitaritätsbedingung lautet:

$$\sum_j V_{ji} V_{jk} = \delta_{ik}. \tag{2.3}$$

Aufgrund dieser Beziehung reduziert sich die Zahl der unabhängigen Parameter der CKM-Matrix auf vier. Von allen Unitaritätsbeziehungen, die sich aus der obigen Bedingung ergeben ist die für die B-Physik relevante Gleichung

$$V_{ud} V_{ub}^* + V_{cd} V_{cb}^* + V_{td} V_{tb}^* = 0. \tag{2.4}$$

Sie läßt sich in der Ebene der komplexen Zahlen als Unitaritätsdreieck darstellen. Dabei sind die vier unabhängigen Parameter als drei reelle Winkel und eine imaginäre Phase gewählt [3]. Die Fläche des Dreiecks (Abbildung 2.3) ist ein Maß für die Größe der CP-Verletzung.

Die Anwendung der CP Operation entspricht der Überführung der $V_{ik}$ in das konjugiert komplexe,

$$V_{ik} \xrightarrow{CP} V_{ik}^*. \tag{2.5}$$

[3] Mit Hilfe der Wolfenstein-Parametrisierung, auf die hier aber nicht näher eingegangen werden soll.

Wäre die CKM-Matrix reell, dann wäre damit die schwache Wechselwirkung CP-invariant ($V_{ik} = V_{ik}^*$) und die Fläche des Unitaritätsdreiecks gleich Null. Die CP-Verletzung äußert sich also in der Existenz eines Imaginärteils der CKM-Matrix.

Bei dem Experiment geht es um den Nachweis dieses Imaginärteils, und damit der Unitarität, der CKM-Matrix. Falls er nicht nachweisbar ist, wäre dies ein Hinweis auf eine Unitaritätsverletzung der CKM-Matrix und stünde im Wiederspruch zu dem Standardmodell. Die Messungen bei HERA-B führen direkt zur Bestimmung einiger der Matrixelemente aus den Kantenlängen und Winkeln des Unitaritätsdreiecks in Abbildung 2.3:

- Linke Seite des Dreiecks: Mit Übergängen vom Typ $b \to u$ kann diese Länge bestimmt werden.
- Rechte Seite des Dreiecks: Die Länge kann aus der bei HERA-B meßbaren Frequenz der $B_s^0 - \bar{B}_s^0$ - Oszillation ermittelt werden.
- Winkel $\alpha$: Im Zerfall $B^0 \to \pi^+\pi^-$ ist die Asymmetrie proportional zu $\sin(2\alpha)$.
- Winkel $\beta$: Im Zerfall $B^0 \to J/\psi K_S^0$ ist die Asymmetrie proportional zu $\sin(2\beta)$.

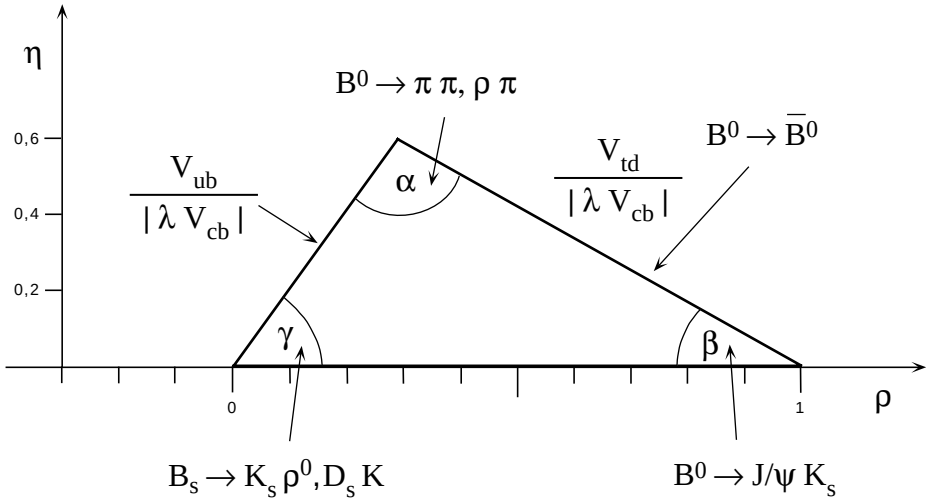


Abbildung 2.3: *Das Unitaritätsdreieck der CKM-Matrix in der komplexen Ebene und physikalische Prozesse, aus denen sich seine Winkel oder Seitenlängen bestimmen lassen.*

## 2.2 Die Erzeugung von B-Mesonen

Während für die Suche nach neuen Teilchen und insbesondere dem postulierten Higgs-Teilchen immer größere und energiereichere Beschleuniger gebaut werden, ist die Untersuchung der CP-Verletzung mit B-Mesonen bei vergleichsweise geringen

Energien möglich. Man benötigt allerdings hohe Ereigniszahlen um die erforderlich Meßgenauigkeit zu erreichen, da die Verzweigungsverhältnisse der interessanten Zerfälle klein sind.

Die beste Möglichkeit zur Erzeugung von B-Mesonen bieten $e^+$ $e^-$-Maschinen, die mit der Schwerpunkstenergie der $\Upsilon$-Resonanz betrieben werden. Hier werden die B-Mesonen praktisch ohne jeden Untergrund erzeugt - entsprechend gut lassen sich dann ihre Zerfälle rekonstruieren. Trotz der vergleichsweise niedrigen Energie stellt dies jedoch neue Anforderungen an die Technologie der Beschleuniger. Zum einen wird eine sehr hohe Luminosität von mindestens $10^{33} cm^{-2} s^{-1}$ benötigt um die erforderlichen Ereigniszahlen zu erreichen. Zum anderen muß der Speicherring asymmetrisch bezüglich der Strahlenergien aufgebaut werden. Dies ist notwendig, weil bei einem symmetrischen Ring das $\Upsilon$-Teilchen praktisch in Ruhe erzeugt wird und die B-Mesonen aufgrund ihrer kurzen Lebensdauer nur ca. 25$\mu$m bis zu ihrem Zerfall zurücklegen. Das Auflösungsvermögen heutiger Detektoren ist zu gering um diese Zerfallsvertices zu vermessen. Dies ist jedoch notwendig, da man für die Bestimmung der CP-Asymmetrie die Zerfallszeiten beider B-Mesonen messen muß.

In einer Studie [2] wurde ein Konzept für eine solche Anlage am DESY entwickelt. Dabei werden zwei Speicherringe in dem ehemaligen PETRA Tunnel mit Strahlenergien von 9,3 und 3 GeV vorgeschlagen. Von einer Realisierung wurde jedoch hauptsächlich aus finanziellen Gründen Abstand genommen. Es gibt aber zwei Konkurrenzexperimente zu HERA-B die mit asymmetrischen Speicherringen die CP-Verletzung bei B-Zerfällen untersuchen wollen. Bei SLAC in den USA und am KEK in Japan befindet sich jeweils eine solche B-Fabrik im Bau.

Das HERA-B Experiment nutzt statt der $e^+$ $e^-$-Annihilation die Möglichkeit B-Mesonen durch hochenergetische hadronische Wechselwirkung zu erzeugen. Die Herausforderung liegt in diesem Fall in erster Linie auf der Seite des Detektors, der aus einem sehr hohen Untergrund effizient die interessanten Ereignisse herausfiltern muß.

Der große Vorteil ist, daß der bereits existierende Protonen-Speicherring als Quelle für B-Mesonen benutzt werden kann.

### Der Speicherring HERA

Der HERA Speichering am DESY in Hamburg ist der erste Beschleuniger weltweit, der aus zwei getrennten Ringen besteht in denen Elektronen (bzw. Positronen) und Protonen entgegengesetzt umlaufen und zur Kollision gebracht werden können (Abbildung 2.4). Die Ringe haben einen Umfang von 6336 m. In Tabelle 2.2 sind die wichtigsten HERA-Parameter zusammengestellt. Der Protonenring ist mit supraleitenden Magneten ausgerüstet, die eine Strahlenergie von maximal 820 GeV ermöglichen. Der Elektronenring ist für eine Energie von maximal 30 GeV ausgelegt, zusammen ergibt dies eine Schwerpunktsenergie von 314 GeV.

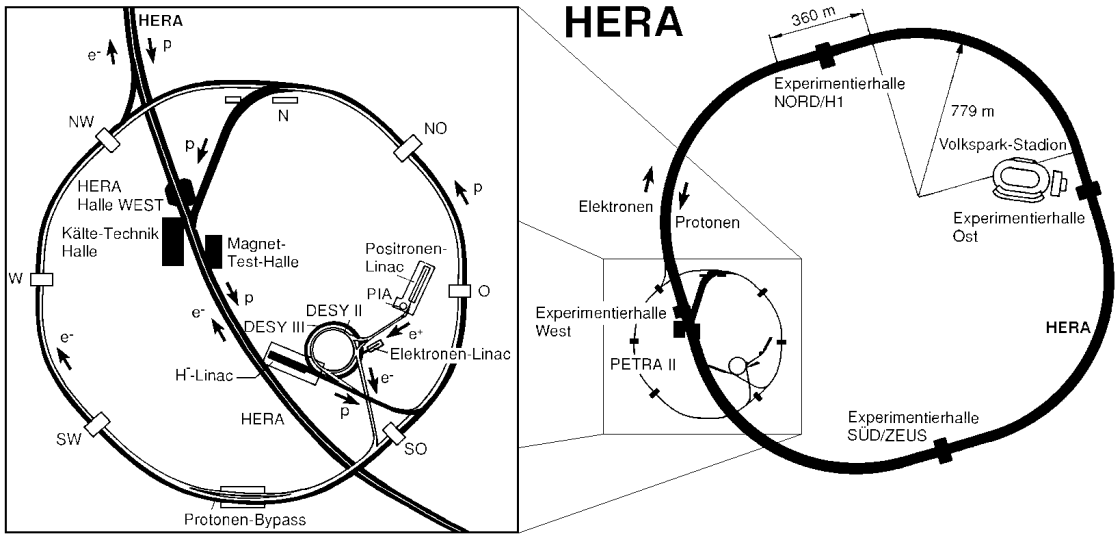


Abbildung 2.4: *Der HERA-Speichering. Im linken Teilbild ist das DESY-Areal mit den Vorbeschleunigern und der Halle West, in der sich das HERA-B Experiment befindet vergrösert dargestellt.*

Um die erforderliche Einschußenergie für HERA zu erreichen, müssen die Teilchen ein komplexes System aus Linearbeschleunigern und Synchrotrons durchlaufen (Abbildung 2.4 linker Teil). Aus diesem Vorbeschleunigersystem heraus werden die Teilchen dann paketweise (Bunche) in den HERA-Speicherring gefüllt.

Es kreisen maximal 210 Pakete in jedem Ring, was einem zeitlichen Abstand von 96 ns zwischen zwei Paketen entspricht. Das heißt, die Pakete durchlaufen mit einer Rate von ca. 10,4 MHz die Experimente. Hierfür besitzt HERA vier sogenannte Wechselwirkungszonen, in denen jeweils eines der vier Experimente untergebracht ist. An Zweien werden Protonen und Elektronen zur Wechselwirkung gebracht: ZEUS in der Experimentierhalle Süd und H1 in der Experimentierhalle Nord. Beide Experimente untersuchen die tiefinelastische Elektron-Proton Streuung. Jedes Experiment besteht aus einem Vielzweckdetektor der den gesamten Raum um den Wechselwirkungspunkt umschließt. Damit sollen möglichst alle Sekundärteilchen der ep-Streuung nachgewiesen, identifiziert und vermessen werden.

Die beiden anderen Experiment verwenden ruhende Tartgets. Das HERMES-Experiment in der Experimentierhalle Ost verwendet nur den Elektronenstrahl, der mit einem internen Gastarget zur Wechselwirkung gebracht wird. Das Gastarget wird z.B. mit polarisiertem $He_3$ betrieben um die innere Spinstruktur von Protonen und Neutronen zu untersuchen. Dabei soll gemessen werden, auf welche Weise sich der Spin der Nukleonen aus den Spins ihrer Bestandteile, den Quarks und Gluonen, zusammensetzt.

Das vierte (und jüngste) Experiment wiederum verwendet nur den Protonenstrahl, der mit einem internen Drahttarget zur Wechselwirkung gebracht wird. Das HERA-B Experiment ist in der Halle West auf dem DESY Gelände untergebracht. Sein vorrangiges Ziel ist die Aufklärung der Ursache der CP-Verletzung, die bei dem Zerfall von B-Mesonen untersucht werden soll.

Bei dieser Verwendung des HERA-Speicherrings als B-Fabrik werden mit einer Schwerpunktsenergie von 40 GeV voraussichtlich bis zu $10^9$ B-Hadronen pro Jahr erzeugt.

| *Generelle HERA Parameter* | |
|---|---|
| Beginn der Messungen | 1992 |
| Durchmesser des HERA-Rings | 6336 m |
| Innendurchmesser des Tunnels | 5,2 m |
| Bunch Crossing Periode | 96 ns |
| Anzahl der Bunche | 210 |
| Luminosität | $(0,2-1,6)\cdot 10^{30} cm^{-2} s^{-1}$ |

| *Strahl-Parameter* | *Elektronen* | *Protonen* |
|---|---|---|
| Strahlenergie | 26,7 GeV | 820 GeV |
| Mittlerer Strahlstrom | 13 mA | 13 mA |
| Horizontale Strahlgröße | 0,26 mm | 0,29 mm |
| Vertikale Strahlgröße | 0,07 mm | 0,07 mm |
| Strahllänge | 8 mm | 110 mm |

Tabelle 2.1: *Daten des HERA-Speicherrings.*

### Das Target

Für die Erzeugung der B-Mesonen verwendet HERA-B ein ruhendes, festes Target, daß in den Protonenstrahl des HERA-Beschleunigers gebracht wird. Es muß die benötigte Wechselwirkungsrate von bis zu 40 MHz zur Verfügung stellen, ohne den Protonenstrahl des Rings und damit die anderen Experimente zu stören. Um dieses Ziel zu erreichen ist das Target aus Drähten aufgebaut, die an den Rand, in das sogenannte Halo, des Protonenstrahls herangeführt werden. Die hochenergetischen Protonen des Strahls treffen auf das Targetmaterial und produzieren dabei durch Proton-Nukleon Wechselwirkung neue Teilchen; in seltenen Fällen auch Paare von b- und Anti b-Quarks. Die anderen Experimente werden durch das HERA-B Target nicht beeinträchtigt, da die Halo-Protonen nicht zu deren Luminosität beitragen. Sie wären dem Strahl ohnehin in Kürze verlorengegangen.

Das geplante Drahttarget besteht aus insgesamt acht Drähten (Abbildung 2.5), von denen jeweils vier so angeordnet sind, daß sie den Strahl vollständig umschließen.

Die Positionen der acht Drähte zum Strahl werden automatisch nachgeführt, so daß die Wechselwirkungen gleichmäßig auf die Drähte verteilt sind und insgesamt eine Wechselwirkungsrate von 40 MHz erreicht wird.(Also im Mittel vier Proton-Target Wechselwirkungen pro Bunch.) Damit erreicht man bei 820 GeV etwa eine $b\bar{b}$-Rate von 40 Hz. In 80% der $b\bar{b}$-Ereignisse entsteht ein $B^0$ ($\bar{B}^0$) Meson.

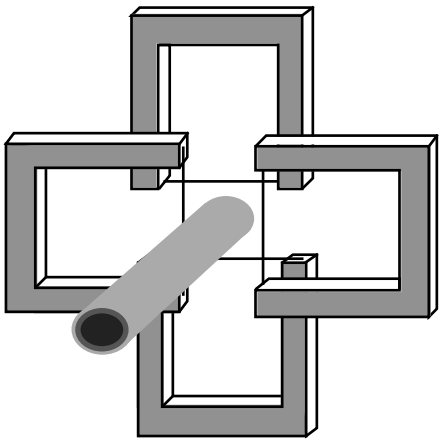

Abbildung 2.5: *Das HERA-B Drahttarget. Die Drähte werden in den Randbereich des Protonenstrahl hineingefahren.*

## 2.3 Teilchennachweis mit dem HERA-B Detektor

Nach einer Strahl-Target Reaktion bewegt sich der Schwerpunkt der Reaktionsprodukte weiter in Strahlrichtung. Der HERA-B Detektor ist daher als magnetisches Vorwärtsspektrometer konzipiert, das einen kegelförmigen Raumbereich abdeckt in den die Reaktionsprodukte hineinstreuen.

Der Teilchendetektor muß mehrere Parameter der Reaktionsprodukte messen können:

1. Spurrekonstruktion
2. Energiemessung
3. Teilchenidentifikation

Für die verschiedenen Aufgaben werden jeweils spezialisierte Detektoren benötigt. Der Spurrekonstruktion dienende Detektoren bezeichnet man als Tracker, energiemessende als Kalorimeter. Für die Teilchenidentifikation können Informationen aus allen Detektoren herangezogen werden. Das HERA-B Spektrometer ist ein Hybriddetektor mit verschiedenen Typen von Spurdetektoren, einem Kalorimeter und Detektoren für die Teilchenidentifikation.

Im folgenden wird anhand von Abbildung 2.6 ein Überblick über den Aufbau des Detektors gegeben. Dabei werden jeweils vom Target aus beginnend zuerst die Komponenten zur Spurrekonstruktion und dann die Komponenten für die Teilchenidentifikation beschrieben (wobei es natürlich Überschneidungen gibt).

### 2.3.1 Spurrekonstruktion

Zur Rekonstruktion der Teilchenspuren besitzt der Detektor ein mehrkomponentiges Spurkammersystem (Abbildung 2.6):

**Der Vertexdetektor** aus Silizium-Streifen befindet sich direkt hinter dem Target. Er besteht aus 28 zu 7 „Superlagen“ gruppierten Siliziumlagen in einem Bereich von 5 cm bis 170 cm Abstand vom Target. Während des Betriebs werden die Lagen bis auf einen Zentimeter an den Strahl herangefahren. Der Vertexdetektor besitzt 136000 Kanäle. Er ermöglicht die Messung der Sekundärvertices der B-Zerfälle mit einer Genaugkeit von etwa 20 $\mu$m senkrecht zur Strahlrichtung und 500 $\mu$m entlang der Strahlrichtung. Nach einem Jahr müssen die Siliziumlagen aufgrund von Alterungserscheinungen, die durch die hohe Strahlenbelastung in Strahlnähe bedingt sind, ausgetauscht werden.

An den Vertexdetektor schließt sich eine Folge aus fünfzehn Superlagen von Spurdetektoren an, die sich über die nächsten ca. 11 m erstrecken. Die ersten neun Lagen befinden sich im Bereich des Magnetfeldes. Da sich die Spurdichte etwa umgekehrt proportional zum Abstand von der Strahlachse verhält, besitzen die Detektoren in Strahlnähe eine feinere Auflösung. Die Lagen sind aus diesem Grunde in ein äußeres- und ein inneres Spurkammersystem eingeteilt.

**Das äußere Spurkammersystem** (outer tracker) reicht bis etwa 20 cm an das Strahlrohr heran. Es wird aus Honeycomb-Kammern, wabenförmigen Einzeldrahtdriftkammern mit 5 oder 10 mm Zelldurchmesser aufgebaut. Wieder sind jeweils drei Lagen um einen kleinen Winkel verdreht zu Superlagen zusammengefaßt. Jede Superlage bestehen ihrerseits aus drei Doppellagen von Röhren. Jede Doppellage enthält 2 Ebenen von Röhren die um einen halben Röhrendurchmesser parallelverschoben sind. Dadurch erreicht man eine hohe Nachweiseffizienz.

**Das innere Spurkammersystem** (inner tracker) besteht aus Gas Mikrostreifen Detektoren („MSGC“). Dabei handelt es sich im Prinzip um Vieldrahtproportionalkammern, allerdings mit etwa zehnmal kleineren Abmessungen. Die Anoden und Kathoden werden als nahe beieinander liegende Streifen auf ein Glassubstrat aufgedampft.

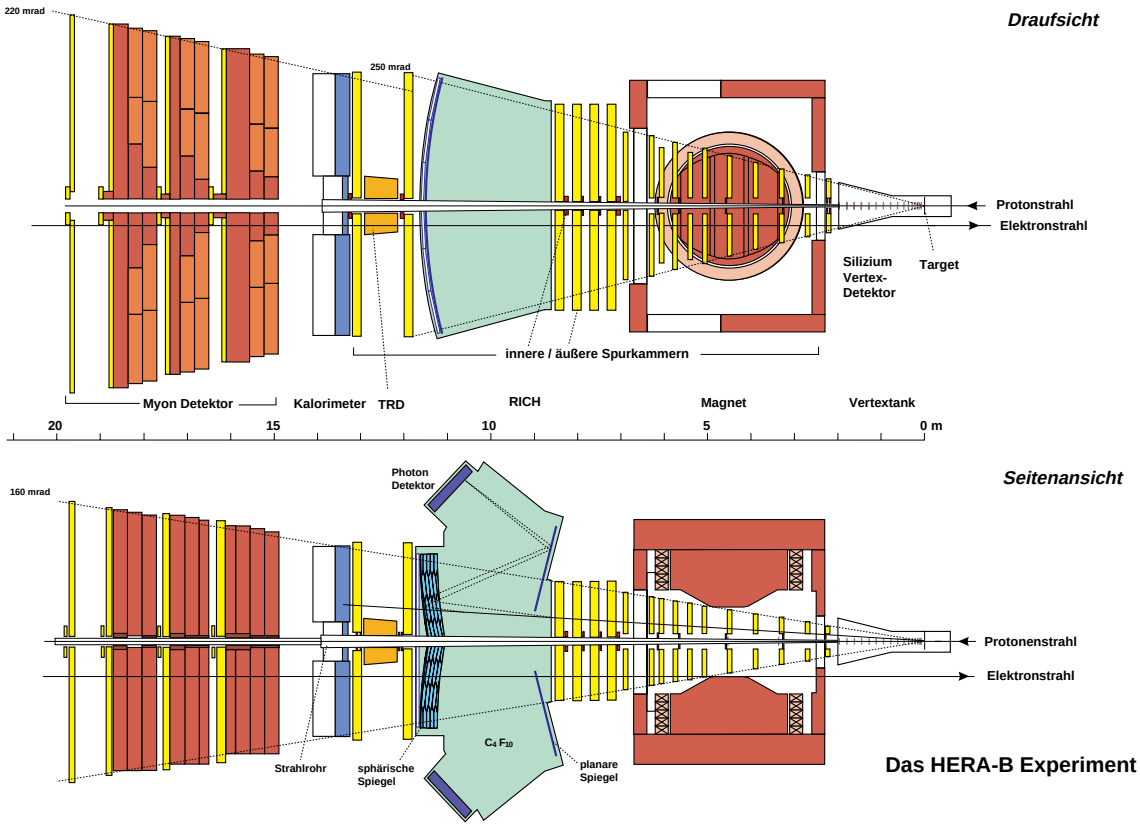


Abbildung 2.6: *Der HERA-B Detektor.*

**Das Myon-System** ist konventionell aufgebaut und nimmt die letzten fünf Meter am Ende des Detektors ein. Es besteht aus vier Myon Detektor Superlagen, vor den ersten drei Lagen befindet sich jeweils ein Myonfilter aus 1 m dicken Eisen-Absorberblöcken. Die Detektoren bestehen nahe der Strahlachse aus Gaspixelkammern, weiter außen aus Drahtkammerröhren. Die ersten beiden Superlagen des Myon-System bestehen wiederum aus drei zueinander verdrehten Einzellagen. Die letzten beiden Superlagen bestehen nur noch aus jeweils einer einzigen Lage mit vertikalen Drähten.

### 2.3.2 Teilchenidentifikation

**Der RICH** ist eine spezielle Form von Cherenkovzähler. Die Cherenkov-Photonen werden mit Photomultipliern nachgewiesen. Die Aufgabe des RICH ist die Identifizierung von Kaonen, im wesentlichen deren Unterscheidung von Pionen. Die Kaonen werden benötigt um die b-Quantenzahl des zweiten, assoziiert produzierten $b$-Quarks zu bestimmen („tagging Kaon“, siehe auch Abbildung 2.2).

**Das elektromagnetische Kalorimeter** hat in erster Linie die Aufgabe, Elektronen von Hadronen zu unterscheiden. Außerdem wird es für die Erzeugung eines Pretriggers benutzt, der dann die Elektronen Spursuche in der ersten Triggerstufe initiiert. Das Kalorimeter ist in abwechselnder Reihenfolge [4] aus Schichten von Absorber- und Szintillatormaterial aufgebaut. Anhand der Reichweite eines Teilchens in dem Kalorimeter wird seine Energie bestimmt. Das dabei entstehende Szintillationslicht wird über Szintillationsfasern auf Photovervielfachern gesammelt und verstärkt.

**Der Übergangsstrahlungsdetektor** ist zwischen dem ECAL und den beiden letzten Spurkammern untergebracht. Er unterstützt im inneren Detektorbereich mit der hohen Spurdichte das Kalorimeter in der Elektron-Hadron Trennung.

**Das Myonsystem** dient neben der Rekonstruktion der Myonspuren auch der Identifikation der Myonen selbst. Die Absorberblöcke können nur von Myonen durchdrungen werden. Für alle anderen Teilchensorten ist der Myonfilter praktisch undurchdringlich. Aus den letzten beiden Myon-Detektor Lagen wird zudem ein Pretrigger Signal abgeleitet, das beim First Level Trigger die Suche nach Myon-Spuren auslöst.

Eine besondere Herausforderung an den Detektorbau bei HERA-B stellt die hohe Strahlenbelastung der Komponenten dar. Sie stellt hohe Anforderungen an die Strahlenhärte der Detektorkomponenten, was insbesondere bei Inner- und Outer-Tracker bereits zu erheblichen Problemen und Verzögerungen geführt hat.

## 2.4 Die Datennahme- und Triggersysteme

HERA-B zeichnet sich, typisch für Hadron Kollisions-Experimente, durch eine große Zahl von über 600000 Meßwerten pro Ereignis (sogenannten Kanälen) und eine hohe Ereignissrate von 10 MHz aus. Dabei wechselwirken pro Ereignis im Mittel 4 Protonen mit dem Target und es entstehen 200 Teilchen. Die Aufgabe des Trigger besteht darin, nach vorgegebenen Triggerbedingungen aus diesen Ereignissen die physikalisch interessanten auszuwählen, damit sie auf Datenträgern abgespeichert werden können. Die Aufzeichnung der Daten kann aber lediglich mit etwa 100 Hz erfolgen. Von dem Triggersystem wird daher eine Unterdrückung von $10^5$ gefordert. Die Datenrate des Detektors von 10 Terabyte/s wird dadurch auf eine für die Speicherung auf Datenträgern geeignete Rate von unter 10 MByte/s reduziert.

Gleichzeitig sind die physikalisch interessanten Ereignisse aber sehr selten - nur in einer von $10^6$ Reaktionen entsteht ein $b\bar{b}$-Paar - und von einem hohen Untergrund überlagert. Sie müssen deswegen von dem Triggersystem trotz der enormen Un-

[4] Shashlik-Bauweise

terdrückung mit hoher Effizienz nachgewiesen werden, damit das Experiment in vertretbarer Zeit zu Resultaten kommt.

Aus den Anforderungen des Experiments ergibt sich die Notwendigkeit eines extrem leistungsfähigen Datennahme- und Triggersytems.

Im Idealfall würde man einen handelsüblichen Mikroprozessor in einer Hochsprache programmieren, ihn ein komplettes Ereignis rekonstruieren und anhand des Ergebnisses die Triggerentscheidung fällen lassen. Ein solcher Rechner würde etwa 2 Sekunden für die Analyse eines HERA-B Ereignisses benötigen. Klarerweise kann damit die erforderliche Rechenleistung nicht zur Verfügung gestellt werden - man bräuchte 20 Millionen Prozessoren und müßte einen 20 Terabyte tiefen Speicher bereitstellen, der die Messungen puffert bis die Entscheidung erfolgt. Hinzu kommt die Datenrate von 10 Terabyte/s mit der kein konventionelles System zurechtkommt.

Um die Anforderungen innerhalb der technischen und finanziellen Rahmenbedingungen trotzdem zu bewältigen, ist das Triggersystem und die Datennahme von HERA-B in vier Stufen eingeteilt. Dabei wird die insgesamt erforderliche Rechenleistung und der Speicherbedarf reduziert, indem man auf den unteren Stufen mit einfachen Algorithmen und möglichst schnell bereits einen Großteil der Ereignisse verwirft (Abbildung 2.7). Jede Triggerstufe untersucht nur solche Ereignisse, die die jeweils vorherige Stufe passiert haben. Die zugeordnete Stufe des Datennahmesystems ist dabei für die Zwischenspeicherung der gefilterten Ereignisse und ihre Bereitstellung für die Triggeranalyse zuständig.

| Stufe | Anzahl Kanäle | Eingangs-rate | Algorithmus | Latenzzeit | | Reduktion |
|---|---|---|---|---|---|---|
| | | | | | 10 MHz | |
| 1 | 100 k | 150 GB/s | Mustererkennung Spuren u. J/ψ Rekonstruktion | 12 µs | | 200 |
| | | | | | 50 kHz | |
| 2 | 400 k | 1,8 GB/s | Spurfit, mit Magnet u. Vertex Teilchen ID | 5-50 ms | | 25 |
| | | | | | 2 kHz | |
| 3 | 600 k | 140 MB/s | Ereignisdaten zusammen-fügen, Globaler Fit | 100 ms | | 20 |
| | | | | | 100 Hz | |
| 4 | 600 k | 7 MB/s | komplette Rekonstruktion | 2s | | - |

Abbildung 2.7: *Die Betriebsparameter der einzelnen Stufen des Datennahme- und Triggersystems.*

Abbildung 2.7 zeigt wie die Ereignisrate nach oben hin abnimmt, während die Latenzzeit zunimmt. Gleichzeitigt steigt aber die pro Ereignis betrachtete Datenmenge

und die Komplexität der Triggeralgorithmen, so daß der Aufwand in den einzelnen Stufen etwa vergleichbar ist.

Die generelle Struktur des HERA-B Trigger- und Datennahmesystem zeigt Abbildung 2.8 (siehe auch [59] [41] [32] ). Die Dicken grauen Pfeile zeigen den Weg der Ereignisdaten an, vom Detektor über die Ausleseelektronik mit ihren Treibern, durch die Pufferspeicher der einzelnen Stufen, bis die von allen Triggern akzeptierten Daten dann auf Band geschrieben werden.

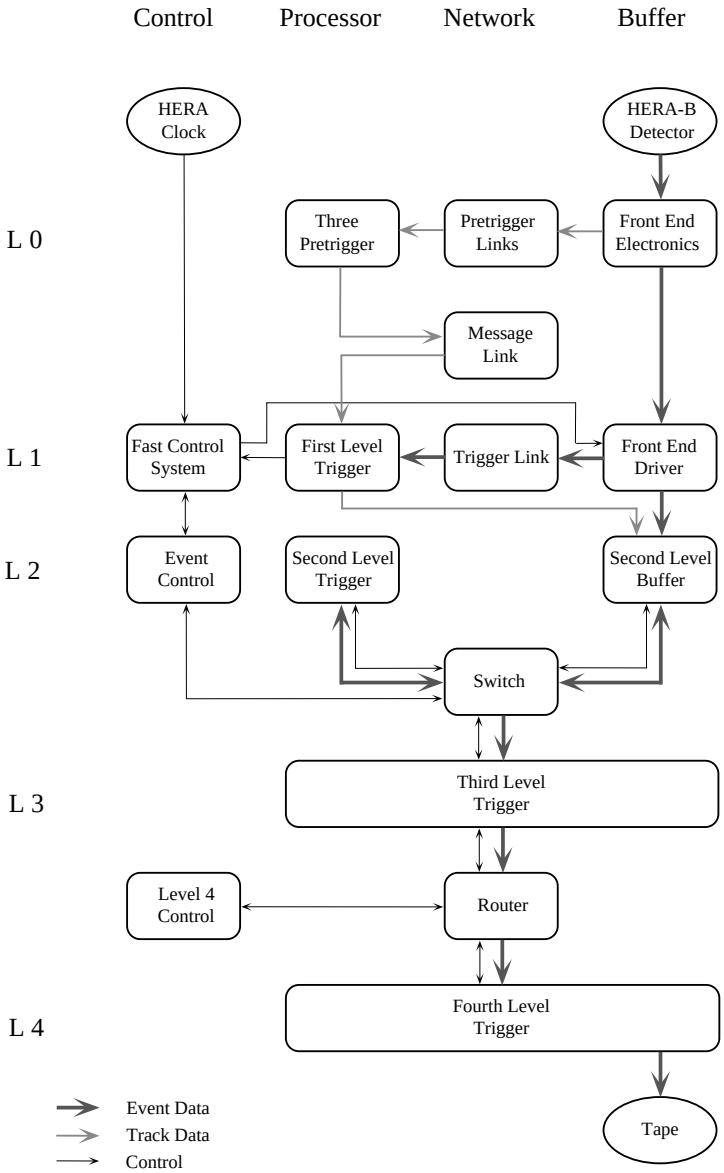


Abbildung 2.8: *Schematischer Aufbau der Datennahme- und Triggersysteme mit den Datenflüssen zwischen den einzelnen Komponenten.*

Die Front End Driver (FED) enthalten die Pufferspeicher für den FLT. Hier werden die letzten 128 Ereignisse zwischengespeichert bis die Entscheidung des FLT gefallen ist. Ein Teil der Ereignisdaten wird gleichzeitig als Kopie von den FED an den FLT geschickt, der darin nach Spuren sucht, um seine Triggerentscheidung zu fällen. Der Weg der Spurinformationen ist mit schmalen grauen Pfeilen eingezeichnet. Er beginnt bei den Pretriggern geht über den FLT bis in den Second Level Buffer (SLB), wo die Spurdaten, die der FLT errechnet hat, wieder mit den eigentlichen Meßdaten zusammengebracht werden. Details hierzu sind im folgenden Kapitel beschrieben.

Der Second Level Buffer empfängt und speichert alle von dem FLT akzeptierten Ereignisse. Er ist mit etwa 1000 Digitalen Signal Prozessoren [5] (DSP) aufgebaut. Neben der reinen Speicherfunktion sind die DSPs auch für das Puffer Management und die Auswahl spezifischer Datensegmente, die von dem SLT angefordert werden zuständig. Die DSP Links bilden ein sogenanntes „Switching Network“ über das die Kommunikation zwischen SLB, SLT und der dritten Triggerstufe erfolgt. HERA-B ist das erste Experiment, das in seinem Datennahmesystem ein solches Netzwerk einsetzt. Frühere Systeme waren datengesteuert ausgelegt, d.h. die gesamten potentiell benötigten Daten eines Ereignisses werden an die Prozeßeinheit geschickt. Das HERA-B System dagegen ist anfragegesteuert, d.h. ein Prozessor erhält auf Anfrage genau die Daten, die er für seine Berechnungen bei einem bestimmten Ereignis benötigt. Dies erhöht zwar den Steuerungsaufwand und die Latenzzeit, aber es minimiert die Datenmenge, die aus dem Puffer ausgelesen werden muß.

Die zweite Triggerstufe selbst besteht aus einer Rechnerfarm mit 200 Pentium Rechnern, die unter Linux betrieben werden. Die Rechner sind über ihren PCI-Bus mit dem Switching Network verbunden.

Nach der zweiten Stufe werden die Ereignisse in den Speicher eines Prozessorknotens der dritten Stufe bewegt. Die Dritte Stufe ist wiederum eine Farm aus ca. 200 Pentium Rechnern. Hier werden erstmals alle Daten eines Ereignisses, die sich in verschiedenen SLB-Modulen befinden, zusammengebracht (Event building). Mit den kombinierten Ereignisdaten werden dann weitere Filterschritte angewendet. Die dritte und vierte Triggerstufe sind über ein normales Fast Ethernet Netzwerk verbunden. Die vierte Stufe besteht aus weiteren ca. 200 Pentium Rechnern, die dann die vollständige Rekonstruktion der Ereignisse vornehmen.

Das Schnelle Kontroll System (Fast Control System, FCS) ist für die Übertragung zeitkritischer Kontrollinformationen zuständig. Es synchronisiert beispielsweise zahlreiche Einheiten der Ausleseelektronik und der schnellen Triggerelektronik mit der Frequenz des Speicherrings. Auch Trigger des FLT werden über das FCS an die Auslesesysteme verteilt, um die Weitergabe der Ereignisse von Front End Driver an den Second Level Buffer zu veranlassen. Das FCS ist aus einem Mutter-Modul und vie-

[5] Typ ADSP21060 „SHARC“ von Analog Devices mit 6 Links a 40 MByte/s, 4 Mbit Speicher und 80 MIPS.

len Tochter-Modulen aufgebaut. Sie sind über Glasfasern verbunden und verwenden G-Link Chips als serielle Schnittstellen für den Datenaustausch.

# Kapitel 3

# Der First Level Trigger für HERA-B

Der First Level Trigger soll in der Lage sein, effizient und mit einer hohen Unterdrückung die gesuchten Ereignisse aus den Untergrundereignissen herauszufiltern. Nach der Darstellung des verwendeten Triggeralgorithmus, mit dem die Anforderungen erfüllt werden können, folgt die Beschreibung des Multiprozessorsystems, das den Algorithmus anwendet. Es wird ein Überblick über den Aufbau und die Datenflüsse des Gesamtsystems und die Kontrollsysteme, die seine Umgebung bilden, gegeben. Darauf folgt eine allgemeine Beschreibung der Prozessor-Boards mit ihren Kommunikations-Schnittstellen. Anschließend wird die Funktionsweise der diskret aufgebauten Prozessor-Einheiten auf den verschiedenen Boards beschrieben. Das Kapitel schließt mit der Einordnung des eigenen Ansatzes für den FLT in den Stand der Forschung.

## 3.1 Anforderungen an den First Level Trigger

### 3.1.1 Die physikalischen Anforderungen

Der HERA-B First Level Trigger soll auf Ereignisse mit Lepton-Paaren, die aus $J/\psi$-Zerfällen stammen und auf Lepton-Hadron- bzw. Hadron-Hadron Ereignisse mit hohem Transversalimpuls triggern können.

Simulationen haben gezeigt, daß man etwa $10^3$ vollständig rekonstruierte $B_0$-Zerfälle benötigt, um eine CP-Verletzung mit ausreichender Genauigkeit zu messen (oder zu widerlegen) [16]. Weitere Randbedingungen sind die begrenzte Laufzeit des Experiments von ungefähr 5 Jahren, die Effizienz des Gesamtdetektors und die Zerfallswahrscheinlichkeiten der interessanten Zerfälle.

Die $b\bar{b}$-Paare werden nur mit einer Rate von $10^{-6}$ erzeugt. In etwa 80 Prozent dieser Ereignisse entsteht daraus ein $B_0(\bar{B}^0)$-Meson. Als effiziente Trigger Signatur des

FLT für den goldenen Zerfall der neutralen B-Mesonen (Abbildung 2.1) bietet sich der Sekundärzerfall $J/\psi \to l^+l^-$ an (vergleiche Abbildung 2.2). Dabei triggert man auf Leptonpaare mit einer invarianten Masse im Massenbereich des $J/\psi$-Mesons. Das Verzweigungsverhältnis für diesen Kanal beträgt aber nur etwa $4 \cdot 10^{-6}$.

Daher läßt sich die erforderliche Ereigniszahl nur mit Wechselwirkungsraten von einigen 10 MHz bei gleichzeitig effizientem Nachweis der gesuchten Ereignisse erreichen. HERA-B arbeitet aus diesem Grund mit einer Wechselwirkungsrate von 40 MHz, also 4 Wechselwirkungen pro Ereignis, womit eine B-Rate von 30 Hz erzeugt wird. Hinzu kommt der enorme Untergrund, aus dem die B-Zerfälle herausgefiltert werden müssen. Insgesamt wird also ein sehr effizientes und hoch selektives Trigger-System benötigt. Daraus ergeben sich folgende Basisanforderungen an den First Level Trigger:

- eine Reduktion der Ereignisrate um den Faktor 200
- hohe Nachweiseffizienz (60 - 80 Prozent) für Leptonpaare aus $J/\psi$ Zerfällen

Große CP-Asymmetrien werden auch für Zerfälle wie $B^0 \to \pi^+\pi^-$ erwartet. Mit dem high-$p_t$ Pretrigger kann auf diesen hadronischen Zerfall getriggert werden, man erwartet etwa 900 Ereignisse pro Jahr. In Kombination von Lepton- und Hadron-Triggern kann auch auf Zerfälle wie zum Beispiel $\bar{B}^0 \to D^+\pi^-$ oder $B^0 \to D^-3\pi^\pm$ getriggert werden.

### 3.1.2 Die technischen Anforderungen

Da die Puffer der ersten Stufe des Datennahme-Systems eine Tiefe von 128 Ereignissen besitzt, darf die Latenzzeit der ersten Triggerstufe maximal 12,3 $\mu$s betragen. Abzüglich der Zeiten, die die Pretriggersysteme und das Fast Control System beanspruchen, verbleiben dem FLT hiervon nur etwa 9 $\mu$s [13]. Innerhalb dieser Zeit muß der FLT seine Entscheidung treffen und dazu folgende Aufgaben erfüllen:

- Rekonstruktion von Teilchenspuren.

  Aus den im Mittel 200 Spuren pro Ereignis diejenigen erkennen, die von den gesuchten B-Zerfällen stammen.

- Berechnung und Überprüfung programmierbarer Spurparameter.

  Aus den Bahndaten der rekonstruierten Spuren mehrere Parameter berechnen und auf Eigenschaften der gesuchten Teilchen überprüfen.

- Berechnung der invarianten Masse aller Spurpaare und Triggerentscheidung.

  Die Einzelspuren paarweise kombinieren, ihre invariante Masse berechnen und daraufhin entscheiden, ob sie aus einem $J/\psi$ Zerfall stammen können.

Um die gesuchten Spuren rekonstruieren zu können, muß der FLT die Daten aus etwa 100000 Detektorkanälen des Spurkammersystems heranziehen. Dabei ist jeweils ein Bit pro Kanal erforderlich. Mit der Ereignisrate von 10 MHz ergibt dies eine Eingangsdatenrate von etwa einem Terabit pro Sekunde für den FLT. Es wird also ein aufwendiges System für die Datenübertragung über 60 Meter vom Detektor zu der Triggerelektronik mit einer Bandbreite von 1 Terabit/s benötigt.

Die genannten Anforderungen können nur durch massive Parallelisierung und Pipelining erreicht werden. Als erstes muß daher ein Algorithmus gefunden werden, der eine Parallelisierung des Problems erlaubt. Dieser wird in Kapitel 3.2 beschrieben.

Die benötigte Rechenleistung, um einen Algorithmus mit den geforderten Randbedingungen anzuwenden, läßt sich nur mit Spezialhardware erfüllen, die an die Aufgabenstellung angepaßt ist. Gewöhnliche Mikroprozessor-basierte Rechner sind um mehrere Größenordnungen zu langsam. Deshalb wird der FLT als Multiprozessorsystem mit etwa 80 parallel arbeitenden Spezialprozessoren aufgebaut.

## 3.2 Der Triggeralgorithmus

Für die Spurensuche des FLT wird ein Algorithmus in Anlehnung an den Kalman Filter Algorithmus [24, 25] verwendet, mit dem die Spur vom Ende des Detektors her in Richtung Target rekonstruiert wird. Ausgelöst wird die Spurensuche grundsätzlich von einem der drei Pretrigger Systeme. Jedes der drei Systeme ist sensitiv auf bestimmte Teilchensorten, der Elektron-Pretrigger, der Myon-Pretrigger und der Hadron-Pretrigger. Der Algorithmus vereinfacht sich dadurch, daß der FLT nur außerhalb des Magnetfeldes nach Spuren sucht, er muß also nur gerade Spuren rekonstruieren. Dabei gilt es einen Kompromiß zu finden zwischen einer hohen Prozeßgeschwindigkeit und dem Berechnen und Aktualisieren der vollen Spurinformationen. Letzteres ermöglicht das frühe Erkennen ungeeigneter Spuren und liefert sofort einen kompletten Spurfit, erhöht aber den Hardware-Aufwand, den Informationstransfer und damit die Rechenzeit. Um eine hohe Geschwindigkeit zu erreichen, wird ein vereinfachter Algorithmus mit minimalem Informationstransfer verwendet. Anhand von Bild 3.1 wird die Spurrekonstruktion am Beispiel eines Elektron-Pretriggers erläutert. Die Elektronspur wird vom Kalorimeter ausgehend durch vier Detektor-Superlagen in Richtung Target rekonstruiert.

Aus der Richtung des Targets kommend passiert ein Teilchen die vier Kammern, die für die Spurensuche des FLT im vorderen Teil des Detektors verwendet werden. Der Kalorimeter Pretrigger ist sensitiv auf Elektron-Kandidaten. Wenn er anspricht, wird aus der Ortsinformation des Kalorimeters und der Tatsache, daß die gesuchten Elektronen aus dem Bereich des Targets kommen müssen, ein Suchbereich für die vor dem Kalorimeter liegende Detektorlage berechnet. Diese Informationen werden an die vorherige Lage übermittelt. Dort wird dann der Suchbereich nach einem

Teilchensignal abgesucht. Wird ein Teilchendurchgang entdeckt, so errechnet sich aus den bereits vorhandenen Spurinformationen und dem neuen lokalen Spurpunkt wiederum ein Suchbereich für die davorliegende Detektorlage. Diese Informationen über die Spur und den neuen Suchbereich werden an die vorige Lage weitergegeben. Dort wird dann nach dem nächsten Spurpunkt gesucht. Diese Prozedur wird solange fortgesetzt, bis man in der vordersten Detektorlage angekommen ist. Die Suchbereiche werden dabei zum Target hin immer kleiner, da immer genauere Kenntnis über den Spurverlauf vorliegt. Wird in einem Suchbereich kein Teilchendurchgang festgestellt, so wird die Spur verworfen und nicht weiterverfolgt.

Im Prinzip erhält der FLT in diesem Beispiel vier Schwarzweißbilder von dem Detektor. Ausgehend von dem Pretrigger Spurpunkt als Wurzel, wird dann eine Breitensuche nach Spurpunkten in den Bildern gestartet. Mit diesem Prinzip kann die Rechenleistung auf parallel arbeitende Einheiten verteilt werden, die nur lokal mit den Daten eines ihnen zugeordneten Bereichs des Detektors arbeiten und untereinander Spurinformationen austauschen.

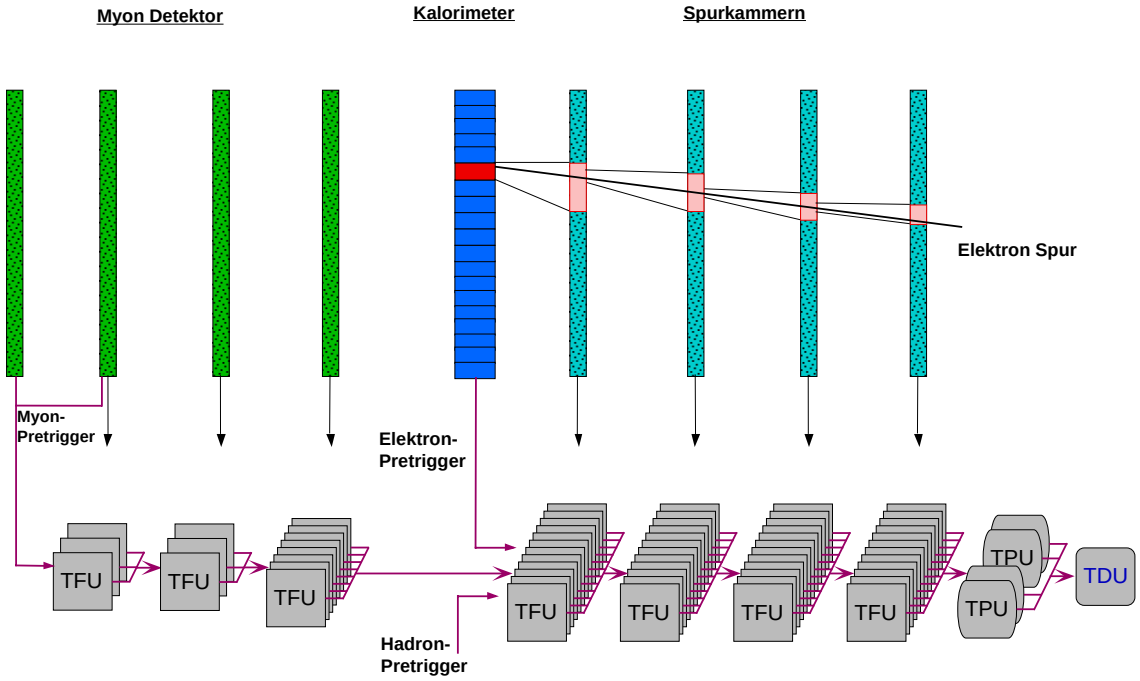


Abbildung 3.1: *Die Rekonstruktion von Teilchenspuren. Ausgehend von einem Pretrigger-Signal wird eine Spur von ihrem Ende durch das Spurkammersystem in Richtung Target zurückverfolgt.*

Für die Spurensuche wird ein spezieller Hardware Prozessor gebaut. Im Endausbau enthält das FLT Multiprozessor-System ca. 75 solcher Track Finding Units (TFU) genannten Prozessoren. Jede TFU wird einem bestimmten Detektorbereich zugeordnet. Die Meßdaten aus diesem Bereich werden über 24 optische Hochgeschwindigkeits-Datenleitungen übertragen. Jede TFU arbeitet nur mit den lokalen Daten

ihres Bereichs. Jeder Detektor Superlage ist also sozusagen eine „Lage“ von TFUs zugeordnet, die deren Spurdaten verarbeiten. Die TFUs sind über ein internes unidirektionales Bussystem untereinander und mit den Pretriggersystemen verbunden. Dabei empfängt jede TFU sog. Standard Messages von der vorherigen Lage und sendet Messages an TFUs in der nächsten Lage. Die erste Lage empfängt Messages von einem Pretriggersystem, die letzte Lage schickt die Daten der komplett rekonstruierten Spur an einen Prozessor, der auf die Überprüfung von Spurparametern spezialisiert ist.

Von diesem zweiten Prozessortyp, der Track Parameter Unit (TPU), werden vier Exemplare benötigt. Sie berechnet als erstes die kinematischen Parameter jeder Spur und wendet Schnitte darauf an. Anschließend überprüft sie, ob eine Spur mehrfach rekonstruiert wurde und verwirft dann die überzähligen Exemplare. Die Spuren, die diesen Filterprozeß passieren, werden dann als Messages an einen dritten Prozessortyp geschickt.

Von dieser Trigger Decision Unit (TDU), die die eigentliche Triggerentscheidung fällt, werden ein oder zwei Exemplare benötigt. Die TDU sammelt alle Spuren eines Ereignisses und berechnet die Masse aller möglichen Kombinationen von Spurpaaren. Anschließend wird aus diesen Ergebnissen und den Spurinformationen die Triggerentscheidung abgeleitet. Dabei ist es nicht ausreichend, nur auf die invariante Dileptonenmasse zu triggern. Da man verschiedene Zerfälle untersuchen will, muß die Möglichkeit bestehen, auf verschiedene Ein- und Zweispurparameter zu triggern.

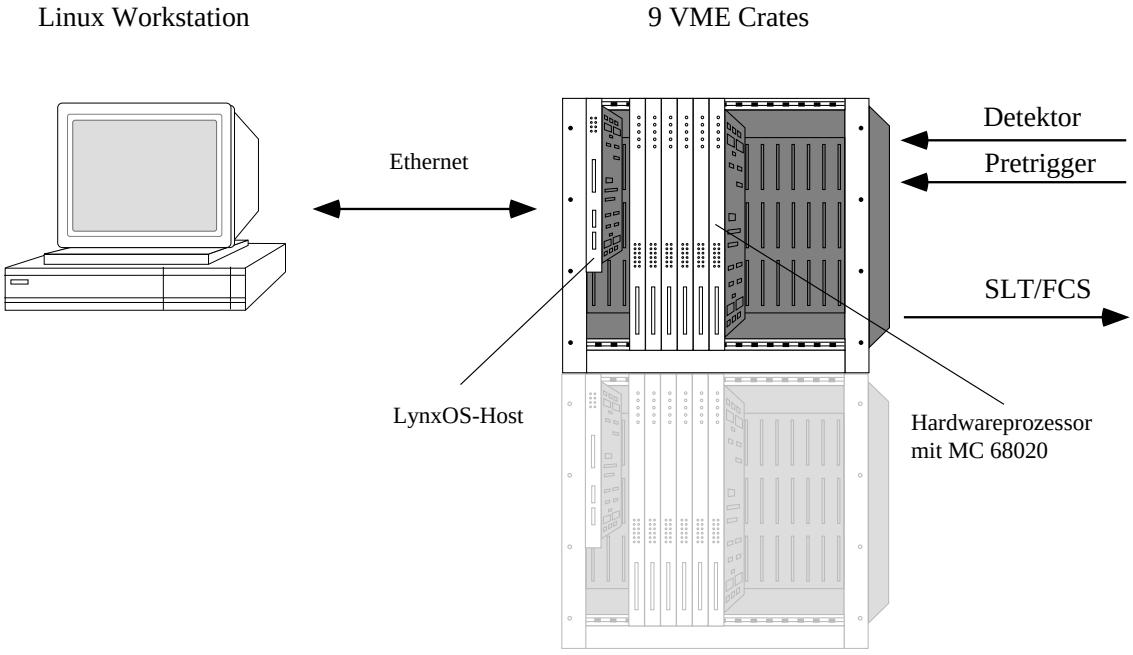


Abbildung 3.2: *Das FLT-Gesamtsystem.*

## 3.3 Das Multiprozessorsystem

In diesem Abschnitt wird ein Überblick über die Hardware des FLT-Gesamtsystems gegeben. Zudem werden die Kommunikations- und Kontrollsysteme, die sich mit den Hardwareprozessoren auf den Boards befinden, beschrieben. Diese Peripherie der eigentlichen Prozessoren verlangt etwa noch einmal den gleichen Aufwand wie die Prozeßeinheiten selbst, die im Anschluß in Abschnitt 3.4 behandelt werden.

Der FLT wird als asynchrones Multiprozessorsystem mit Pipeline- und Parallel-Architektur aufgebaut. Von den drei Typen von Spezialprozessoren kommen voraussichtlich insgesamt etwa 80 Exemplare zum Einsatz: 75 TFUs, 4 TPUs und 1-2 TDUs. Die Prozessorboards sind jeweils als VME-Einschübe ausgeführt und werden in 8 - 9 VME-Crates untergebracht (Abbildung 3.2). Die Crates sind mit einer speziellen Rückwand (Backplane) ausgestattet. Sie entspricht dem VME-Standard mit zusätzlichen Verbindungen zwischen Prozessor-Board und Message-Board, das von hinten in das Crate gesteckt wird (siehe hierzu 3.3.2). Die einzige Abweichung vom VME-Standard ist die Beschränkung auf nur eine Interrupt Leitung auf der Backplane, so daß dem VME-Bus nur eine Interrupt Priorität zur Verfügung steht.

Für die weiteren Betrachtungen zeigt Abbildung 3.3 ein Datenfluß-Diagramm der Datenflüsse innerhalb des FLT-Systems und mit seiner Umgebung. Der Übersichtlichkeit wegen ist jeder Prozessortyp nur einmal aufgeführt. Datenquellen und Senken sind mit Doppelstrichen versehen, datenverarbeitende Prozesse werden durch Kreise und Datenflüsse mit Pfeilen symbolisiert.

Im rechten Teil wird deutlich, daß die eigentliche Datenverarbeitung im FLT vollständig datengesteuert ist. Als Datenquellen fungieren die Pretriggersysteme und der Detektor. Die Detektordaten werden synchron mit dem Beschleunigertakt in die lokalen Datenspeicher der TFUs geschrieben. Dies geschieht unabhängig davon, ob die Daten eines bestimmten Ereignisses für die Spurensuche in der jeweiligen TFU tatsächlich benötigt werden. Die Daten müssen für den Fall, daß eine Message empfangen wird, sofort zur Verfügung stehen. Würde man ereignisspezifisch die benötigten Detektordaten anfordern und dann nur diese übertragen, so würde das Anfordern und Warten bis sie eintreffen viel zu lange dauern. Die Spurdaten werden von den Pretriggern ausgehend unidirektional durch den FLT prozessiert, bis am Ende die Triggerentscheidung an das Fast Control System und die vollständig rekonstruierte Spur an die zweite Triggerstufe (Datensenken) weitergeben wird.

Die große Menge an Daten, die vom Detektor zum Trigger geleitet werden muß, und die Interprozessorkommunikation erfordern zwei unterschiedliche Kommunikationsnetze, die eine hohe Datenrate aufweisen müssen [15].

Die Kontrolle der Prozessoren erfolgt vollständig per Software (linker Teil in Abbildung 3.3). Der Kontrolldatenfluß ist dabei von dem Prozeßdatenfluß völlig unabhängig. Auf jedem der boardeigenen Mikroprozessoren läuft ein Programm, das vollen Zugriff auf die Betriebsparameter des Boards hat und alle Kontrollaufgaben übernimmt. In Fehlersituationen kann der Hardwareprozessor Interrupts auslösen,

die dann in der Interruptroutine des Kontrollprogramms bearbeitet werden. Über den VME-Bus stellt das Kontrollprogramm die Verbindung zu dem VME-Host-Rechner her. Als Host besitzt jedes Crate einen Unix-Rechner, auf dem ein entsprechendes Kommunikationsprogramm läuft.

Der VME-Host wird ohne Massenspeicher betrieben und bootet über das Netzwerk von einer zentralen Linux Workstation. Auch die zentrale Kontrolle des Triggersystems wird von der Workstation aus vorgenommen. Sie dient zudem als Dateiserver für das gesamte System.

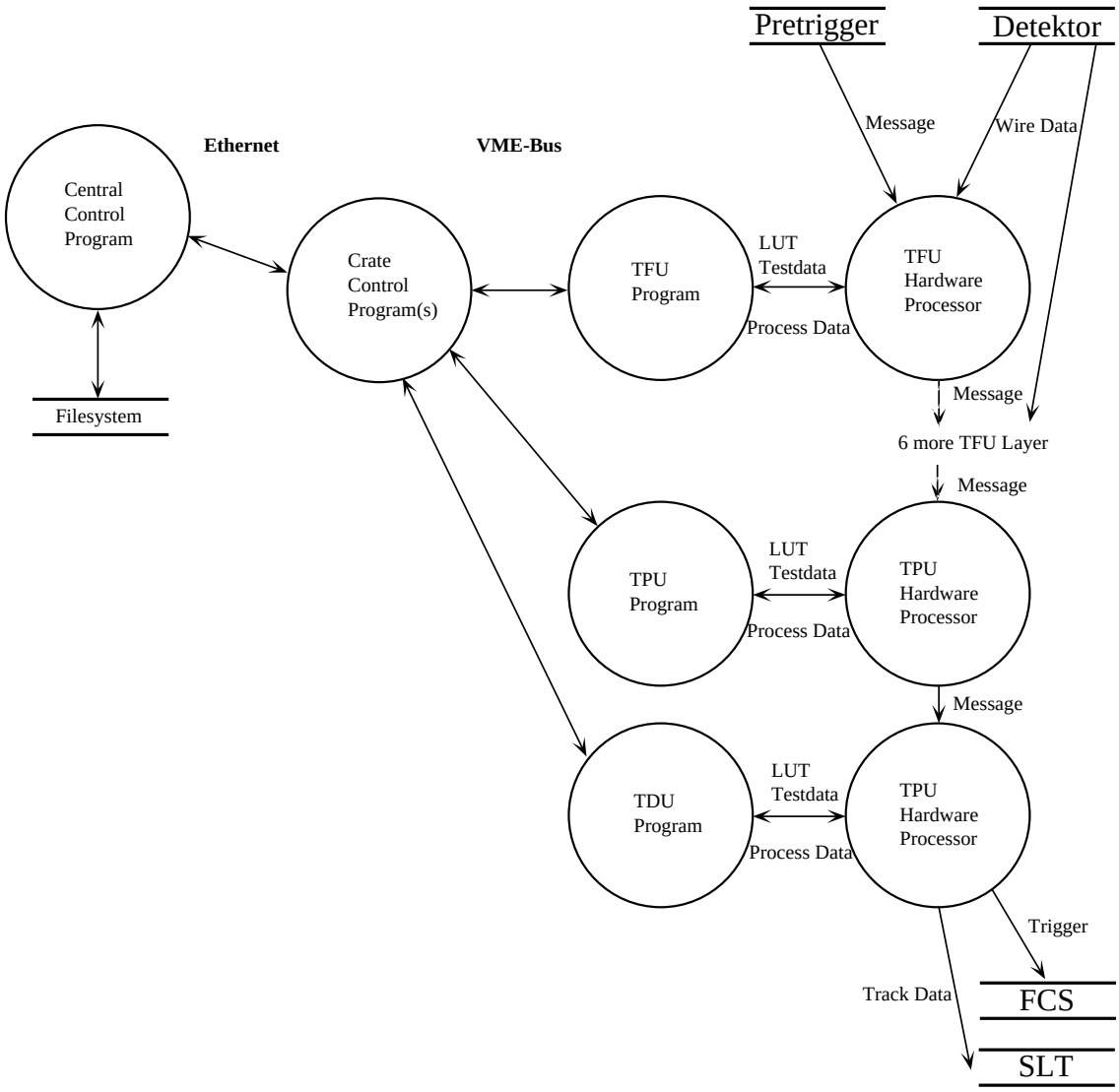


Abbildung 3.3: *Das Datenfluß-Diagram des FLT-Gesamtsystems. Für einen besseren Überblick ist jeder Prozeß nur einmal aufgeführt.*

### 3.3.1 Genereller Aufbau der Prozessor Boards

Die Komponenten des FLT sind als synchrone gepipelinete Spezialprozessoren realisiert. Jeder Prozessor für sich arbeitet asynchron zum Beschleunigertakt und zu allen anderen Prozessoren. Sie sind mit 50 MHz getaktet, so daß ein Prozessor alle 20 ns eine Spur bearbeiten kann, indem er sie in seine Pipeline übernimmt. Im Mittel können also maximal 5 Spuren pro Ereignis pro TFU verarbeitet werden.

Die Leiterplatten, z.B. der TFU, bestehen aus 14 Lagen, davon 10 Signallagen und 4 Lagen mit Masse oder Betriebsspannung. Die Bauteile werden beidseitig in SMD Technologie aufgelötet. Ein Board besitzt ca. 20000 Lötstellen. Die Spezialprozessoren sind hauptsächlich mit programmierbaren Logikbausteinen und SRAMs für Tabellen (Lookup-Tables) aufgebaut.

Soweit möglich wird bei der Umsetzung des Algorithmus von Tabellen Gebrauch gemacht. Damit erreicht man die größte Rechengeschwindigkeit, da die Ergebnisse lediglich aus Tabellen gelesen werden müssen und die Rechenschritte in einem Taktzyklus durchgeführt werden können. Weiterhin erreicht man eine gewisse Flexibilität, die es gestattet, alle TFUs identisch aufzubauen und kleinere nachträgliche Änderungen in dem Algorithmus aufzufangen. Die individuellen Eigenschaften der lokalen Detektorgeometrie werden durch entsprechendes Programmieren der Tabellen berücksichtigt. Bei späteren Änderungen ist dann lediglich ein Umprogrammieren der Tabellen notwendig.

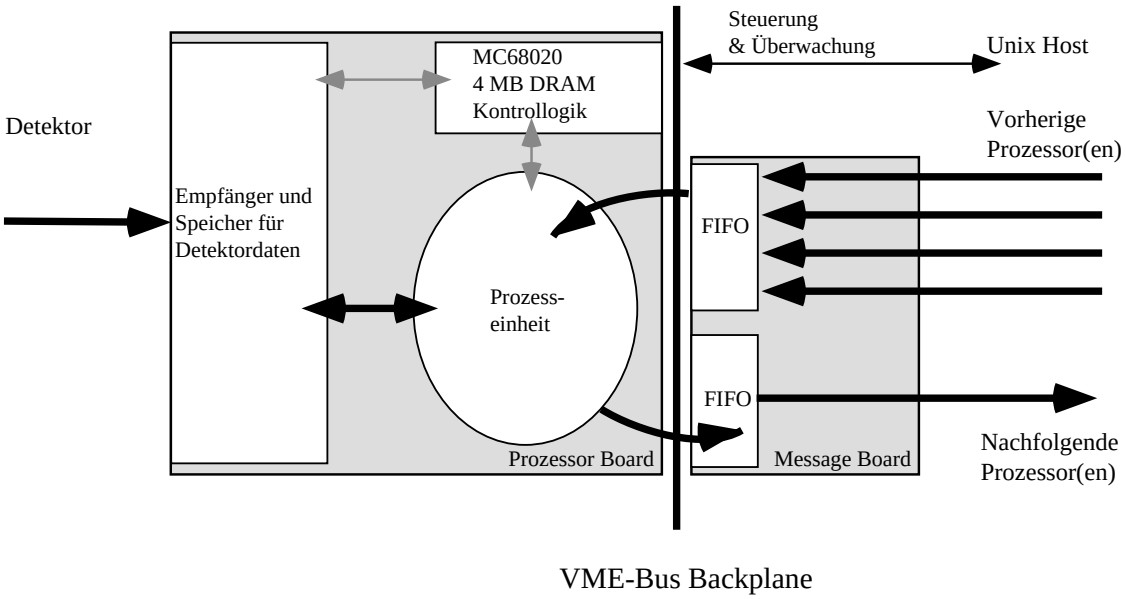


Abbildung 3.4: *Der prinzipielle Aufbau der Prozessor-Boards am Beispiel der TFU. Das Messageboard stellt jedem Prozessor vier Eingangs- und einen Ausgangs-Kanal für Spurdaten zur Verfügung.*

Abbildung 3.4 gibt einen schematischen Überblick der Prozeßelemente auf den Prozessor-Boards und dem Message-Board. Das Beispiel zeigt eine TFU, die Größenverhältnisse und Platzierung der Elementsymbole sind vergleichbar mit den Verhältnissen auf dem realen Board. Die TPU und TDU sind im Prinzip gleich aufgebaut wie die TFU (natürlich mit anderen Prozeßeinheiten), mit dem Unterschied, daß kein Empfänger und Speicher für Detektordaten vorhanden ist.

Das Prozessor- und das Message-Board sind über die Stecker der VME-Bus-Backplane miteinander verbunden. Das Message-Board empfängt von maximal vier Sendern die Spurinformationen. Die Prozesseinheit übernimmt die Spurdaten, greift in Abhängigkeit von ihrem Inhalt auf den Speicher der Detektordaten zu. Die gefundenen Spuren übergibt sie dann, mit verbesserten Spurparametern an das Message-Board, daß sie an die nachfolgenden Prozessoren sendet.

Neben dem eigentlichen Hardware-Prozessor besitzt jedes Board noch einen Mikroprozessor Motorola 68020 mit 4 MB Hauptspeicher, der mit 25 MHz getaktet ist. Dieser zusätzliche On-Board-Rechner wird bei der Inbetriebnahme und zu Testzwecken eingesetzt. Beim Start des Systems berechnet er die Tabellen , die die Prozesseinheit für ihre Berechnungen verwendet. Während des Betriebs dient er der Initialisierung sowie der Steuerung und Überwachung des eigentlichen Prozessors. Er hat auf alle wichtigen Betriebsparameter des Boards Zugriff. Für die Kommunikation mit dem Host Rechner besitzt er eine 32-Bit VME-Schnittstelle.

### 3.3.2 Das Message Transfer System

Die gesamte Interprozessorkommunikation wird über das Message Transfer System abgewickelt. Es hat die Aufgabe, die Nachrichten (Messages), die jeweils eine Spur repräsentieren, von den Pretriggern zu dem FLT und zwischen dessen Prozessoren zu übertragen.

Hierfür wurde eigens ein Message-Board entwickelt, das eine Hochgeschwindigkeits-Schnittstelle mit einem einheitlichen Datenformat zur Verfügung stellt. Die Message-Boards werden auf die Rückseite der VME-Backplane gesteckt. (Abbildung 3.4) Dadurch ist jede Prozessor Einheit über die nicht vom VME-Bus benutzten Stecker direkt mit genau einem Message-Board verbunden.

Der Takt des Message-Boards und des Prozessor-Boards sind dabei über FIFOs entkoppelt (Abbildung 3.5). Dies ist notwendig, weil das Message-Board, im Gegensatz zu den 50 MHz schnellen Prozessoren, mit 100 MHz getaktet ist und weil die Bearbeitungszeit einer Spur (Message) durch die TFU nicht fest ist. Zu der Mindestzeit von 240 ns können zusätzliche Taktzyklen hinzukommen, wenn mehrere Spurpunkte gefunden werden (siehe Abschnitt 3.4.1). Eine Message hat eine Größe von 80 Bit. Zur Übertragung werden die Messages 4:1 gemultiplext und dann mit 100 MHz über ein 50 adriges Flachbandkabel (20 Datenleitungen) gesendet. Dies ergibt eine Datenrate von 240 MByte/s pro Message-Kanal. Die Übertragungsstrecke beträgt innerhalb eines VME-Crates oder zwischen zwei Crates typischerweise 1 bis 2 Me-

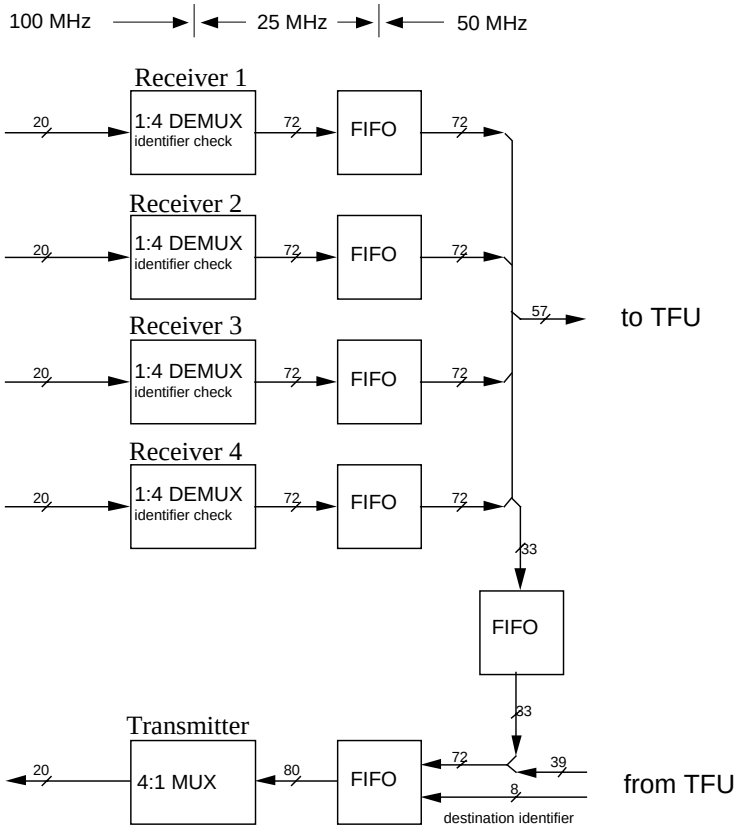


Abbildung 3.5: *Der Aufbau eines Message-Boards.*

ter. Jedes Message-Board besitzt einen Sender und vier Empfänger. Die Empfänger müssen eine Message demultiplexen und schreiben sie dann in ein FIFO. Die TFU liest ihre Eingangsdaten aus den vier Empfänger-FIFOs nach einem gleichberechtigenden Auswahlverfahren (Round Robin) ein. Dabei wird nur der Teil der Message gelesen (57 Bit), der von der TFU benötigt bzw. verändert wird. Der Rest wird in einem FIFO zwischengespeichert. In dem Fall, daß Spuren gefunden werden, wird der FIFO-Inhalt mit den Ausgangsdaten der TFU wieder zu einer Message zusammengestellt und in das Sender FIFO geschrieben.

### 3.3.3 Die optische Datenübertragung

Die Detektordaten der Ereignisse fallen mit der Rate des Beschleunigertakts von 10 MHz an und müssen in den lokalen Speicher der TFU hineingeschrieben werden (Abbildung 3.6).

Das optische Datenübertragungssystem sendet die für die Spurensuche benötigten Detektordaten, etwa 1 TBit/s, über eine Entfernung von 60 Metern an die TFUs. Um hier auf eine handhabbare (und bezahlbare) Anzahl von Leitungen zu kommen, wer-

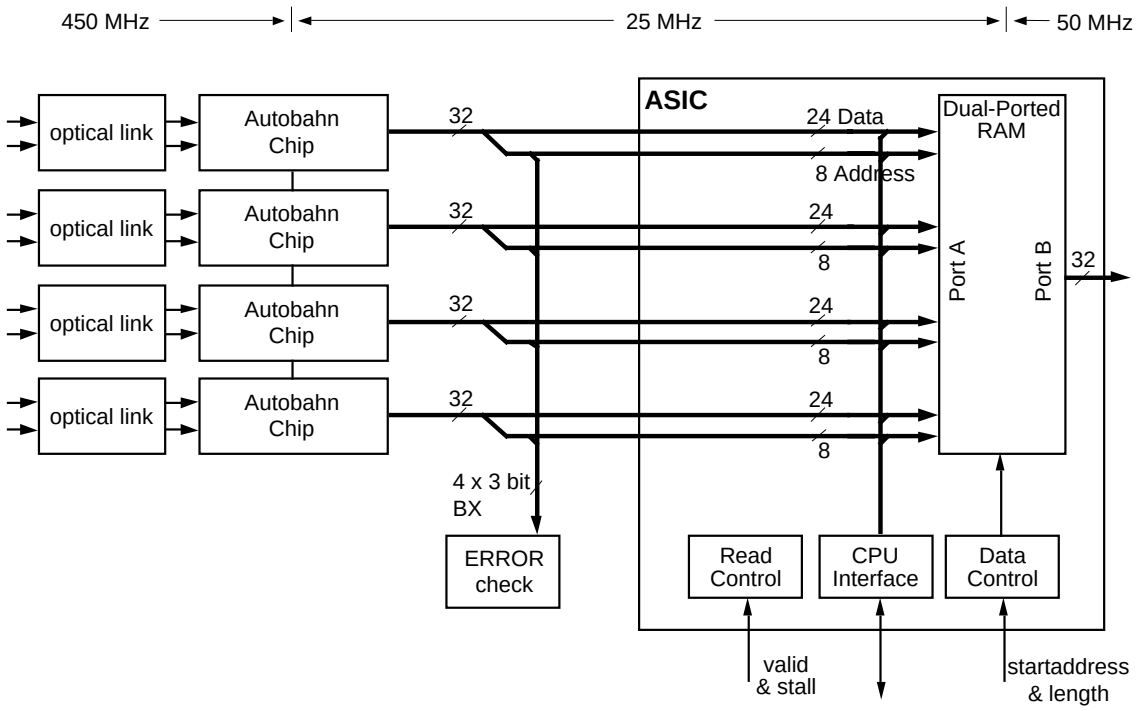


Abbildung 3.6: *Die Detektor Daten werden über die Autobahn-Chip Schnittstelle übertragen und in einem ASIC abgespeichert. Über den zweiten Port des Speichers kann der Prozessor auf die Daten zugreifen. Die TFU besitzt insgesamt sechs ASICs, für jede der drei Lagen des ihr zugeordneten Bereichs einer Detektor-Superlage zwei.*

den Spezialchips für die serielle Hochgeschwindigkeits-Datenübertragung eingesetzt. Die TFUs arbeiten mit den Autobahn-Chips von Motorola. Die Autobahn-Chips übernehmen die Parallel-Seriell- bzw. Seriell-Parallel-Wandlung und übertragen die Daten mit differentiellen ECL-Signalen bei 900 MBit/s.

Die Übertragung erfolgt auf optischem Wege. Mit geeigneten Lichtwellenleitern ist bei Datenraten von 900 MBit/s die Dämpfung über die Distanz von 60 m vernachlässigbar. Wichtige Vorteile gegenüber Koaxialkabeln sind auch die hohe Störsicherheit und die Potentialtrennung gegenüber dem Detektor. Jede TFU hat 24 Glasfaser-Eingänge mit jeweils 900 MBit/s, zusammen 2,7 GByte/s. Insgesamt werden dann ca. 1600 Verbindungen benötigt.

Die Autobahn-Chips wandeln die empfangenen seriellen Daten in ein 32 Bit Parallelformat um. Acht Bit jedes Datenworts enthalten die Ereignisnummer. Der gesamte Detektordaten-Speicher und die Zugriffslogik sind in einem ASIC untergebracht. Er verwendet die acht Bit Ereignisnummer, um damit seine Datenspeicher zu adressieren und die restlichen 24 Bit, die eigentlichen Detektordaten, unter dieser Adresse abzuspeichern. Der Datenspeicher ist als Dual-Ported RAM ausgeführt. Dadurch kann der Hardware-Prozessor gleichzeitig, unter Angabe von Adresse und Länge des gewünschten Speicherbereichs auf den Speicher zugreifen.

## 3.4 Die Hardware Prozessoren

Nach dem generellen Aufbau der Prozessorboards und ihrer Kommunikationsschnittstellen folgt nun eine Beschreibung der eigentlichen Prozeß-Einheiten. Von ihnen gibt es drei verschiedene Typen, den TFU-Prozessor, den TPU-Prozessor und den TDU-Prozessor. Hinzu kommt das Test Board, das keinen Hardware Prozessor, dafür aber eine leistungsfähigere CPU und zusätzliche Kommunikationsschnittstellen besitzt. Es wird für die Entwicklung und allgemeine Testzwecke eingesetzt.

### 3.4.1 Track Finding Unit

Der Hardware-Prozessor der Track Finding Unit (TFU) hat die Aufgabe, anhand empfangener Messages in seinen lokalen Detektordaten nach Spurdurchgängen zu suchen und gegebenenfalls die Spurrekonstruktion fortzuführen. Insgesamt besitzt er 22 Pipelinestufen und benötigt daher mindestens 240 ns, um eine Spur zu verarbeiten. Die Funktionalität eines TFU-Prozessors läßt sich grob in drei Blöcke (siehe Abbildung 3.7) einteilen:

1. Einlesen des nächsten Datenworts (die Spur-Message) aus dem Message Empfänger und Koordinatentransformation der globalen Spurkoordinaten in lokale Koordinaten. Mit diesen wird auf den lokalen Detektordaten-Speicher zugegriffen.

2. Von jeder der drei Lagen wird eine Bitfolge ausgelesen. Koinzidenzbildung aus den drei Bitfolgen in der Matrix und Weitergabe der Koordinaten der gefundenen Treffer.

3. Neuberechnung der Spurgeometrie, Extrapolation der gefundenen Spuren zu der nächsten Lage und Ausgabe in das Ausgangs-FIFO.

Die eingelesene Spur enthält in globalen Koordinaten den Ort und die Größe des Bereichs, in dem die TFU in ihren lokalen Daten nach einem Durchgang dieser Spur suchen soll. Der Ort und die Länge des Suchbereichs muß als erstes in lokale Koordinaten transformiert werden, d.h. jeweils in eine Bitadresse der drei Datenspeicher, die dem Beginn des Suchbereichs entspricht und eine Anzahl von Bits, die der Größe des Suchbereichs entsprechen. Dies geschieht parallel für alle drei Lagen der zugeordneten Superlage. Anschließend werden die drei Bitfolgen mit der berechneten Bitadresse und der Länge (maximal 32 Bit) aus dem Datenspeicher gelesen und an die Koinzidenzmatrix weitergegeben. Die Matrix sucht nach Dreierkoinzidenzen in den drei Bitfolgen. Wenn an einer Stelle alle drei Drähte gesetzt waren, so wird dies als ein Spurpunkt eines Teilchendurchgangs interpretiert. Alle gefundenen Spuren werden seriell von der Matrix ausgegeben. Wenn mehr als eine Spur gefunden wurde, muß für jede zusätzliche Spur die Pipeline vor der Matrix für einen Takt

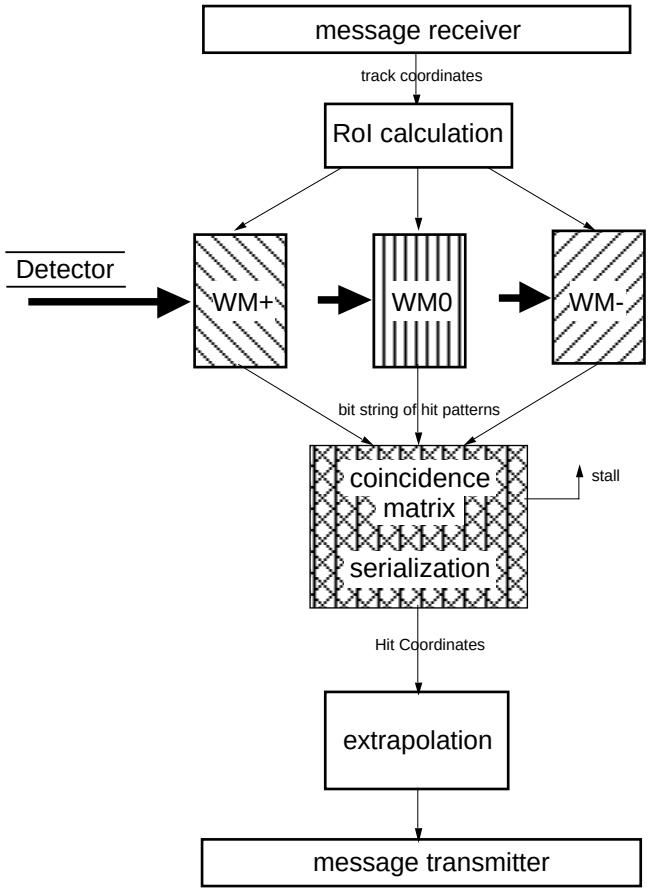


Abbildung 3.7: *Der Track Finding Prozessor.*

angehalten werden (Stall-Zyklus). Die Matrix-Ausgabe wird anschließend verwendet um den Spurverlauf erneut und genauer zu berechnen. Mit dem Ergebnis wird dann die Spur zu der nächsten Detektorlage extrapoliert und die Größe des dortigen Suchbereichs berechnet. Anhand des Resultats wiederum wird entschieden, an welche TFUs der nächsten Lage die Spurdaten gesendet werden müssen. Die letzte Pipeline-Stufe schreibt die Daten dann in den FIFO des Message-Senders, der sie dann an die nachfolgenden TFUs verschickt.

Die Koinzidenzmatrix ist in fünf großen PLDs untergebracht. Sämtliche anderen Berechnungen werden mit Hilfe von Lookup-Tables und einigen wenigen Addierern vorgenommen. Bei der Initialisierung des Prozessors müssen alle Lookup-Tables gefüllt werden. Das Berechnen und Beschreiben der Tabellen übernimmt ein Initialisierungsprogramm auf dem MC 68020 Prozessor. Die Software ist außerdem für den Abgleich von Offset-Spannungen der optischen Empfangsmodule zuständig. Während des Betriebs überwacht sie die Temperaturwerte, Betriebsspannungen, den Füllgrad der FIFOs und mehrere Zähler für Message Raten.

### 3.4.2 Track Parameter Unit

Die TFUs der letzten Detektorlage senden die Daten der rekonstruierten Teilchenspuren über das Message Transfer System an eine Track Parameter Unit (TPU).

Der Hardware Prozessor der TPU hat im wesentlichen eine Filterfunktion für Einzelspuren und berechnet kinematische Parameter einzelner Spuren. Er besitzt 12 Pipelinestufen und ist ebenfalls mit 50 MHz getaktet. Die gesamten Berechnungen werden auch hier mit Hilfe von Tabellen durchgeführt, die bei der Initialisierung mit dem MC 68020 Prozessor berechnet werden. Den schematischen Aufbau der TPU zeigt Abbildung 3.8.

Als erstes werden aus den vorhandenen geometrischen Spurdaten die kinematischen Größen der Spur, Impulsvektor, Transversalimpuls und ihre Ladung bestimmt. Anhand der Rechenergebnisse werden Schnitte auf den Impuls und den Transversalimpuls gemacht. Bei Elektronenspuren wird zusätzlich ein Vergleich zwischen Energie und dem berechneten Impuls vorgenommen und gegebenenfalls die Spur verworfen. Bei Spuren, deren Rekonstruktion ursprünglich von dem Hadron-Pretrigger initiiert wurde, wird zusätzlich die von der TPU errechnete Teilchenladung mit der von dem Pretrigger bestimmten verglichen und die Spur verworfen, wenn sie nicht übereinstimmen.

Photon Trigger Kandidaten, die von dem Elektron-Pretrigger entdeckt wurden, für die aber keine Spur rekonstruiert werden konnte, werden parallel zu den oben genannten Berechnungen gesondert behandelt (Abbildung oben rechts). Hier wird aus der Position, die der Elektron-Pretrigger liefert und der Targetposition eine Spur errechnet, wobei man davon ausgeht, daß das Photon aus der Targetregion stammt.

Im Anschluß an die Parameter-Berechnungen werden sogenannte Zwillingsspuren herausgefiltert. Das sind Messages, die sich auf die gleiche reale Teilchenspur beziehen. Solche doppelt vorkommenden Messages können beispielsweise durch das Überlappen von verschiedenen Detektoren entstehen. Dabei wird ein Teilchen, das durch eine solche Überlapp-Region fliegt, von zwei TFUs in derselben Lage registriert. Entsprechend gibt dann jede von ihnen eine Message weiter.

Um Zwillingsspuren zu entdecken, vergleicht der Track Comparator eine neue Spur, die die Parameter Berechnung passiert hat, mit den bereits vorher bearbeiteten Spuren eines Ereignisses. Dabei können maximal die letzten zwanzig Spuren verglichen werden. Wurde die aktuelle Spur bereits vorher schon einmal bearbeitet, so wird sie von dem Track Comparator verworfen.

Parallel zu dem Track Comparator arbeitet der Track Counter, der die Anzahl der Spuren eines Ereignisses zählt. Mit ihm kann die Anzahl der Messages, die die TPU von einem Ereignis passieren läßt, begrenzt werden. Hierfür kann eine Maximalzahl eingestellt werden, so daß bei Erreichen dieser Anzahl keine Spuren eines Ereignisses mehr durchgelassen werden.

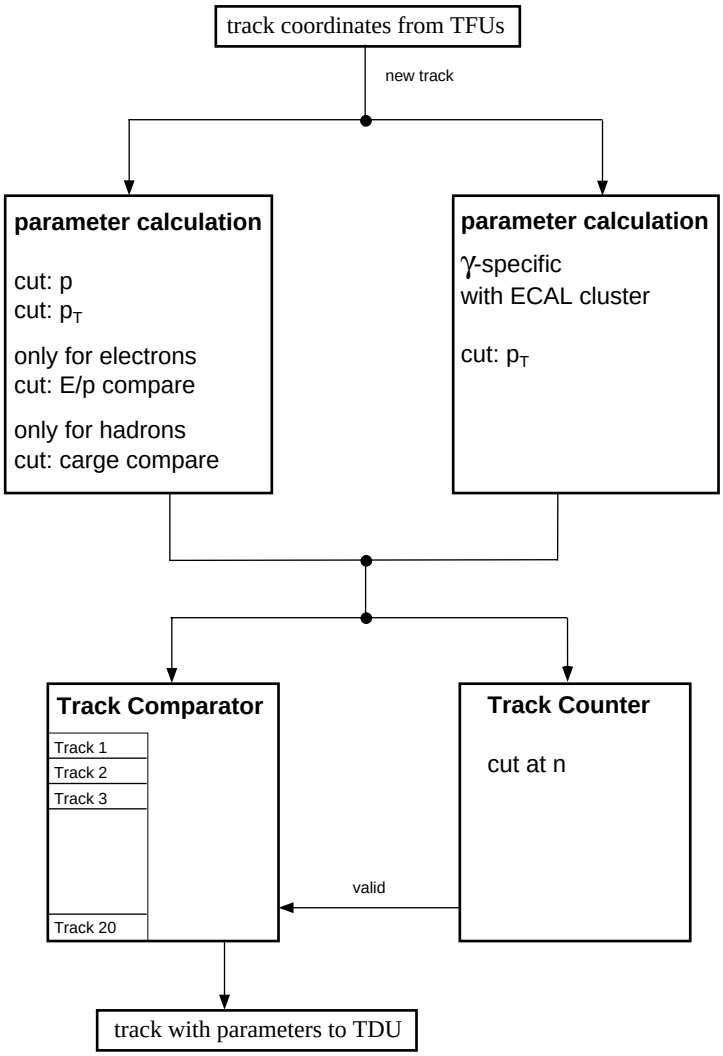


Abbildung 3.8: *Der Track Parameter Prozessor.*

Spuren, die den Comparator und den Counter passiert haben, werden als Message mit den berechneten kinematischen Parametern in das Sender-FIFO des Message Boards geschrieben, das sie dann an die Trigger Decision Unit weiterschickt.

### 3.4.3 Trigger Decision Unit

Die Trigger Decision Unit (TDU) hat die Aufgabe, die entgültige Triggerentscheidung über ein Ereignis zu treffen. Die Entscheidung basiert auf zwei verschiedenen Arten von Triggern. Der eigentliche Trigger für die Messung von B-Ereignissen analysiert die kinematischen Parameter von Spurpaaren. Der zweite Trigger wird hauptsächlich für Effizienzmessungen verwendet. Er zählt die Anzahl von Spuren, die ein Ereignis enthält, nach Spurtypen getrennt. Das Überschreiten einer vorein-

gestellte Anzahl löst einen Trigger aus. Eine positive Triggerentscheidung wird dann dem Fast Control System (FCS) mitgeteilt.

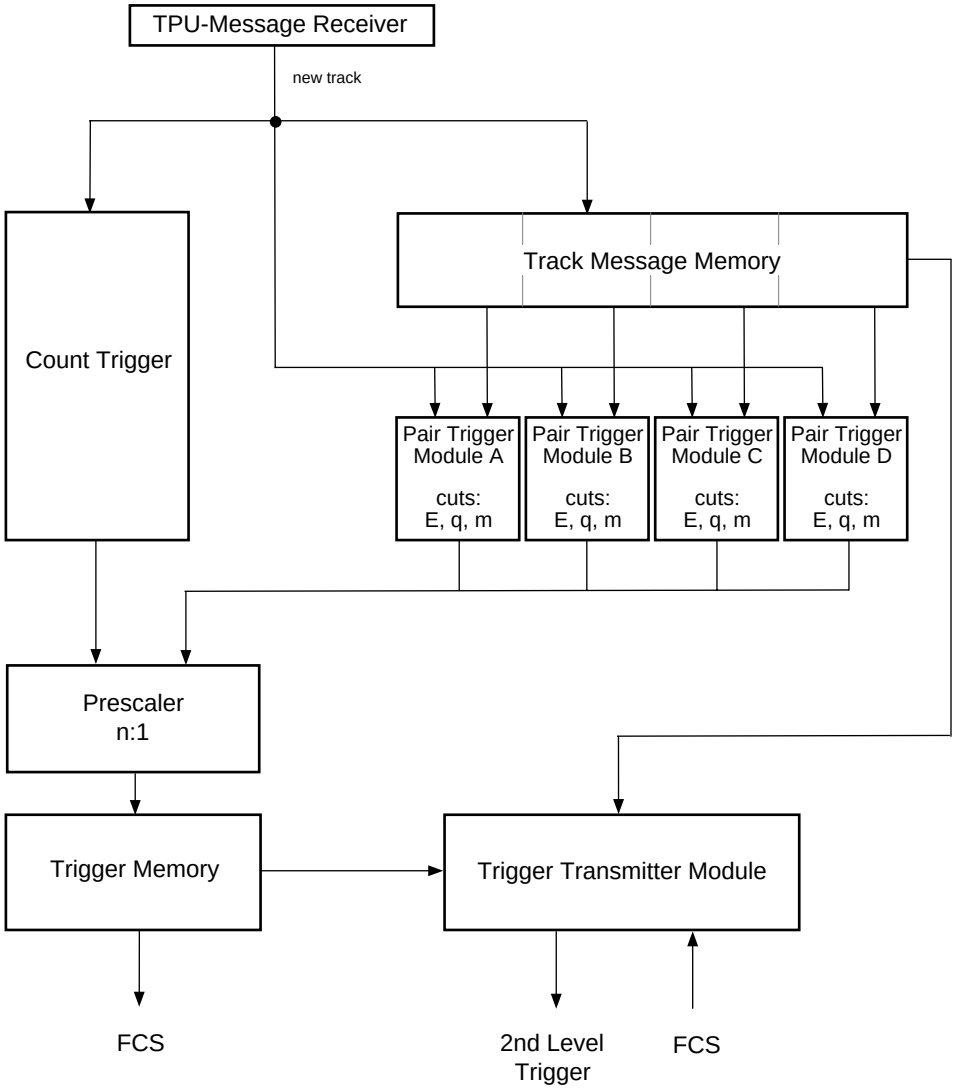


Abbildung 3.9: *Der Trigger Decision Prozessor.*

Den schematischen Aufbau des Hardware Prozessors der TDU zeigt Abbildung 3.9. Der Prozessor speichert alle gefundenen Teilchenspuren eines Ereignisses, die er von den TPUs empfängt, nach Ereignisnummer sortiert, in dem Track Memory ab. Kommt eine neue Spur in das System, wird sie zudem mit allen bereits vorhandenen Spuren des Ereignisses paarweise kombiniert und die invariante Masse des Paares berechnet. Liegt diese Masse in einem Toleranzbereich, so wird ein Trigger ausgelöst. Für die paarweise Kombination und Berechnung sind vier parallel arbeitende Pair Trigger Module zuständig. Sie bestehen im wesentlichen aus jeweils einem Lookup-Table-Prozessor. Die Parallelverarbeitung ist notwendig, um bei Er-

eignissen mit mehreren Spuren genügend Rechenleistung zur Verfügung zu haben. Bis zu vier Spurkombinationen können damit in einem Takt durchgeführt werden. Sind mehr Kombinationen vorhanden, so müssen zusätzliche Taktzyklen eingefügt werden, wobei die vorhergehenden Stufen der Pipeline dann angehalten werden. Der Lookup-Table-Prozessor vergleicht zuerst die elektrischen Ladungen der beiden Teilchen und macht Schnitte auf die Energie. Anschließend wird dann die Triggerentscheidung anhand der invarianten Masse gefällt, die in der letzten, 2 MBit großen, Lookup-Table errechnet wird.

Parallel zu dem Pair-Trigger arbeitet der Count Trigger, der ebenfalls die neu eintreffenden Spuren empfängt und für jedes Ereignis nach Typ getrennt zählt. Wird die vorgegebene Anzahl eines Typs erreicht, so löst er einen Trigger aus. Auch die Summe zweier Zähler kann als Trigger verwendet werden.

Spuren, die den Pair- oder den Count-Trigger passiert haben, werden an einen Prescaler weitergereicht. Der Prescaler kann so eingestellt werden, daß er nur jeden n-ten Trigger durchläßt. Die Trigger werden anschließend in dem Trigger Memory abgelegt. Bereits bei dem Eintreffen des ersten Triggers eines Ereignisses, wird dieser sofort dem FCS gemeldet.

Das FCS kann seinerseits die Weitergabe der Triggerinformationen eines Ereignisses von der TDU an den Second Level Trigger veranlassen. Dies geschieht für Ereignisse, auf die der FLT triggert und die anschließend von dem FCS akzeptiert werden. Aber auch Daten von Ereignissen, über die dem FCS keine positive Triggerentscheidung des FLT vorliegt, können (soweit vorhanden) weitergeleitet werden, zum Beispiel Zufallstrigger. Die TDU schickt auf Veranlassung des FCS die gesamten Daten, die sie von einem Ereignis besitzt (300 Byte), über einen SHARC-Link an den Second Level Trigger (siehe auch Abschnitt 2.4). Der Readout Controller greift hierfür, unabhängig von dem TDU-Prozessor, auf seine Trigger- und Message-Speicher zu.

### 3.4.4 Das Test Board

Das Test Board wird zum Testen der Hardwarekomponenten des FLT, insbesondere der beiden Datenübertragungs-Systeme eingesetzt. Dies ist unbedingt notwendig, da die Datenquellen des FLT, die Pretriggersysteme und Detektoren, erst nach Abschluß der Entwicklungsphase zur Verfügung stehen. Natürlich ist es auch generell günstiger, im Labor eine Testumgebung zur Verfügung zu haben, als die Elektronik am Experiment zu testen. Das Test Board nimmt außerdem eine Prototyp-Funktion bei der Entwicklung der verschiedenen Schnittstellen wahr, die bei den Prozessorboards verwendet werden. Zusätzlich wurde es als einfaches Triggersystem bei HERA-B Testläufen eingesetzt.

Das Test Board ist wie die Prozessorboards als VME-Einschub ausgeführt, besitzt aber eine leistungsfähigere CPU. Auf dem Board befindet sich ein MC 68060 Prozessor mit 50 MHz und bis zu 32 MB DRAM. Hinzu kommen 2 MB SRAM, von denen 1 MB als Second Level Cache eingesetzt werden.

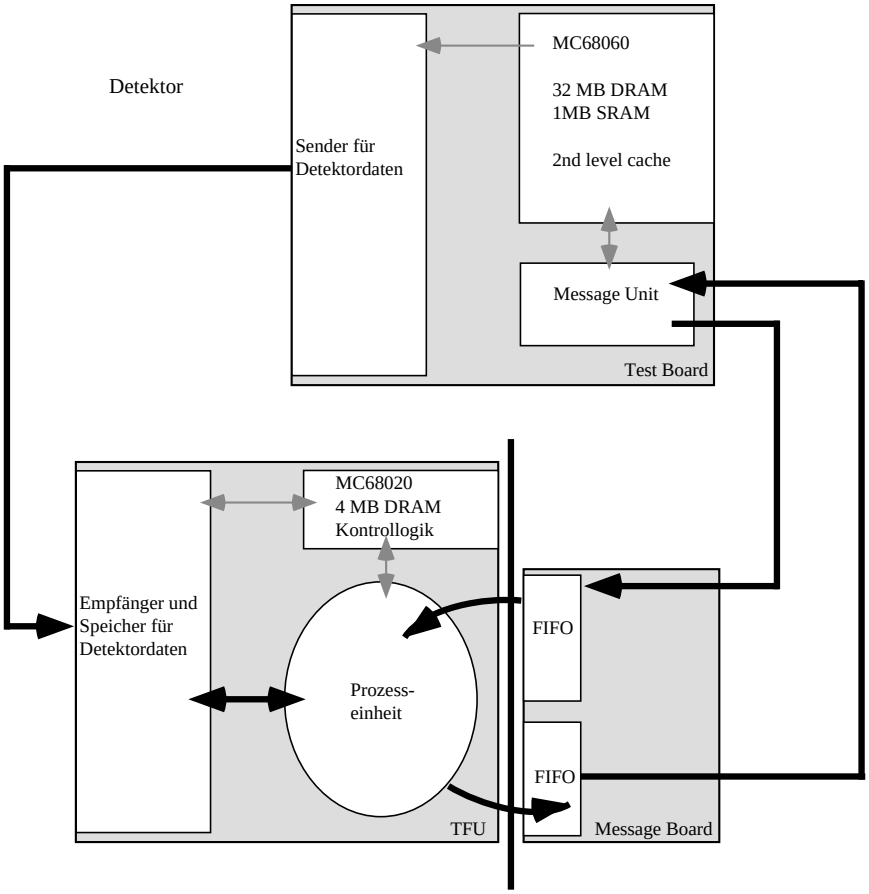


Abbildung 3.10: *Das Test Board in Verwendung für einen TFU Test. Es sendet der TFU alle benötigten Eingangsdaten und empfängt von ihr die Prozessresultate.*

Es besitzt zudem eine Schnittstelle zu einem Message Board. Sie entspricht genau der Schnittstelle der Prozessorboards, so daß das Test Board in der gleichen Weise Messages empfangen und versenden kann.

Für Testzwecke besitzt das Test Board zudem zwölf optische serielle Verbindungen mit Autobahn-Chips. Sie entsprechen den Eingängen der TFU für Detektordaten mit dem Unterschied, daß sie Daten nicht nur empfangen, sondern auch senden können. Das Benutzerprogramm auf der CPU hat hierfür Zugriff auf ein Dual Ported RAM, das als Puffer für empfangene oder zu sendende Daten dient. Um die volle Bandbreite zu ermöglichen, werden die Daten zwischen Autobahn-Chips und dem Dual Ported RAM über DMA-Controller ausgetauscht.

Zusätzlich zu diesen beiden besitzt das Test Board noch zwei weitere Schnittstellen: Auf dem Test Board selbst befindet sich eine zusätzliche Message-Schnittstelle für das Versenden und das Empfangen von Messages. Damit kann das Test Board Messages austauschen, ohne auf ein Message Board angewiesen zu sein. Insbesondere dient dies auch dem Test von Message Boards.

Sowohl die externe als auch die boardeigene Message-Schnittstelle kann in einem Burst Modus betrieben werden. In diesem können von dem Benutzerprogramm, das auf der CPU läuft, bis zu 128 K Messages abgespeichert und dann mit voller Bandbreite, d.h. mit jedem CPU-Takt eine, gesendet werden. Die Messages werden zuvor in ein an die Message-Datenbreite angepaßtes SRAM geschrieben. Mit einem DMA-Controller können die Daten dann mit voller Datenrate zwischen dem SRAM und den Sender-FIFOs (bzw. Empfänger-) ausgetauscht werden. Die Message-Schnittstelle kann auch in einem „Repeat Mode“ betrieben werden. In diesem Modus werden empfangene Messages direkt wieder über den Sender weiterverschickt. Ein weiterer Modus, der „Snoop Mode“, erlaubt zusätzlich die Entnahme von Stichproben aus den weitergeleiteten Messages. Dies ermöglicht das „Mithören“ in einem Message-Datenstrom. Abbildung 3.10 zeigt als Beispiel einen Testaufbau, bei dem die Funktion einer TFU mit Hilfe des Test Boards überprüft wird. Das Test Board sendet hierzu Detektordaten und Spur-Messages an die TFU. Die Messages, die sich als Ergebnisse des TFU-Prozessors ergeben, werden an das Test Board zurückgeschickt, das sie zum Beispiel mit Simulationen vergleichen kann.

Die zweite Zusatzschnittstelle ist eine Schnittstelle zu dem Fast Control System. Sie wurde in den Probeläufen des Detektors in den Jahren 1996 und 1997 am DESY eingesetzt. Das Test Board wurde als einfaches Triggersystem betrieben, das Messages von einem Pretriggersystem empfängt, mit dem Mikroprozessor einfache Berechnungen durchführt und die Triggerentscheidung an das Fast Control System weiterleitet. Damit konnte die Datenübertragung von den Pretriggersystemen (soweit vorhanden) bereits in einem frühen Projektstadium unter realen Bedingungen getestet werden.

## 3.5 Stand der Technologie

Die Technik der $e^+$ $e^-$-Speicherringe (siehe auch 2.2) zur Erzeugung von Elementarteilchen hat aufgrund der hohen Energieverluste durch Synchrotronstrahlung der kreisenden Teilchen eine Grenze erreicht. Der noch bis etwa zum Jahr 2001 betriebene Speicherring LEP am CERN wird vermutlich der letzte seiner Art sein. Um zu höheren Teilchenenergien vorzustoßen, verwendet eine neue Generation von Experimenten Hadron-Speicherringe, wie den HERA-Ring und in Zukunft den LHC am CERN, da bei den schweren Hadronen die Energieverluste um Größenordnungen geringer sind.

Das Hauptproblem bei den Experimenten, die mit Hadron-Hadron Wechselwirkungen ihre Teilchen erzeugen, ist der enorme Untergrund. Dies stellt zum einen neue Anforderungen an die Detektortechnologie, da die Detektoren hohe Teilchenraten verarbeiten und entsprechend strahlungshart sein müssen. Zudem stellt es auch extreme Anforderungen an die Trigger- und Datennahmesysteme. Sie müssen Ereignisraten in der Größenordnung von 10 Mhz und Datenraten im Bereich von einigen

Terabit/s verarbeiten können.

Das HERA-B Experiment ist hier als Vorläufer für die zukünftigen Experimente am LHC anzusehen, wo Ereignis- und Datenraten noch einmal um etwa einen Faktor fünf höher liegen.
Die Anforderungen an den First Level Trigger des HERA-B Experiments unterscheiden sich grundsätzlich von anderen Systemen:

- Bisher durchgeführte typische „Fixed"-Target Experimente (zum Beispiel am Fermilab NA789) arbeiteten nur mit kurzen Strahlimpulsen, die von langen Pausen gefolgt wurden, in denen die Daten aufgezeichnet und sortiert werden konnten.
- Bisherige Kollisions Experimente an $e^+$ $e^-$-Speicherringen sind zwar für eine kontinuierliche Datenaufnahme ausgelegt, die typischen Wechselwirkungsraten, die aus den Kollisionsereignissen resultieren, sind aber viel kleiner als bei HERA-B (zum Beispiel bei LEP etwa 10 Hz).

Wie am Beginn des Kapitels beschrieben, müssen bei HERA-B dagegen aus einer kontinuierlichen Ereignissrate von 10 Mhz spezifische Endzustände herausgefiltert werden, um eine Reduktion der Ereignisrate von zwei Größenordnungen zu erreichen. Über die Pretrigger ist keine Reduktion zu erreichen, weil mit einem hohen Teilchenuntergrund gerechnet wird und keine Reduktion aufgrund einfacher Signaturen möglich ist. Die angestrebte Verringerung der Ereignisrate ist nur realisierbar, wenn nach erfolgter Rekonstruktion der Teilchenspur aufgrund der Bestimmung von Energie, Impuls und invarianter Masse eine Triggerentscheidung getroffen wird. Im folgenden wird der FLT des HERA-B Experiments mit dem von H1 verglichen, einem der beiden großen Experimente am HERA Beschleuniger (siehe Abb. 2.4).
Eine Besonderheit von HERA-B ist, daß es trotz der hohen Wechselwirkungsrate ohne Totzeit arbeitet. Das heißt, die Meßdaten des Detektors werden in dem First Level Buffer zwischengespeichert und, im Fall einer positiven Triggerentscheidung, ausgelesen, ohne daß hierfür die Messung unterbrochen werden muß. Im Vergleich besitzt die erste Stufe von H1 ebenfalls einen Buffer, in dem die Meßdaten des Detektors ohne Totzeit abgespeichert werden. Bei einem Triggersignal der ersten Stufe muß der Buffer jedoch angehalten werden, um die Daten auszulesen. Dies bedingt eine Totzeit von etwa fünf Prozent.
Eine weitere Besonderheit ist der Switch für die Kommunikation zwischen Second Level Buffer und Second- und Third Level Trigger (siehe auch Abbildung 2.8). Er zeichnet sich durch eine hohe Bandbreite und durch eine niedrige Latenzzeit aus und ist damit richtungsweisend für zukünftige Experimente [59].

First Level Trigger Komponenten, die für andere Experimente entwickelt wurden, sind infolge erheblich niedrigerer Ereignisraten ($\approx$ kHz) meist sehr viel einfacher

aufgebaut und auf Korrelationen bzw. Koinzidenzen einfacher Signale beschränkt. Zum Beispiel die erste Triggerstufe von H1 besteht aus 192 unabhängigen Triggerelementen, die bei den einzelnen Subdetektoren lokalisiert sind. Für die Triggerentscheidung der einzelnen Triggerelemente werden in der Regel einfache Größen, wie die Überschreitung eines Schwellenwertes bei der Energie, herangezogen. Aus diesen Einzelwerten können dann beliebige Koinzidenzen für die Triggerentscheidung der Stufe eins gebildet werden. Die Ereignisrate von H1 liegt bei 100 kHz, die Ausgangsrate der ersten Triggerstufe bei 5 kHz. Die erste Stufe erreicht also eine Reduktion der Rate um den Faktor 20 bei einer Trigger Latenzzeit von 2,3 $\mu$s [22].

Der HERA-B First Level Trigger setzt daher neue Maßstäbe für Ereignisraten und Algorithmen in der ersten Stufe. Der verwendete Algorithmus entspricht in seiner Komplexität Algorithmen, wie sie in anderen Experimenten normalerweise frühestens ab der zweiten Stufe angewendet werden. Ein großer Teil der klassischen Funktionen eines Second Level Triggers ist beim HERA-B Experiment im First Level Trigger implementiert, um den hohen geforderten Reduktionsfaktor (ca. 200) zu erreichen. Die Eingangsrate des FLT liegt zwei Größenordnungen über der von H1. Gleichzeitig ist sein Reduktionsfaktor zehnmal größer. Spezialhardware zur Mustererkennung in den Detektordaten wird erst in der zweiten Triggerstufe von H1 eingesetzt.

Entsprechend umfangreicher und qualitativ aufwendiger ist die Elektronik des FLT im Vergleich zu Komponenten anderer Experimente. Hier werden Multi-Layer Leiterkarten im 9 HE VME-Bus Format mit SMD-Technik verwendet. Für den Aufbau der Logik werden hochintegrierte Logikbausteine, die mit 50 MHz getaktet sind, eingesetzt. Ungewöhnlich ist auch die Verwendung zusätzlicher Mikroprozessoren auf den VME-Boards. Normalerweise erfolgt in Physik-Experimenten die Steuerung ausschließlich von einem VME-Host Rechner aus, der über Register auf VME-Einschübe zugreift.

Die Architektur des FLT bedingt damit auch eine wesentlich aufwendigere Software zum Betrieb des Systems, als dies normalerweise der Fall ist. Der Betrieb erfolgt mit Hilfe eines heterogenen verteilten Rechnersystems, das aus etwa 80 Rechnern besteht, und zudem sehr viele Einstell- und Testmöglichtkeiten auf den Hardwareprozessoren besitzt. Gleiches gilt für eine Simulation des Triggers, die ein sehr komplexes und zudem asynchron arbeitendes System beschreiben muß. Sie muß daher das Zeitverhalten des Triggers mit berücksichtigen und kann sich nicht auf reines Datenprozessieren beschränken.

# Kapitel 4

# Die Konzeption der Software

In diesem Kapitel wird ein Überblick über den Softwarebedarf des FLT gegeben und dabei das Gesamtkonzept für die Software, das in [58] erstellt wurde, dargelegt. Das Kapitel beginnt mit einer Beschreibung der Hardware und Betriebssysteme, die die Umgebung der zu erstellenden Software bilden. Anschließend wird eine Einteilung in verschiedene Pakete und deren hierarchische Aufteilung in System- und Applikationssoftware vorgenommen. An die Beschreibung der einzelnen Pakete schließt sich ein Überblick über die Methoden und Werkzeuge an, die eingesetzt werden, um eine effiziente und zielgerichtete Entwicklung der Software zu unterstützen. Am Ende des Kapitels wird eine Erläuterung der Grundlagen externer Pakete gegeben, die Sprache Tcl, Netzwerk-Sockets, der Cross-Compiler und Bibliotheken, die im Rahmen der FLT-Software Verwendung finden. Sie ist zum Verständnis der nachfolgenden Kapitel notwendig.

In diesem Kapitel werden ein Teil der Entwurfsentscheidungen und die Auswahl der Technologie, die in den nächsten Kapiteln getroffen werden, bereits vorweggenommen, um einen vollständigen Überblick geben zu können. Die ausführliche Anforderungsanalyse und Auswahl der Technologie für die Teile, die im Rahmen dieser Arbeit implementiert wurden, erfolgt in den jeweiligen Kapiteln.

## 4.1 Die Hardware- und Betriebssystem-Umgebung der ersten Triggerstufe.

Die Systemgrenzen und der Kontext des First Level Triggers wurden bereits in Kapitel 3 beschrieben. Intern dient der Kontrolle und Steuerung des FLT ein verteiltes heterogenes Rechnersystem aus etwa 90 Rechnern (vergleiche Abbildung 3.2). Es bildet die Entwicklungs- und Laufzeitumgebung für die Software des FLT. Insgesamt werden drei verschiedene Hardware-Plattformen mit ihren entsprechenden Betriebssystemen eingesetzt. Die Rechner nutzen zwei verschiedene Kommunikationsschnittstellen, Ethernet und VME-Bus, für den Datenaustausch.

### 4.1.1 Systeme und Betriebssysteme

**M68k Prozessor.** Auf den Prozessor-Boards TFU, TPU und TDU kommen Mikroprozessoren Motorola 68020 mit 4 MB RAM (Test-Board: MC68060 mit 16 MB - 32 MB) zum Einsatz. Die Board-Rechner besitzen aber keine eigenen Schnittstellen für Massenspeicher oder andere Ein-/Ausgabegeräte. Als Schnittstellen zur Außenwelt besitzt der Rechner im normalen Betrieb nur den VME-Bus. Für Testzwecke ist zusätzlich ein RS-232 Terminalanschluß vorhanden, der an der Frontplatte jedes Prozessorboards angebracht ist. Sämtliche Dateizugriffe, das Laden von Programmen und Interaktionen eines Programms mit dem Benutzer müssen daher über den VME-Bus abgewickelt werden. Die 68k-Rechner besitzen keinerlei Betriebssystem oder sonstige Standardsoftware, welche diese Funktionen für die Boards übernimmt.

Die Kommunikation der Board-Rechner via VME-Bus mit der Außenwelt erfordert daher einerseits in jedem VME-Crate einen Host-Rechner, der Zugriff auf einen Massenspeicher oder ein Netzwerk besitzt. Andererseits wird eine Betriebssoftware benötigt, die den Datentransfer über den VME-Bus zwischen diesem Host und dem Board regelt.

**LynxOS mit PowerPC.** Als VME-Host-Rechner werden in den Crates des FLT Rechner der Firma Cetia verwendet, die als Standardsystem für VME-Hosts bei HERA-B eingesetzt werden. Die Cetia CPUs sind als VME-Einschübe mit 6 HE ausgeführt. Bislang werden Versionen mit PPC601/100MHz und PPC604/200MHz Prozessoren benutzt.

Der Host-Rechner besitzt ein VME-Bus Interface, dessen Adreßraum für VME-Bus Zugriffe in den Adreßraum eines Prozesses eingeblendet werden kann. Für Netzwerkkommunikation steht je nach Ausführung eine 10 MBit/s bzw. 100 MBit/s Ethernet-Schnittstelle zur Verfügung.

Als Betriebssystem wird LynxOS verwendet. LynxOS ist ein vollständiges Unix-System mit Realtime-Eigenschaften und Multithreading. Dies erlaubt den Einsatz weit verbreiteter Unix-Programmpakete, wie zum Beispiel GNU-Compiler, BSD-Sockets und Tcl/Tk. Weiterhin ist eine effiziente Programmausführung durch Multithreading mit einstellbaren Prozeß- und Thread-Prioritäten möglich.

Der Host-Rechner wird ohne lokale Festplatte betrieben. Er bootet mittels des TFTP-Protokolls [1] von einem Bootserver. Beim Starten des Rechners wird auf eine intern angelegte Ram-Disk, ein „elementares“ Dateisystem geladen. Nach dem Systemstart werden externe Dateisysteme dann über NFS [2] gemountet. Trotz des Verlustes von Dateisystemperformance ist der Betrieb über ein Netzwerk bei solchen experimentellen Systemen vorteilhaft. Bei einem Rechnerabsturz kann das Dateisystem einer Festplatte so stark beschädigt werden, daß kein Neustart mehr möglich

[1] Trivial File Transfer Protocol.
[2] Network File System

ist. [3] Während der Entwicklung von Software, die auf die VME-Hardware zugreift, kommt es häufig zu Systemabstürzen - hier ist dann aber lediglich die RAM-Disk betroffen, die beim Start ohnehin neu erzeugt wird.

Als Software Schnittstelle zu dem VME-Bus wird eine VME-Bibliothek zusammen mit einem VME-Treiber der Firma Mizzi-Computer verwendet (siehe Abschnitt 4.6.2). Der Treiber wird beim Start des Host-Rechners geladen und initialisiert einen angegebenen Bereich des VME-Bus, auf den dann mit virtuellen Adressen zugegriffen werden kann. Die Bibliothek stellt eine Programmierschnittstelle zu dem VME-Bus zur Verfügung. Dies entlastet den Entwickler von hardwarenaher Programmierung und erhöht die Portabilität der Programme.

**Linux Workstation.** Alle zentralen Funktionen werden von einem Linux-System wahrgenommen, daß auch als einziges über einen Massenspeicher verfügt. Es dient daher als Dateiserver für das gesamte System. Teile des Linux Dateisystems werden mit NFS von Lynx gemountet.

Der Linux-Server wird auch als Bootserver für Lynx verwendet. Hierzu arbeitet unter Linux ein Dämon-Prozeß „bootpd“, der auf Netzwerk-Anfragen startender Lynx Rechner wartet. Die anfragenden Rechner werden über ihre Ethernet-Adresse identifiziert und können anschließend mit TFTP-Zugriff ihr Lynx-System laden.

### Schnittstellen der Hardware

Die Kommunikation zwischen Prozessen auf den beschriebenen Rechnersystemen findet über zwei verschiedene Schnittstellen statt:

Die Interprozeßkommunikation zwischen einem Prozeß auf dem Unix-Host und einem Prozeß auf einem Prozessorboard erfolgt über den VME-Bus als gemeinsame Schnittstelle. Der Unix-Host kann über den VME-Bus auf den Arbeitsspeicher des 68k-Prozessors und die Kontrollregister des Boards zugreifen, um beispielsweise 68k-Binärdateien in den Speicher zu laden oder einen Reset des Boards auszulösen. Der Board-Prozeß wiederum kann über den VME-Bus einen Interrupt auf dem Unix-Host auslösen.

Der zentrale Linux-Rechner kommuniziert via Ethernet mit Prozessen auf den VME-Hosts. Seine Funktion als Dateiserver für das gesamte System nimmt er mit Hilfe des NFS wahr, das Teile des Linux-Dateisystems durch Mounten unter Lynx zur Verfügung stellt. Zusätzlich wird eine Interprozeßkommunikation zwischen Prozessen auf den VME-Host Rechnern und Linux-Prozessen benötigt, die beispielsweise zentrale Steuerungs- oder Archivierungsaufgaben wahrnehmen.

Für einen Zugriff auf das Linux-Dateisystem, oder für den Datenaustausch zwischen einem Linux-Prozeß und einem Board-Programm muß also ein LynxOS-Prozeß vor-

[3] Hauptsächlich wegen der von Unix zur Beschleunigung von Plattenzugriffen eingesetzten Disk-Write-Caches.

handen sein, der als Bindeglied zwischen VME-Bus und Ethernetschnittstelle fungiert.

## 4.2 Bedarf und Modularisierung

Um die Komplexität der Triggersoftware und ihre möglichen Anwendungsfälle zu überblicken, wird die Software in verschiedene Anwendungsbereiche und Pakete unterteilt. Zu jedem Paket wird im folgenden eine kurze Zieldefinition und die Nutzer der Software angegeben.

Der Softwarebedarf für den FLT läßt sich grob in drei Anwendungsbereiche einteilen, auf die im folgenden eingegangen wird: Betriebssoftware („Online-Software“) Testsoftware und Simulationssoftware. Diese Pakete sind natürlich nicht wirklich unabhängig, im Gegenteil, es sollen Teile der Software soweit wie möglich in den verschiedenen Paketen wiederverwendet werden.

Aus diesem Grund wird die Software noch hierarchisch in Applikationssoftware und Systemsoftware strukturiert. Hierfür gilt es Komponenten zu identifizieren, die von mehreren Applikationen eingesetzt werden können. Sie werden unter der Bezeichnung Systemsoftware zusammengefaßt. Die Systemsoftware bildet eine Basis, die dann von mehreren, darauf aufbauenden Applikationen genutzt werden kann. Dadurch werden Mehrfachentwicklungen vermieden und die Wiederverwendung von Code ermöglicht. Zudem reduziert sich das benötigte Wissen und damit der Einarbeitungsaufwand für Entwickler der Applikationen, wenn in bestimmten Bereichen auf Standardlösungen zurückgegriffen werden kann.

Abbildung 4.1 zeigt in einem Paket-Diagramm die Einteilung der verschiedenen Pakete in UML-Notation [4]. Dieser Diagrammtyp dient der Strukturierung großer Softwaresysteme. Pakete können alle Arten von Software-Elementen enthalten, wie zum Beispiel Module, Bibliotheken und Komponenten. Sie können geschachtelt oder mit gestrichelten Kanten, die am Ende einen Pfeil enthalten, zueinander in Beziehung gesetzt werden.

### 4.2.1 Applikationssoftware

#### Online-Software

Die Online-Software wird zur Steuerung und Überwachung des First Level Triggers während des Betriebs benötigt. Abbildung 4.2 zeigt ein Paket-Diagramm der Online-Software in UML-Notation

Die Online-Software beinhaltet einen zentralen Kontrollprozeß, der auf einer Linux Workstation abläuft. Von hier aus wird das komplette System überwacht. Die

[4] Unified Modeling Language. Die UML wurde im November 1997 von der Object Management Group als Standard-Modellierungssprache verabschiedet [35]. (siehe auch Abschnitt 7.3.2)

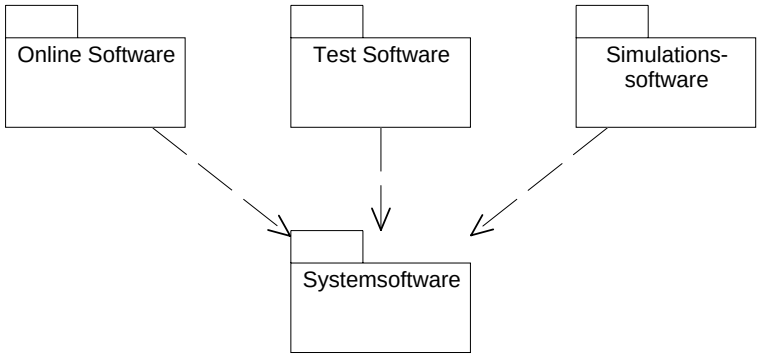


Abbildung 4.1: *Paket-Diagramm mit den drei Paketen der Applikationssoftware, die auf die Systemsoftware des FLT zurückgreifen.*

Kontrolle des Systems muß auch für Personen ohne detaillierte Kenntnisse der Triggerhardware möglich sein. Aus diesem Grund ist eine grafische Benutzeroberfläche erforderlich, die einen Überblick über den Aufbau und den Betriebszustand des Gesamtsystems vermittelt („System Display“). Dies beinhaltet auch die Anzeige des Zustands einzelner Boards und boardbezogener Daten. Weiterhin muß die Konfiguration und der Ablauf eines Trigger-“Runs“ zentral verwaltet werden. Für die Implementierung der Benutzeroberfläche wird die Programmiersprache Tcl/Tk mit dem Erweiterungspaket Tix eingesetzt.

Auf jedem einzelnen Prozessorboard muß ebenfalls ein Online-Prozeß ablaufen. Für jeden der drei Prozessortypen gibt es ein eigenes (oder auch mehrere) Online-Programm. Es initialisiert und steuert vor Ort die Boardhardware, gibt Betriebsdaten und gegebenenfalls Fehlermeldungen an die Zentrale weiter. Hierzu müssen die Lookup-Tabellen des jeweiligen Boards mit Hilfe der zugehörigen Tabellenfunktionen, die in einer Bibliothek organisiert sind, berechnet werden. Außerdem muß das Programm auf die Hardware des jeweiligen Boards zugreifen.

Die Kommunikation zwischen Zentrale und Board erfolgt über Ethernet und VME-Bus, was einen zusätzlichen Serviceprozeß auf dem Unix-Host erfordert, der die Vermittlung übernimmt. In Abbildung 4.2 ist die Software für die Board- und lynx-seitige VME-Kommunikation und den Serviceprozeß in dem Paket VME-Systemsoftware zusammengefaßt, das von allen Board-Prozessen benötigt wird. Für die Netzwerkkommunikation greifen sowohl der zentrale Kontrollprozeß, als auch der Serviceprozeß auf das in dem Tcl-Paket enthaltene Netzwerkmodul zurück.

Für Testaufbauten und Inbetriebnahme des Triggers ist eine häufige Umprogrammierung von Online-Prozessen für die Boards erforderlich. Diese Versuche werden in der Regel ohne zentrale Steuerung vorgenommen. Im Laufe der Entwicklung und im späteren Betrieb werden viele Benutzer an der Programmierung von Boards beteiligt sein. Daher ist es unbedingt erforderlich abstrakte Schnittstellen zur Programmierung der Hardware zur Verfügung zu stellen, um insgesamt den Entwicklungs- und Einarbeitungsaufwand zu reduzieren.

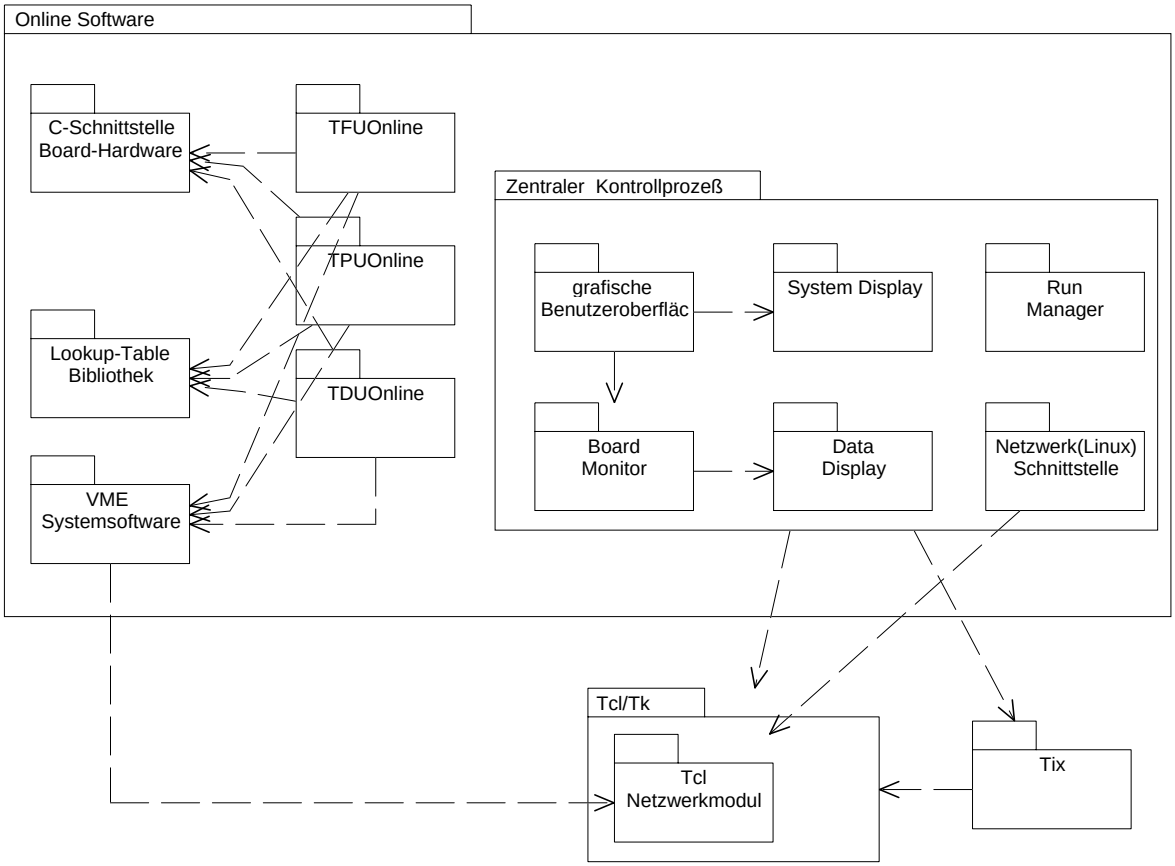


Abbildung 4.2: *Paket-Diagramm mit den Komponenten der Online Software für den FLT.*

Dies beinhaltet zum einen, daß die Boards nicht nur in Assembler, sondern auch in einer Hochsprache programmiert werden können. Zudem muß die VME-Bus Kommunikation möglichst einfach zu bedienen sein. Weiterhin ist es sehr lohnend eine abstraktere Schnittstelle zu der Board-Hardware als Bibliothek zur Verfügung zu stellen. Dann muß man sich bei der Programmierung der Boards nicht jedesmal mit den Details der Hardware auseinandersetzen.

### Testsoftware

Für die Inbetriebnahme der FLT-Boards und ihrer Schnittstellensysteme sind aufgrund deren Komplexität umfangreiche Tests notwendig. Diese Tests werden von Hardware-Experten durchgeführt. Da der FLT aus etwa 80 Prozessorboards und ebensovielen Message-Boards aufgebaut ist, müssen die Testprogramme für eine komfortable und effiziente Benutzung ausgelegt sein.

Zusätzlich zu den Tests für die Einzelkomponenten erfordert der Betrieb dann Tests des Gesamtsystems oder Teilen davon, zum Beispiel die Überprüfung der richtigen Verkabelung und Funktionsfähigkeit des Message-Systems (etwa 300 Verbindungen).

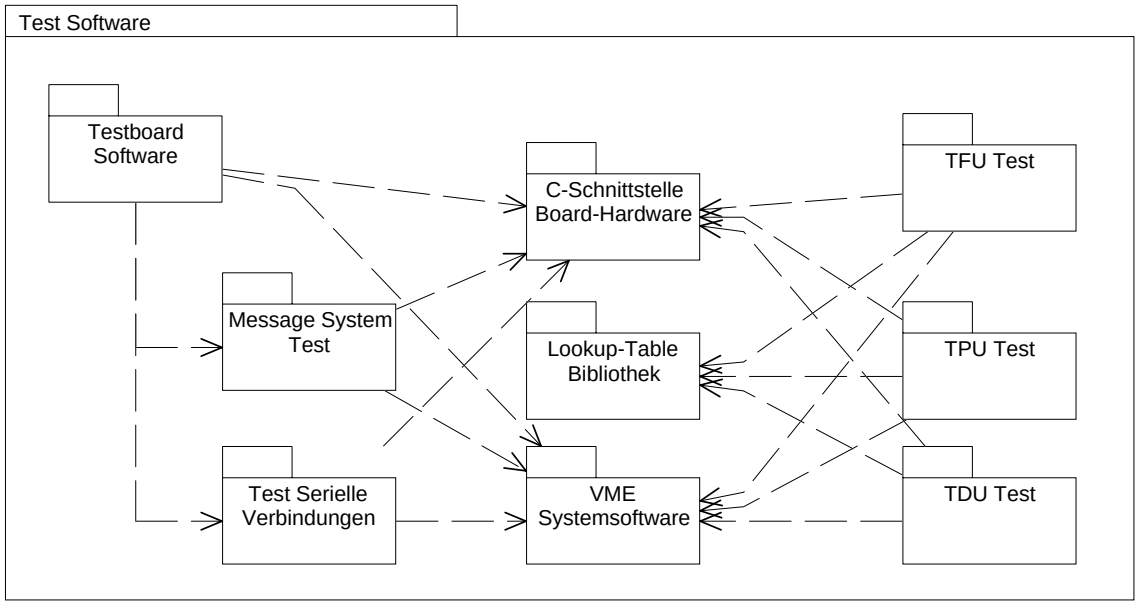


Abbildung 4.3: *Die Komponenten der Test Software.*

Hinzu kommen Standardtests, die vor und während dem Betrieb des FLT automatisch durchgeführt werden, und eine Reihe von Standardtests, die beim Auftreten von Fehlern per Hand ausgelöst werden können.

Eine wichtige Rolle bei den Tests spielen die beiden Test-Boards. Sie ermöglichen insbesondere den Test der Message- und Detektor-Schnittstellen eines Boards. Zudem dienen sie generell als Sender und Empfänger von Testdaten im Rahmen von Tests der Prozessorboards. Alle Module die Testsoftware enthalten machen Gebrauch von der VME-Systemsoftware, die die Umgebung liefert in der Board-Programme ablaufen (siehe Abbildung 4.3). Ebenso verwenden auch die Tests die Lookup-Tabellen und müssen natürlich auf die Hardware zugreifen.

Die Realisierung von Testsoftware und die Durchführung von Tests ist nicht Thema dieser Arbeit und wird daher im Folgenden nicht weiter betrachtet.

### Die Simulation des Triggersystems

Um die FLT-Boards testen zu können, wird eine detaillierte zeitaufgelöste Simulation der einzelnen Prozessorboards benötigt. Mit ihren Ergebnissen können die jeweiligen Testresultate dann überprüft werden. Diese Simulation wird von Benutzern mit genauen Kenntnissen der Hardware durchgeführt.

Zudem wird eine Simulation des Gesamtsystems für die Beantwortung eher physikalischer Fragestellungen, wie zum Beispiel nach der Effizienz des Triggers, benötigt. Um die Konsistenz zwischen der Hardware und ihrer Simulation sicherzustellen, muß diese Gesamtsimulation auf den verifizierten Einzelboardsimulationen aufbauen. Teil

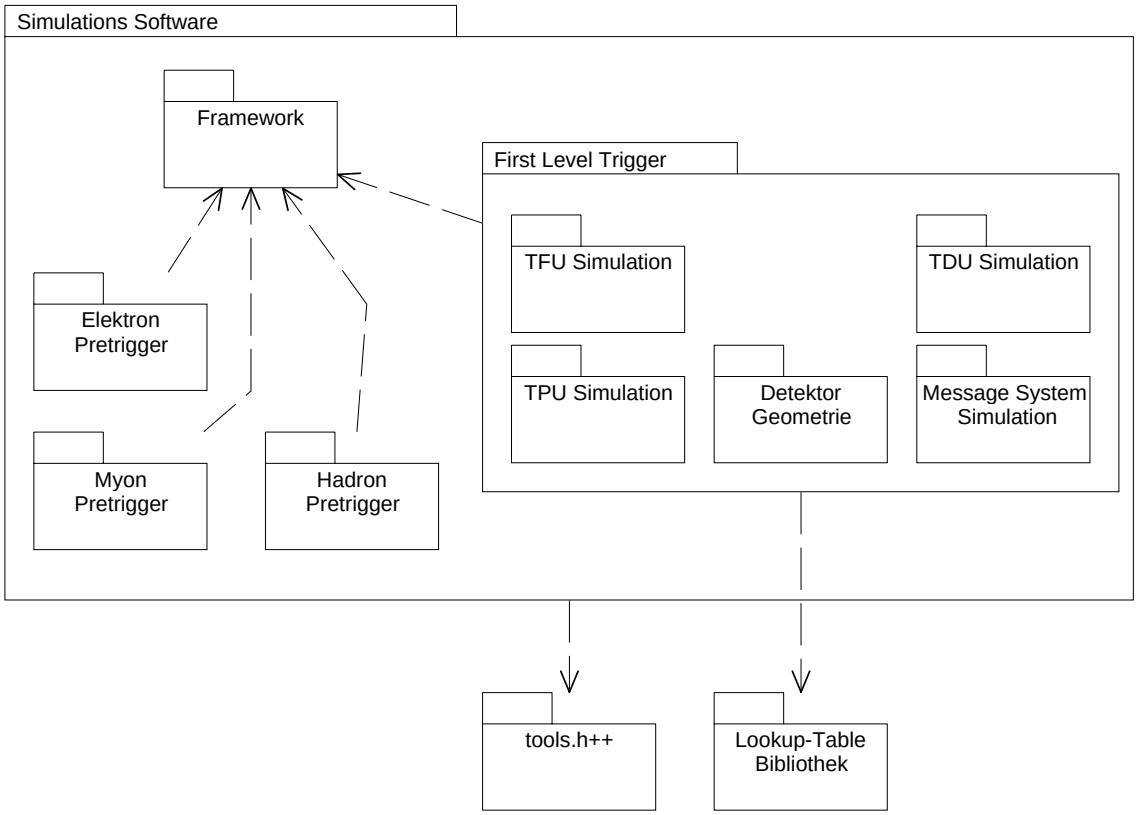


Abbildung 4.4: *Die Komponenten der Simulations Software.*

der Gesamtsimulation sind auch die drei Pretriggersysteme, deren Subsystem-Simulationen an den jeweiligen Instituten entwickelt werden müssen, wo auch die Hardware gebaut wird. Die Gesamtsimulationen müssen von Personen durchgeführt werden, die keine vollständige Kenntnis der Hardware besitzen - schon allein deswegen, weil es niemanden gibt, der die gesamte Hardware aller vier Triggersysteme im Detail kennt. Es muß daher ein Weg gefunden werden, Simulationen von Teilsystemen auf relativ einfache Weise für Gesamtsimulationen handhaben zu können. Weiterhin stellen der Umfang des zu simulierenden Systems und die Verteilung der Entwicklung auf mehrere Institute besondere Anforderungen dar. Um diesen zu begegnen, wird ein objektorientiertes Framework eingesetzt, auf dessen Grundlage die Simulationen in den einzelnen Instituten entwickelt werden. Die Simulation verwendet die Klassenbibliothek Tools.h++ (siehe Abschnitt 4.6.2) und in den FLT-Simulationen die Bibliothek mit den Lookup-Table Funktionen.

### 4.2.2 Die Systemsoftware

In der Systemsoftware sind die Teile der FLT-Software zusammengefaßt, die als Grundlage der eigentlichen Applikationsentwicklung dienen. Abbildung 4.5 zeigt die

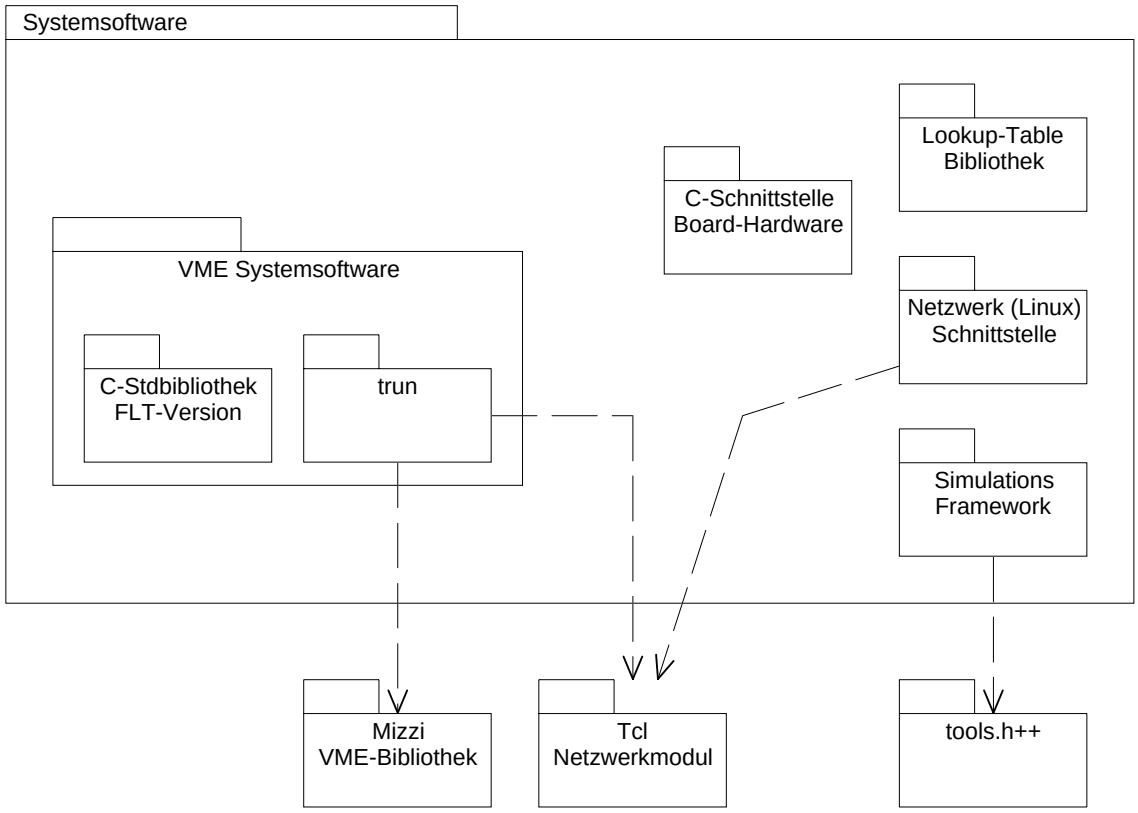


Abbildung 4.5: *Die Komponenten der System Software.*

einzelnen Pakete der Systemsoftware. Die meisten kommen bereits in mehreren der vorigen Paket-Diagramme vor, da sie in der Regel in verschiedenen Applikationen eingesetzt werden. Im wesentlichen besteht die Systemsoftware aus fünf Teilen:

- Eine C-Schnittstelle zu der Hardware der verschiedenen Boards. Sie liegt für jeden Boardtyp als Bibliothek vor, die in eigene Programme eingebunden werden kann, um eine abstraktere Schnittstelle zu der Board-Hardware zur Verfügung zu haben.

- Eine Laufzeitumgebung für C-Programme auf den Prozessorboards, im folgenden VME-Systemsoftware genannt. Sie beinhaltet für die Boardprogramme eine an die spezielle Hardwareumgebung angepaßte Version der C-Standardbibliothek, in der die VME-Bus Kommunikation enthalten ist und einen Service-Prozeß unter Lynx als Kommunikationspartner. Für den Service-Prozeß, der den Namen `trun` [5] erhalten hat, wird die VME-Bibliothek der Firma Mizzi-Computer als Schnittstelle zum VME-Bus eingesetzt (siehe Abschnitt 4.6.2).

[5] trigger run

- Eine C-Bibliothek mit Funktionen zur Berechnung der Lookup-Tables auf allen Boards. Diese Bibliothek wird sowohl zum Konfigurieren der Prozessoren, als auch für die Simulation des Triggers verwendet. Es gibt nur eine Version jeder Tabellenfunktion, die am DESY entwickelt und gepflegt werden.
- Eine Interprozeßkommunikation über das Netzwerk, zwischen Prozessen, die auf einem der Lynx Rechner, und Prozessen, die unter Linux laufen. Hier wird auf beiden Seiten das in der Sprache Tcl enthaltene Netzwerk-Paket eingesetzt.
- Ein Framework für die Simulation des Triggersystems. Während die anderen Pakete der Systemsoftware eher Bibliothekscharakter besitzen, dient der auf verschiedene Institute verteilten Entwicklung der Subsystem-Simulationen ein objektorientiertes Framework. Es bietet einen hohen Grad an Code-Wiederverwendung. Zudem legt es die Subsystem-Simulationen auf eine gemeinsame Architektur fest, so daß sie in einer Gesamtsimulation vereint werden können.

Im Rahmen dieser Arbeit wurden die meisten Teile der Systemsoftware realisiert. Die VME-Systemsoftware wurde vollständig implementiert und wird in Kapitel 5 beschrieben. Auf der Linux-Seite wurde die Netzwerkkommunikation zusammen mit einem Prototyp für den zentralen Kontrollprozeß mit grafischer Benutzeroberfläche realisiert (Kapitel 6), um zu zeigen, daß in diesem Bereich das gewählte Konzept umgesetzt werden kann. Ebenfalls realisiert wurde das objektorientierte Framework für die Simulation der Triggersysteme, siehe hierzu Kapitel 7.

## 4.3 Allgemeine Softwarestandards

Nach einer mehrjährigen Entwicklungsphase ist für das HERA-B Experiment eine Laufzeit von 5 Jahren vorgesehen. Die Software des FLT soll daher eine Lebensdauer von mindestens acht Jahren besitzen. Dies bedeutet, daß diejenigen, die am Design und der Entwicklung der Software beteiligt waren, während der eigentlichen Durchführung des Experiments in der Regel nicht mehr verfügbar sind. Im Laufe der Betriebsjahre ist auch immer wieder mit Änderungen und Erweiterungen der Software zu rechnen, da ein solches Experiment eigentlich nie fertig aufgebaut ist, sondern ständigen Verbesserungen unterliegt. Weiterhin sollen Teile der FLT Software, insbesondere die Systemsoftware, auch von Benutzern ohne Spezialwissen über die Programme oder die Hardware verwendet werden können.

Um dieser Situation gerecht zu werden, werden Software Engineering Methoden bei der Entwicklung angewendet. Dabei ist es nicht sinnvoll, die volle Bandbreite dieser Methoden einzusetzen. Vielmehr wurde in jedem Teilprojekt nach einem sinnvollen Kompromiß zwischen Ad hoc Programmieren und einem übertriebenen Modellierungs- und Verwaltungs-Aufwand gesucht.

Dies beinhaltet eine kontrollierte Vorgehensweise, also einen Entwicklungsprozeß als Basis. Der Prozeß beginnt mit der Erstellung eines generellen Konzepts [58] für die

Software, das in Abschnitt 4.2 vorgestellt ist. Das Konzept enthält einen Überblick über die gewählten Methoden und Werkzeuge für die Entwicklung, über die Systemsoftware und die einzelnen Applikationen für den FLT. Daran schließt sich ein nachvollziehbarer Software Entwurf an. Die FLT Software wird hierfür in Projekte eingeteilt und deren jeweilige Anforderungen, das grundlegende Design und der Datenfluß spezifiziert. Bei der Implementierung wird Software-Technologie nach dem Stand der Technik eingesetzt und eine Dokumentation der Software erstellt. Hinzu kommt die Verwendung von Werkzeugen, die diese Vorgehensweisen unterstützen.

Die Verwendung bestimmter Methoden und Entwicklungswerkzeuge hat im Wesentlichen zwei Ziele:

1. Die Zusammenarbeit von Entwicklern, die räumlich oder zeitlich getrennt arbeiten, zu verbessern.

2. Die Entwicklung guter Software: Gutes Design, Wartbarkeit, Erweiterbarkeit und möglichst wenige Fehler.

Hierzu gehört als genereller Rahmen der bereits erwähnte Prozeß. Mehr implementierungbezogene Aspekte betreffen die Wahl der Programmiersprachen (hier C, C++ und Tcl/Tk), die Vereinbarung von Programmierrichtlinien und die Verwendung von Entwurfsmustern bei C++ Programmen.

Zudem werden soweit wie möglich fertige Softwarepakete verwendet. Dies betrifft den Einsatz von Bibliotheken und von Skriptsprachen, die einen hohen Abstraktionsgrad bei der Programmierung bieten.

An Werkzeugen wird das Versions-Kontrollsystem CVS, das Analyse und Design Werkzeug Rose mit C++ Code-Generator und DOC++ zur Dokumentationserstellung eingesetzt.

### 4.3.1 Richtlinien und Dokumentation

Von Anfang an wird die FLT-Software nach einheitlichen Programmierrichtlinien entwickelt („Programming Style Guide“). Die darin enthaltenen Konventionen sollen dem Code ein einheitliches Aussehen und eine sinnvolle Struktur geben. Dadurch verbessern sich die Lesbarkeit und die Nachvollziehbarkeit des Codes.

Zudem wird auf eine ausreichende Entwickler- und Benutzerdokumentation Wert gelegt. Aufgrund der Erfahrung, daß in den allermeisten Fällen entweder während der Entwicklung dokumentiert wird, oder überhaupt nicht, wird der größte Teil der Dokumentation als Kommentare in den Quellcode geschrieben. Mit dem Werkzeug DOC++ wird dann daraus, durch Analyse des Quellcodes, automatisch eine Dokumentation in HTML oder Tex generiert. Besonders wichtig ist die Dokumentation der Systemsoftware, die in anderen Applikationen eingesetzt werden soll.

### 4.3.2 Die Programmiersprachen

Als Programmiersprachen zur Entwicklung der FLT Software werden C, C++ und Tcl/Tk verwendet. In einigen Ausnahmefällen werden 68k-Assemblerroutinen bei der Programmierung der Boards benötigt.

Die Programme auf den Boards und der Service-Prozeß `trun` unter Lynx (also die VME-Systemsoftware und darauf aufbauende Programme) sind in C geschrieben, da es sich sehr gut zur Systemprogrammierung eignet und unter Unix ohnehin die Sprache der Wahl ist.
Auf die Möglichkeit der Programmierung der Boards in C++ wurde verzichtet, obwohl als C++ Cross-Compiler der GNU g++ zur Verfügung steht. Dies zum einen, um die Komplexität der Laufzeitumgebung nicht weiter zu erhöhen, zum anderen auch, um Entwicklungs- und Wartungsaufwand für diese Umgebung zu sparen. Da die Programme auf den Boards nicht allzu komplex sind und C-Programme außerdem schneller ausgeführt werden, stellt dies keinen Nachteil dar.
Unter LynxOS stand bei Beginn der Entwicklung der VME-Systemsoftware (März 96) kein C++ Compiler zur Verfügung. Daher wurde das bereits bestehende objektorientierte Design für den Service-Prozeß verworfen und `trun` in C geschrieben.

Die Trigger Simulation ist in C++ geschrieben. Der C++ Code läßt sich mit dem Code Generator direkt aus dem Rose Design Modell heraus erzeugen.
Der zentrale Kontrollprozeß mit grafischer Oberfläche und die Netzwerkkommunikation sind in Tcl/Tk implementiert.

## 4.4 Die Sprache Tcl

Wie bereits in den Paket-Diagrammen 4.2 und 4.5 zu sehen ist, wird in der FLT Software an mehreren Stellen die Sprache Tcl verwendet. Tcl [6] und Tk [7] sind zwei Software Pakete, die zusammen ein System zur Entwicklung grafischer Benutzerschnittstellen bilden.
Tcl ist eine relativ einfache Skriptsprache [8], mit der neue Applikationen geschrieben werden oder auch bestehende, in einer anderen Sprache geschriebene, erweitert werden können [36]. Wie bei Skriptsprachen üblich, ist Tcl eine interpretierte Sprache und nicht typisiert, das heißt es gibt nur den Typ String [37].
Tcl ist in C geschrieben, der Tcl-Interpreter besteht im Prinzip aus einer Bibliothek von C-Funktionen (Abbildung 4.6). Dadurch kann der Befehlssatz der Sprache um beliebige, in C (oder C++) geschriebene, benutzerspezifische Befehle erweitert

[6] Tool Command Language
[7] Toolkit
[8] Auch als VHLL- Very High Level Language bezeichnet.

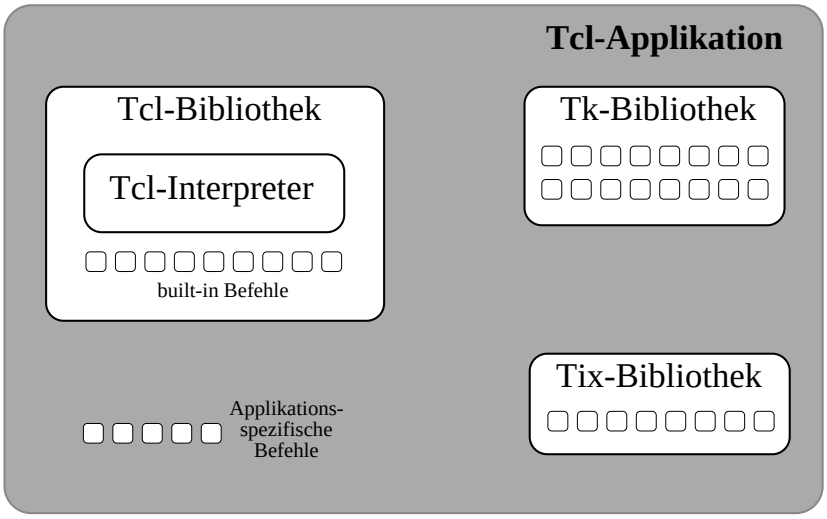


Abbildung 4.6: *Eine TCL-Applikation besteht aus einem Tcl-Interpreter und applikationspezifischen Befehlen. Zusätzlich können Erweiterungspakete wie Tk und Tix hinzugezogen werden. [36]*

werden, falls der vorhandene Sprachumfang nicht ausreicht. Komplexe Operationen können als Tcl-Kommando gegebenenfalls in C realisiert werden, während ein Tcl-Scipt diese Operationen dann zusammenspielen läßt.

Eine weitere Möglichkeit ist das Einbinden eines Tcl-Interpreters in eine C-Applikation. Das C-Programm wird dadurch in die Lage versetzt, Tcl-Skripte zu interpretieren. Diese Möglichkeit wird in der VME-Systemsoftware von `trun` genutzt (siehe Abschnitt 5.4).

Tk ist ein Toolkit für das X Window System [9]. Es erweitert den Satz der Tcl-Befehle um zahlreiche Funktionen zur Erzeugung von Oberflächenelementen, sogenannte Widgets, und, zu deren Layoutgestaltung auf dem Bildschirm, um Geometry Manager. Mit Hilfe dieser Elemente, die Tk zur Verfügung stellt, lassen sich grafische Oberflächen durch Tcl-Skripte anstatt durch C-Code erstellen.

Tk selbst besteht ebenfalls aus einer Bibliothek von C-Funktionen, bei deren Implementierung der Erweiterungsfähigkeit besondere Aufmerksamkeit geschenkt wurde. Dies ermöglich es Tk´s Funktionalität um neue Widgets und neue Geometry Manager zu erweitern.

Aufgrund dieser Erweiterungsmöglichkeiten wurden von der Tcl/Tk-Benutzergemeinschaft bereits eine Vielzahl von Bibliotheken hervorgebracht. Sie konzentrieren sich meist auf ein spezielles Anwendungsgebiet, wie zum Beispiel die Anbindung von Datenbanken an Tcl. Auch für den FLT-Kontrollprozeß wird eines dieser Zusatzpakete, Tix, verwendet.

[9] Mittlerweile ist Tcl/Tk auch für MS-Windows und MacOS verfügbar.

### 4.4.1 Die Tk Erweiterung Tix

Die Tix Bibliothek ist eine Erweiterung von Tk, die etwa 40 neue Widgets enthält. Tix stellt komplexe Widgets zur Verfügung, deren Verwendung zusätzliche Programmierarbeit spart und der Oberfläche ein professionelles Aussehen verleiht. Zum Beispiel das Notebook-Widget (siehe Abbildung 6.3) und die Datei-Auswahlbox aus Tix werden für den zentralen Kontrollprozeß eingesetzt.

## 4.5 Netzwerkkommunikation mit Sockets

Unter Unix gibt es zwei Arten von Interprozeßkommunikation mit denen über ein Netzwerk Daten ausgetauscht werden können, Datenströme und Sockets [10] [52]. Die Sockets, die von Berkley (BSD-Unix) eingeführt wurden, haben sich im wesentlichen durchgesetzt. Daher werden auch für die FLT-Software Sockets eingesetzt. Bei der Verwendung von Sockets muß noch geklärt werden, was für eine Art von Verbindung zwischen den Prozessen bestehen soll und was für ein Netzwerkprotokoll eingesetzt wird.

### 4.5.1 Verbindungsart und Netzwerk-Protokoll

Die Sockets unterstützen drei verschiedene Arten der Vernetzung:

1. Zuverlässiger verbindungsorientierter Bytestrom
2. Zuverlässiger verbindungsorientierter Paketstrom
3. Unzuverlässige Paketübertragung

Die Art der Vernetzung wird in Verbindung mit einem Protokoll bei der Erzeugung eines Sockets angegeben [54].

Für zuverlässige verbindungsorientierte Byte- und Paketströme wird in der Regel das TCP/IP [11] Protokoll verwendet. TCP ist ein verbindungsorientiertes Protokoll, vor der Datenübertragung muß daher erst eine Verbindung aufgebaut werden, die dann während des gesamten Vorgangs bestehen bleibt. Daher fungiert einer der beiden Prozesse immer als Server, der Verbindungsanfragen eines Clients entgegennimmt. Steht die Verbindung, so ist der Unterschied zwischen Client und Server aufgehoben und beiden Prozessen steht ein bidirektionaler Kommunikationskanal zur Verfügung, analog zu einer Pipe.

Für unzuverlässige Paketübertragung wird normalerweise das verbindungslose UDP [12] Protokoll eingesetzt. Es bietet eine maximale Paketgröße von 8 KB. Die Pakete

[10] Kommunikationsendpunkte
[11] Transmission Control Protocol
[12] User Datagram Protocol

werden verschickt ohne vorher eine Verbindung aufzubauen und ohne zu überprüfen, ob sie auch tatsächlich ankommen. Gesendete Datenpakete können verloren gehen, doppelt ankommen oder in vertauschter Reihenfolge eintreffen. Bei Verwendung von UDP muß der Programmierer selbst Vorsorge für eine zuverlässige Datenübertragung treffen.

Aufgrund der zuverlässigen Übertragung und des geringeren Implementierungsaufwands, werden für die FLT-Software TCP-Sockets mit Bytestrom eingesetzt. Idealerweise ist genau diese Art von Socket-Kommunikation bereits Bestandteil von Tcl [56]. Es ist daher keine Eigenentwicklung oder die Einbindung eines Zusatzpakets notwendig, um Sockets für den zentralen Kontrollprozeß zu verwenden, dessen Oberfläche ebenfalls mit Tcl erstellt ist.

### 4.5.2 Sockets

Sockets sind Schnittstellen zwischen dem Benutzer und einem Netzwerk, die von einem Programm dynamisch erzeugt und zerstört werden können. Das Erzeugen eines Sockets erzeugt einen Kommunikationsendpunkt an den von der Betriebssystemseite her („von unten“, vergleiche Abbildung 4.7) Netzwerkverbindungen angefügt werden können und es liefert („nach oben“) einen Dateideskriptor, der für jede Art von Zugriff des Benutzerprogramms auf den Socket benötigt wird. (Siehe einheitliches Modell für Ein-/Ausgabe unter Unix, Abschnitt 5.2.2.)

Für die Erzeugung eines Sockets muß das Adreßformat angegeben werden, mmit dem der Socket bei der Kommunikation angesprochen werden kann. Bei den FLT-Sockets wird hierfür das Adreßformat des Internet Protocols, also die IP-Nummer des Rechners oder sein Internet-Domainname, verwendet. Eine andere Möglichkeit um Sockets zu identifizieren ist zum Beispiel die Verwendung von Unix Pfadnamen. Dies geht aber nur zwischen Prozessen auf einem gemeinsamen Host.

Weiterhin muß bei Erzeugung eines Sockets sein Typ angegeben werden. Dieser legt die Semantik der Kommunikation fest (siehe 4.5.1). Hier verwenden die FLT-Sockets den STREAM Typ, das heißt zuverlässigen verbindungsorientierten Bytestrom.

Bevor ein Socket zur Netzwerkkommunikation verwendet werden kann, muß ihm noch eine eindeutige Kennung, die Portnummer, zugewiesen werden. Jede Portnummer existiert nur einmal pro Host.

Wenn zwei Prozesse miteinander über Sockets in Kontakt stehen, so müssen sie neben der IP-Nummer des jeweils anderen Hosts auch wissen, unter welcher Portnummer desselben der andere Socket anzusprechen ist. Über diese Verbindung können sie dann bidirektional Daten austauschen. Abbildung 4.7 zeigt dies am Beispiel einer bestehenden Verbindung zwischen den Programmen `tricon` und `trun`.

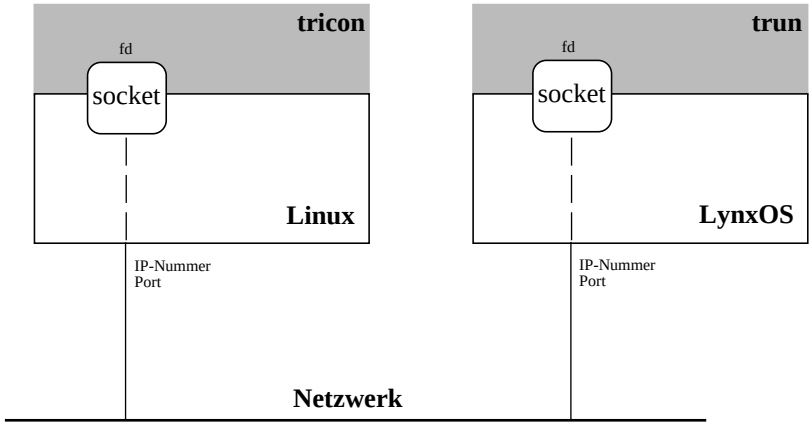


Abbildung 4.7: *Eine bestehende Netzwerkverbindung zwischen den Programmen* `tricon` *und* `trun` *über ihre Sockets. Die Programme verwenden einen Dateidescriptor (fd) für Zugriffe auf ihren Socket. Das jeweilige Betriebssystem stellt dem Socket einen Port zur Verfügung, über den er seine Netzwerkkommunikation abwickeln kann.*

### Tcl Sockets

Die Tcl Sprache besitzt eine eingebaute Schnittstelle zur Verwendung von Sockets, mit deren Hilfe Netzwerk-Clients und -Server programmiert werden können. Die Schnittstelle ist sehr einfach, macht dafür aber Einschränkungen bezüglich den Verbindungsarten und den verwendeten Protokollen. Tcl Sockets verwenden ausschließlich das TCP Netzwerk Protokoll mit Bytestrom Verbindung.

Ein Tcl Skript kann Netzwerk Sockets wie eine geöffnete Datei oder eine Pipeline verwenden. Statt das `open`-Kommando für Dateien wird das `socket`-Kommando verwendet um einen Socket zu erzeugen. Bei der Erzeugung muß dann lediglich noch die IP-Adresse und die Portnummer angegeben werden. Danach können zum Beispiel die Tcl-Kommandos `puts` oder `gets` wie bei einer Datei zur Datenübertragung über das Netzwerk verwendet werden.

## 4.6 Zusätzliche Installationen

Fast die gesamte Softwareentwicklung, sowohl für Lynx als auch für die 68k-Prozessoren, kann auf dem Linux System stattfinden. Einzige Ausnahme ist der Aufruf des nativen Lynx Compilers, der natürlich unter LynxOS aufgerufen werden muß. Dies bietet die Möglichkeit die gewohnte Umgebung für die Softwareentwicklung zu verwenden, zum Beispiel integrierte Entwicklungsumgebungen, die in der Regel nicht für Lynx verfügbar sind.

Weitere Software-Installationen für den FLT sind der Cross-Compiler für 68k-Prozessoren und die bereits in den Paket-Diagrammen aufgeführten Bibliotheken.

### 4.6.1 Cross-Compiler für 68k-Prozessoren

Eine FLT Installation, die unter Linux betrieben wird, ist das Cross Entwicklungs Paket für die 68k-Prozessoren. Hierzu wurden die GNU Software Entwicklungswerkzeuge als Cross Versionen „Intel nach Motorola 68k“ compiliert und installiert. Im Einzelnen beinhaltet dies:

- Cross Compiler `gcc2.7.2`
- Cross Assembler `as`
- Cross Linker `ld`
- Die Cross Versionen der Programme `ar` und `ranlib`, die für das Verwalten von Bibliotheken notwendig sind.

Das Compilieren des Cross Compilers verlief nach geringfügigen Source Code Modifikationen ebenso problemlos wie das der anderen Programme und er wird seit März 1996 für das Übersetzen der 68k-Programme und -Bibliotheken eingesetzt.

Dem Link-Vorgang, der normalerweise weitgehend transparent erfolgt, muß hier besondere Beachtung geschenkt werden. Der Code des Benutzers muß an die richtigen Adressen (Board-Adressen) und mit den richtigen System-Bibliotheken gelinkt werden (nicht die Linux Bibliotheken). Außerdem muß ein Assemblerprogramm mit dem Startup-Code eingebunden werden, das die Funktion `main` aufruft.

### 4.6.2 Bibliotheken

Der Einsatz von Programmbibliotheken verkleinert die zu entwickelnden Programme und reduziert damit den Entwicklungsaufwand. Außerdem verringert sich die Fehlerhäufigkeit durch den Rückgriff auf Bewährtes, und es verbessert sich die Portabilität der Software auf andere Plattformen, für die die Bibliotheken ebenfalls verfügbar sind. Im Rahmen der FLT Software werden zwei gekaufte Bibliotheken eingesetzt:

1. Die VME-Bibliothek der Firma Mizzi Computer stellt eine Schnittstelle zum VME-Bus zur Verfügung. Dadurch enthalten die Programme, die auf den VME-Bus zugreifen eine einheitliche Schnittstelle auf einem höheren, hardwareunabhängigen Niveau. Es ist daher einerseits nicht notwendig sich mit den Lynx spezifischen Eigenschaften der VME-Schnittstelle auseinanderzusetzten, andererseits sind die Programme dadurch auf andere Systeme portabel (zum Beispiel auf OS9). Die Programme können auch von anderen Programmierern leichter verstanden werden, da die Bibliothek als Standard bei HERA-B eingesetzt wird und eine gute Dokumentation existiert.

2. Die Klassenbibliothek Tools.h++ wird für die Trigger Simulationsprogramme verwendet. Tools.h++ ist ein Industrie Standard und stellt eine Sammlung von Klassen für häufig benötigte Typen zur Verfügung, zum Beispiel Strings und eine große Zahl von Container-Klassen.

# Kapitel 5

# Die VME-Systemsoftware

Die Aufgabe der VME-Systemsoftware ist die Bereitstellung einer Programmierumgebung für die Mikroprozessoren auf den VME-Boards. Diese Entwicklungs- und Laufzeitumgebung soll möglichst weitgehend der gewohnten Unix- und C-Umgebung entsprechen und dadurch ein einfaches Arbeiten mit den Boards ermöglichen. Die für diese Aufgaben entwickelte Software exportiert einen Teil der Schnittstelle zwischen dem Unix-Betriebssystem und Prozessen des VME-Host Rechners auf die Boards und stellt sie damit einem Board-Prozeß zur Verfügung. Die Laufzeitumgebung umfaßt einerseits einen Service Prozeß „trun“, der auf dem Host Rechner läuft. Er ist für das Laden der Board-Programme und die VME-Bus Kommunikation zuständig. Den zweiten Teil auf der Board-Seite bildet eine an die spezielle Hardware der Boards angepaßte Version der C-Standardbibliothek, die um betriebssystemartige Funktionen erweitert wurde. Neben der fast kompletten Implementierung der C-Standardbibliothek enthält sie den boardseitigen Part der VME-Bus Kommunikation und die Speicherverwaltung des Boards. Im letzten Teil des Kapitels wird die Anbindung der Board-Prozesse an die Netzwerkschnittstelle des Host beschrieben. Sie wird für den Zugriff auf einen Dateiserver und die Kommunikation mit Steuerprozessen benötigt.

## 5.1 Die Laufzeitumgebung für Board Programme

Das FLT-Multiprozessorsystem ist in neun VME-Crates untergebracht. Die typische Konfiguration eines FLT-VME-Crates zeigt Abbildung 5.1. Das Crate enthält einen Unix-Host-Rechner und bis zu 13 Prozessorboards. Die Kontrolle eines Boards erfolgt vollständig per Software mit Hilfe seines MC68020 Mikroprozessors, der über 4 MB Arbeitsspeicher verfügt (vergleiche auch Abbildung 3.3).

Der boardeigene Rechner verfügt im EPROM über ein Monitorprogramm. Es wird beim Einschalten automatisch gestartet und kann über eine serielle Terminalschnittstelle angesprochen werden. Damit stellt der Rechner eine Testumgebung bei der

Inbetriebnahme eines Boards zur Verfügung. Durch Programmierung in Assembler kann das Board getestet werden, ohne daß, außer einer Stromversorgung, zusätzliche Peripherie benötigt wird.

Während des Betriebs und auch für die Softwareentwicklung und Serientests ist diese Schnittstelle aber nicht brauchbar. Hier ist es nicht praktikabel, jedes Board an ein Terminal anzuschließen, sondern die Prozessorboards sollen über den VME-Bus kontrolliert werden. Der Board-Rechner besitzt zu diesem Zweck eine VME-Bus-Schnittstelle. Da die FLT-Boards keine weitere Peripherie besitzen, muß jegliche Art von I/O über den VME-Bus mit Hilfe des Host-Rechners abgewickelt werden. Der Unix-Host wiederum hat über den VME-Bus Zugriff auf den Arbeitsspeicher und einen Teil der Kontrollfunktionen der Prozessorboards.

Die Binärdateien der Programme, die auf den Boards ablaufen, müssen ebenfalls über den VME-Bus geladen und gestartet werden. Anschließend übernehmen sie die Initialisierung, Betriebstests, Fehlermeldung und Kontrolle des Hardwareprozessors. Falls eine Ein- oder Ausgabe notwendig ist, müssen sie diese via VME-Bus mit dem Unix-Host abwickeln. Die Ethernet-Schnittstelle des Host stellt die einzige Verbindung des Kontrollsystems nach außen dar.

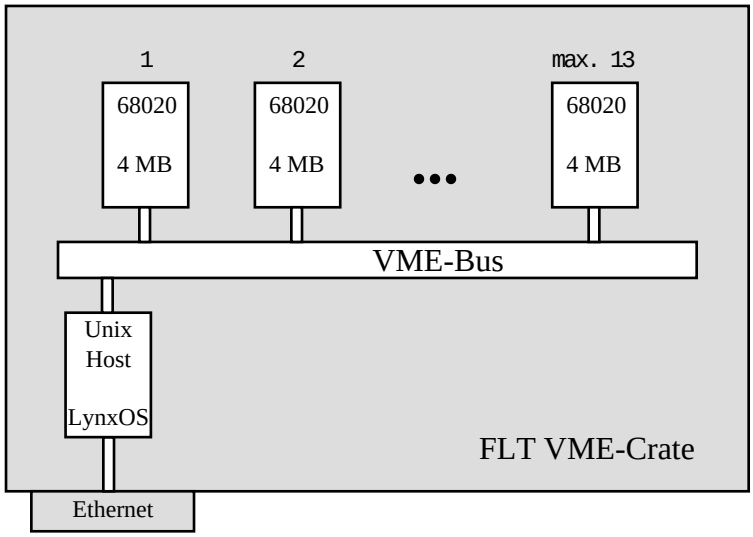


Abbildung 5.1: *VME-Crate mit Unix-Host und mehreren Prozessorboards. Der Host stellt mit seiner Ethernet-Schnittstelle die einzige Verbindung der Mikroprozessor-Systeme nach außen zur Verfügung.*

### 5.1.1 Anforderungen

Das Ziel der Laufzeitumgebung ist die Bereitstellung von Betriebssystemfunktionen für Applikationen, die auf den Boards laufen. Ein weiteres übergeordnetes Ziel bei der Entwicklung der Laufzeitumgebung ist die einfache Programmierbarkeit der

boardeigenen Rechner. Es sollen auch Entwickler oder Benutzer ohne detaillierte Kenntnisse der Systemsoftware oder -Hardware in der Lage sein, Programme für die Boards zu entwickeln. Damit kann dann ohne größeren Einarbeitungsaufwand Betriebs- oder Testsoftware für die Boards entwickelt und benutzt werden. Dies stellt natürlich höhere Ansprüche an die Laufzeitumgebung der Programme und führt damit zu einem größeren Entwicklungsaufwand. Im einzelnen sollen folgende Bedingungen von der Laufzeitumgebung erfüllt werden:

- Die Schnittstelle zum Programmieren von Applikationen (API [1]) soll weitgehend hardwareunabhängig sein.
- Die Interaktion mit einem Benutzerprogramm (Benutzerschnittstelle) soll in üblicher Weise erfolgen.
- Die Programmentwicklung soll in der gewohnten Unix Umgebung möglich sein.
- Der Zugriff auf ein Dateisystem, zum Aufrufen eines Benutzerprogramms oder für die Handhabung von Dateien, soll ebenfalls in üblicher Weise erfolgen.

Als Basisanforderung ergibt sich demnach, daß die Boards in einer Hochsprache, in C, programmiert werden können. Dabei sollen die C-Programme eine möglichst hohe Transparenz aufweisen gegenüber der speziellen Hardwareumgebung, in der sie ablaufen.

Unabhängig von ihrer gewählten Implementierung muß die VME-Systemsoftware folgende Betriebssystemfunktionen erfüllen:
Auf der Unix-Host Seite:

- Laden und Starten von Benutzerprogrammen durch den Host in den Arbeitsspeicher eines Boards.
- VME-Bus Kommunikation. Auf dem Host-Rechner muß ein Service-Prozeß laufen, der via VME-Bus Anfragen von einem Board-Programm entgegennimmt. Weiterhin muß er auch in der Lage sein, Anfragen an das Board-Programm zu stellen bzw. weiterzuleiten (z.B. von einem zentralen Steuerprogramm).
- Weitergabe von Signalen des Unix-Systems an den Board-Prozeß. Dies ist eine von dem Host aus initiierte Kommunikation mit dem Board-Prozeß.
- Netzwerkkommunikation mit einem zentralen Datei-Server und Steuerprogrammen.

Auf der Boardseite:

[1] Application Programming Interface

- Initialisieren und Ausführen geladener Programme.
- VME-Bus Kommunikation mit dem Serviceprozeß auf dem Host bei jeder Art von I/O.
- Speicherverwaltung des Board-Rechners.
- Es wird keine Kommunikation der Board-Prozesse untereinander benötigt.
- Es wird kein Multiprozessing auf den Boards gebraucht.

Für die Interprozeßkommunikation zwischen einem Host-Prozeß und einem Board-Prozeß müssen die jeweiligen Protokolle definiert werden.

### 5.1.2 Auswahl der Laufzeitumgebung

Als ersten Schritt gilt es zwischen verschiedenen grundlegenden Konzepten zu wählen, mit denen die gestellten Anforderungen erfüllt werden können. Unabhängig von dem gewählten Konzept muß auf jeden Fall ein Host-Prozeß gestartet werden, der anfangs das Laden ausführbarer Software in den Board-Arbeitsspeicher übernimmt und anschließend mit der Software auf dem Board über den VME-Bus kommuniziert. Der Ablauf eines geladenen cross-compilierten Programms auf der Board-CPU ist dann direkt möglich und solange problemlos, wie kein Betriebssystemaufruf erfolgt. Ein solcher Systemaufruf ist nicht Bestandteil der Programmiersprache und muß daher von der Laufzeitumgebung eines Programms ausgeführt werden. Für die Realisierung einer Laufzeitumgebung für Programme auf den Board-CPUs bieten sich drei alternative Konzepte an:

1. Der direkteste Weg ist die Implementierung einer eigenen VME-Bus Kommunikationsbibliothek, die als Schnittstelle zu dem Unix-System dient und zu dem jeweiligen Benutzerprogramm hinzugelinkt wird. In diesem Fall werden überhaupt keine Systemaufrufe verwendet, sondern die entsprechenden Aufgaben werden von der speziellen Bibliothek wahrgenommen. Sie enthält zum Beispiel ein Aufruf wie `VmeWrite(Dateiname, Text);` für eine Dateiausgabe via VME-Bus auf das Dateisystem des Host. Ein Unix Service-Prozeß, der auf dem Host läuft, nimmt dann die Parameter entgegen, führt das entsprechende Kommando aus und gibt das Ergebnis zurück. Zuvor lädt er das ausführbare Programm, das sowohl den Code des Benutzers als auch den der Bibliothek enthält, in den Arbeitsspeicher des Boards und startet das Programm. Man wird versuchen, die Schnittstelle so schmal wie möglich, aber dennoch für den gewünschten Zweck vollständig zu programmieren.

   Der Applikationsprogrammierer muß dann spezielle Kenntnisse für die Benutzung der Bibliothek erwerben. Stößt er auf Anforderungen, die von der

Bibliothek nicht erfüllt werden, so muß er sich entweder mit ihrem Innenleben auseinandersetzen oder einen Spezialisten kontaktieren. Da es sich um eine Neuentwicklung handelt, sind auf längere Zeit immer wieder solche Änderungen zu erwarten. Wenn die Deklaration einer Funktion geändert wird, zum Beispiel die Anzahl oder der Typ ihrer Parameter, so muß der gesamte Code, der diese Funktion benutzt, überarbeitet werden. Diese Lösung erfordert Spezialwissen, bietet nur eine geringe Flexibilität gegenüber nachträglichen Änderungen und es ist mit Kompatibilitätsproblemen zu bereits existierenden Programmen zu rechnen.

2. Um diese Nachteile zu vermeiden, kann als zweite Möglichkeit die C-Standardbibliothek als Schnittstelle verwendet werden. Ihre Ein- und Ausgabe-Schnittstelle beispielsweise ist in `stdio.h` deklariert und macht etwa ein Drittel der Bibliothek aus. Bei ihrer Verwendung ist gewährleistet, daß die volle Funktionalität des C-I/O Systems zur Verfügung steht. Nachträgliche Änderungen sind ausgeschlossen, da auch auf Unix-Seite nicht mehr vorhanden ist als eben der C-Sprachumfang bietet. Über die Ein-/Ausgabe hinaus können weitere Teile der Standardbibliothek nach Bedarf ebenfalls hinzugezogen werden. Ein Applikationsprogrammierer benötigt kaum Zusatzkenntnisse für die Programmentwicklung in dieser Umgebung, da sie voll dem Standard entspricht. Hier werden ebenfalls der Benutzercode und die Bibliothek cross-compiliert, gelinkt und dann als eine ausführbare Datei auf das Board geladen.

   Die C-Standardbibliothek kann dabei aber nicht unverändert übernommen werden, da ihre Basis von einer Reihe von Systemaufrufen gebildet wird und auf dem 68k-Rechner kein Betriebssystem und keine Peripherie vorhanden sind. Sie muß also an die spezielle Hardwareumgebung angepaßt werden. Dienste des Betriebssystem stehen nur auf dem Unix-Host zur Verfügung. Bei der Anpassung der Bibliothek muß man daher einen Weg finden, die Bibliotheks-Aufrufe in dem Board-Programm über den VME-Bus umzuleiten und von einem Service-Prozeß auf dem Host bearbeiten zu lassen. Hier gibt es im Prinzip zwei Wege.

   (a) Man schreibt eine eigene Version aller benötigten Funktionen aus der Standardbibliothek. Wird eine solche Funktion aufgerufen, so löst sie eine VME-Bus Kommunikation aus. Dabei übergibt sie ihre Parameter an den Unix-Host, der dann seinerseits die identische C-Funktion mit den Parametern aufruft.

   Der Vorteil dieser direkten Lösung ist, daß kein Wissen über das Innenleben der C-Standardbibliothek erforderlich ist und sehr schnell eine erste Version mit den wichtigsten Funktionen implementiert werden kann. Bei Bedarf können weitere Teile hinzugefügt werden. Dabei sind keine Kompatibilitätsprobleme zu erwarten, da die Bibliothek höchstens erweitert wird, vorhandene Funktionen aber nicht verändert werden. Dem

stehen aber mehrere Nachteile gegenüber. Um der o.g. Anforderung nach allgemeiner Verwendbarkeit gerecht zu werden, muß ein großer Teil der Funktionen, Typen und Makros implementiert werden. Damit steigt der Entwicklungsaufwand rasch an, und die Schnittstelle über den VME-Bus wird unübersichtlich. Außerdem ist für jeden Bibliotheksaufruf eine VME-Kommunikation erforderlich. Die Ein-/Ausgabe über den VME-Bus ist also nicht gepuffert, und ein Teil der benötigten Rechenleistung wird von den lokalen Boards auf den VME-Host verlagert.

(b) Da die Funktionen der C Standard Ein-/Ausgabe aufeinander aufbauen, liegt der Gedanke nahe, nur wenige Basisfunktionen neu zu implementieren, womit dann die „höheren“ Funktionen unverändert übernommen werden können. Dies führt zu dem zweiten, wesentlich eleganteren und effizienteren Weg, der Neuimplementierung der Systemaufrufe (siehe Abschnitt 5.1.3 für genauere Erläuterungen). Dabei wird der Code der Standardbibliothek weitgehend unverändert übernommen. Lediglich die Schnittstelle zum Betriebssystem, die zum Beispiel für `stdio.h` aus nur 6 Systemaufrufen besteht, wird neu implementiert. Die Systemaufrufe werden stellvertretend von einem Service-Prozeß auf dem Unix-Host ausgeführt. Es wird also ein Teil der Unix API über einen RPC-Mechanismus via VME-Bus auf die 68k-Rechner exportiert. Dies ist vollständig transparent für die Applikationsentwicklung und -benutzung.

3. Als dritte Alternative bietet sich der Kauf und die Installation eines kompletten Betriebssystems auf den Board-Rechnern an. Insbesondere die in jüngster Zeit entstandenen Systeme mit Microkernel Architektur erscheinen hier interessant [54] [6]. Ein Microkernel enthält nur eine möglichst kleine Menge essentieller Systemfunktionen. Der Microkernel stellt aber keine API, sondern eine Schnittstelle für Systemprogramme SPI [2] zur Verfügung. Er bildet damit die Basis für die Realisierung aller anderen Systemdienste. Diese werden von Systemprozessen wahrgenommen, die auf Benutzerebene ausgeführt werden. Der Vorteil dieser Architektur liegt in der hohen Modularität und Flexibilität. Beim Einsatz eines Microkernels werden nur die wirklich benötigten Dienste realisiert. Einzelne Dienste können an benutzerspezifische Anforderungen, wie zum Beispiel eine spezielle Hardwareumgebung, angepaßt werden. In unserem Fall müßte bei einer Kernel Lösung ein Treiber für die Kommunikation über den VME-Bus als Systemdienst implementiert werden. Was die Ein-/Ausgabe betrifft, ist daher eine Reduktion des Implementierungsaufwandes nicht zu erwarten.

   Eine der Hauptaufgaben beim Kauf eines Kernel ist die Auswahl des richtigen Produkts. Hier gibt es rund 40 Betriebssystemanbieter, von denen etwa die Hälfte Versionen für die Motorola 680x0 Prozessoren anbietet [7].

[2] System Programming Interface

Die technischen Anforderungen an eine Laufzeitumgebung für Board-Programme können mit allen drei Konzepten erfüllt werden. Daher beruht die folgende Auswahl im wesentlichen auf den Kriterien Entwicklungsaufwand, Bedienbarkeit und Kosten.

Die Vorteile beim Kauf eines Kernels liegen hauptsächlich in den umfangreichen Möglichkeiten, die ein solches System bietet, da es sich um ein vollwertiges Betriebssystem handelt (zum Beispiel Multitasking und virtueller Speicher). Da diese aber für die FLT-Software nicht unbedingt benötigt werden, kommen diese Vorteile nicht zum Tragen. Ihnen steht zudem ein hoher Aufwand für die Auswahl, die Einarbeitung in ein neues Betriebssystem und die Implementierung der hardwarespezifischen Teile gegenüber. Hinzu kommt die Abhängigkeit von dem Hersteller-Support bei Problemen und der Preis. In der Literatur wird von zum Teil katastrophalem Support berichtet [29]. Die Kosten für die etwa 80 benötigten Installationen können je nach System incl. Support während der Laufzeit des Experiments eine Größenordnung von 80000 DM erreichen[29]. Zudem benötigen die meisten Systeme zusätzliche Hardware auf den Boards, wie zum Beispiel eine MMU.

Die Entscheidung fiel daher im wesentlichen zwischen einer eigenen VME-Kommunikationsbibliothek und einer Anpassung der C-Standardbibliothek. Die VME-Kommunikationsbibliothek zeichnet sich durch den geringsten Entwicklungaufwand, aber schlechtere Bedien- und Wartbarkeit aus. Eine angepaßte C-Standardbibliothek dagegen ist sehr einfach zu bedienen und benötigt voraussichtlich kaum Wartung. Ihre Anpassung erfordert aber zu Beginn einen deutlichen Mehraufwand.

Insgesamt fiel die Entscheidung zugunsten einer Spezialversion der C-Standardbibliothek mit angepaßten Systemaufrufen, auch weil der Entwicklungsaufwand aufgrund der bereits vorliegenden Erfahrungen genau abschätzbar war[3]. Es ist zu erwarten, daß der anfängliche Mehraufwand aufgrund weniger Änderungen und geringerem Aufwand bei den Applikationsentwicklungen wieder ausgeglichen wird.

### 5.1.3 Das Prinzip der Laufzeitumgebung

Bei Unix-Systemen werden die Dienste des Betriebssystems den Anwendungen über eine Schnittstelle für Anwendungsprogramme zur Verfügung gestellt (API). Sie wird von einer Sammlung von Funktionsaufrufen in den Unix-Kern gebildet. Diese Systemaufrufe sind auf allen UNIX-Systemen in C-Syntax definiert. Auch die C-Standardbibliothek enthält gleichnamige Funktionen zu den Systemaufrufen [52]. Ein Benutzerprozeß kann unter Verwendung der Konventionen von C diese Funktionen aufrufen, die dann ihrerseits die entsprechenden Systemaufrufe aktivieren (siehe Abbildung 5.2). Er kann dabei sowohl direkt auf einen Systemaufruf zugreifen, als auch eine Bibliotheksfunktion aufrufen. Viele „höhere“ Bibliotheksfunktionen basieren ihrerseits wieder auf Systemaufrufen.

[3]Bei den Projekten Nerv und Synapse wurde das gleiche Prinzip verwendet (siehe auch Abschnitt 5.1.4).

Die Laufzeitumgebung für Programme auf den FLT-Boards exportiert nun einen Teil der Unix-API auf die VME-Boards. Hierfür wird eine zusätzliche Schicht zwischen den Systemaufrufen und dem UNIX-Betriebssystem eingefügt. Diese Schicht kapselt den VME-Bus, so daß es sowohl aus Sicht der Anwendung (bzw. dem Anwender), als auch aus Betriebssystemsicht scheint, als würde der Board-Prozeß auf dem UNIX-Host ablaufen. Auf der Board Seite wird die Schicht von speziellen Versionen der Systemaufrufe und einem Systemcall-Handler gebildet. Auf der Host Seite läuft ein Stellvertreterprozeß für den Board-Prozeß („trun“). Das Protokoll für die VME-Bus Kommunikation wird zwischen dem Systemcall-Handler und `trun` abgewickelt. Die Board Systemaufrufe übergeben dabei ihre Parameter an den `trun` Service Prozeß, der sie dann stellvertretend auf dem Unix-Host aufruft und ihre Rückgabewerte an den Board-Prozeß zurück liefert.

Zudem muß nach den Anforderungen auch das Betriebssystem die Möglichkeit haben, eine Kommunikation mit dem Board-Prozeß auszulösen. Über das Unix Signalsystem kann ein Prozeß asynchron von dem Betriebssystem angesprochen werden. Unix-Signale sind Software Interrupts, die in den Signalbehandlungsroutinen eines Prozesses abgearbeitet werden. Die `trun` Signalbehandlungroutine hat dabei lediglich die Aufgabe, das Signal an den Board-Prozeß weiterzureichen, damit dieser seinerseits eine Routine aufruft, um auf das Signal zu reagieren.

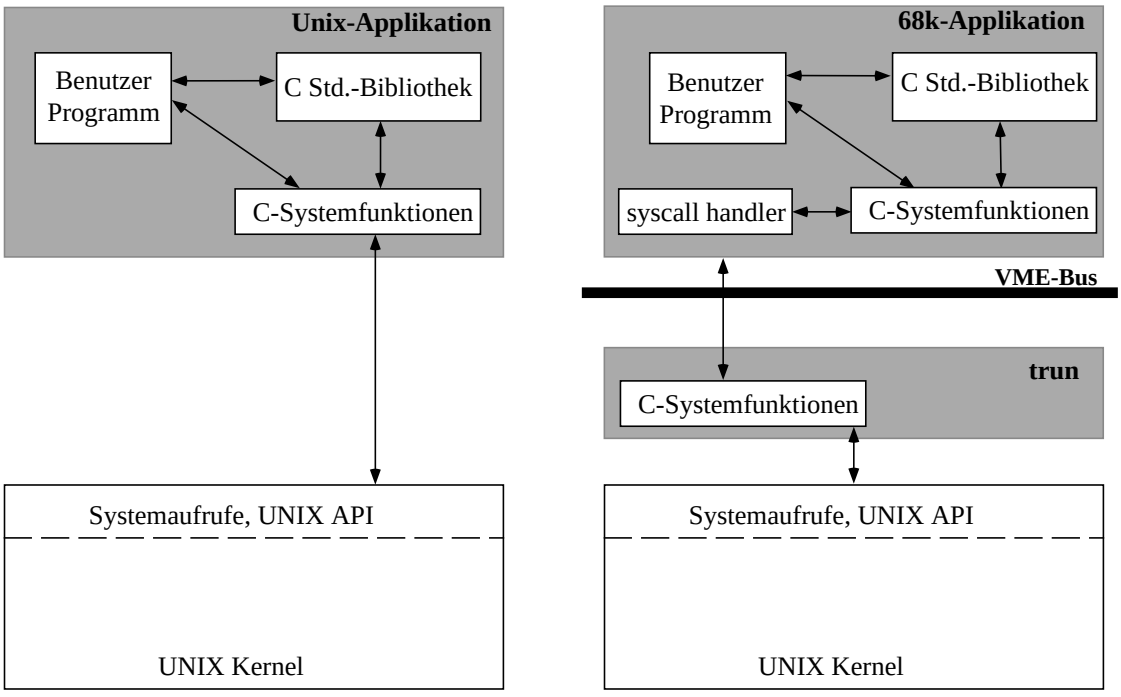


Abbildung 5.2: *Export des Unix-API auf die Prozessorboards.*

Bereits zwei Vorgängerprojekte verwenden das gleiche Prinzip wie die First Level Trigger Laufzeitumgebung, um C-Programme auf VME-Boards ablaufen zu lassen, der Nerv [20] [21] [12] und die Synapse [46] . Es gibt bei dem FLT allerdings wesentliche Unterschiede in der Hardware und Software des VME-Systems gegenüber diesen beiden Vorgängern, die untereinander praktisch identisch sind. Im folgenden Abschnitt wird die Soft- und Hardware-Architektur der FLT-Laufzeitumgebung beschrieben und dabei zu den beiden Vorgängerprojekten in Bezug gesetzt.

Die Laufzeitumgebung basiert auf vier grundlegenden Design Entscheidungen. Aufgrund etwas anderer Anforderungen und des aktuellen Stands der Technik unterscheidet sie sich dabei in allen vier Punkten von den beiden Vorgängerprojekten:

- Die Kommunikation über den VME-Bus wird grundsätzlich durch Interrupts ausgelöst.
- Als Kommunikationspuffer für die Abwicklung des Protokolls dient ein Teil des jeweiligen Board-Arbeitsspeichers.
- Jedem Board-Prozeß ist genau ein Unix-Prozeß zugeordnet.
- Es gibt über den VME-Bus keine Kommunikation der Board-Prozesse untereinander.

### 5.1.4 Unterschiede zu Nerv, Synapse

Der Hauptunterschied gegenüber den Nerv/Synapse-Systemen ist die Verwendung eines anderen Unix-Host Rechners, der sowohl vom Betriebssystem als auch von der Hardware her besser an die VME-Bus Umgebung angepaßt ist. Bei den Nerv/Synapse-Systemen wird ein Sparc II-Rechner von SUN mit einem SBus-/VME-Bus Interface als Host eingesetzt. Das Interface besitzt ein Dualported-RAM, das den Adreßräumen aller Prozesse angehört und als eine Art Shared Memory für den Kommunikationspuffer verwendet wird. Eine Kommunikation kann nur von einem 68k-Programm ausgelöst werden, indem es in den Puffer schreibt. Um das Eintreffen einer Nachricht von einem Board zu registrieren, muß der Host ständig über seinen Ein-/Ausgabe Port das Schnittstellen RAM abfragen.

Der FLT verwendet Rechner der Firma Cetia als VME-Hosts, die mit dem Echtzeit Unix Betriebssystem LynxOS betrieben werden, das Multithreading erlaubt. Sie sind als VME-Steckkarten ausgeführt und verwenden keinen VME-Bus Adapter, sondern besitzen selbst eine vollwertige VME-Schnittstelle.

Für Kommunikationsanfragen eines Boards wird ein Interrupt auf dem VME-Host ausgelöst. Da dies asynchron geschieht, muß `trun` nicht auf eine Anfrage warten, indem es auf einen Kommunikationspuffer pollt und damit Rechenleistung verbraucht. Der `trun`-Prozeß wird sogar in den Schlaf-Modus versetzt, so daß er praktisch überhaupt keine Rechenzeit mehr verbraucht. Für die Interruptroutine werden

die Multithreading Möglichkeiten von Lynx genutzt. Nachdem ein Board-Prozeß einen Interrupt ausgelöst hat, wird die Interruptroutine als eigener Thread abgearbeitet, während `trun` weiterschläft. Dadurch unterliegt die Interrupt Bearbeitung dem Prozeß-Scheduler und ist mit eigener, wählbarer Prozeßpriorität ausführbar.

Die Verwendung von Interrupts ermöglicht es auch, den Kommunikationspuffer in den Arbeitsspeicher des jeweiligen Boards zu legen, da kein Pollen des Hosts über den VME-Bus auf den Puffer notwendig ist. Die VME-Bus Belastung ist genau die gleiche wie wenn der Puffer im Host lokalisiert wäre, da der Host nur einmal nach Eintreffen eines Interrupts den Puffer liest. Es ist konzeptionell logischer, diesen Speicherbereich lokal auf einem Board zur Verfügung zu halten anstatt zentral, je nach Anzahl der vorhandenen Boards, eine Sammlung von Kommunikationspuffern anzulegen.

Die FLT-Laufzeitumgebung stellt zu der Möglichkeit, daß ein Board-Prozeß auf dem Host einen Interrupt auslöst, auch eine Host initiierte Kommunikation zur Verfügung, die ebenfalls mit Interrupts gesteuert wird. Der Host löst hierzu lokal auf dem Board (also nicht über die VME-Bus Interrupt Leitungen) einen Interrupt aus. Die Kommunikation wird anschließend analog zu der Board initiierten über den Kommunikationspuffer abgewickelt. Diese Art der Kommunikation wird benötigt, wenn von dem Benutzer oder einem zentralen Steuerprogramm Anfragen an das Board gestellt werden.

Bei den Nerv/Synapse-Systemen ist ein Host-Prozeß für alle Boards eines Crates zuständig. Es besteht also eine 1 : N Beziehung zwischen dem Kommunikationsprozeß und den Board-Prozessen. Die Laufzeitumgebung für den FLT ist so konzipiert, daß in einer 1 : 1 Beziehung für jedes Board ein eigener Prozeß auf dem Host gestartet wird. Dies hat mehrere Vorteile:

1. Engere Einpassung in das Unix Prozeß Konzept.
   Dieser wichtigste Vorteil erhöht die Transparenz bei der Programmierung und Benutzung der Boards. Jeder Board-Prozeß ist eindeutig mit einem Unix-Prozeß identifiziert und kann daher auch von Betriebssystem und Benutzer als solcher angesprochen werden. Beispielsweise können Unix-Signale an den `trun`-Prozeß gesendet, und dann an den Board-Prozeß weitergeleitet werden.

2. Vereinfachte Schnittstelle zwischen `trun`- und Board-Prozessen.
   Bei der Kommunikation mit Board-Prozessen muß nicht zwischen verschiedenen Kanälen separiert werden, da ein `trun`-Prozeß nur einem einzigen Board-Prozeß zugeordnet ist. Es gibt auch keinen Broadcast-Modus wie bei Nerv/Synapse, wo der Kommunikationprozeß via VME-Bus mit einem Befehl eine Nachricht an alle Boards schicken kann.

3. Einfachere Fehlersuche und höhere Fehlertoleranz.
   Tritt bei der Kommunikation mit den Boards ein Fehler auf, so ist er eindeutig einem Board zuzuordnen. Die anderen Prozesse bleiben davon in der Regel unberührt. Falls im Betrieb ein Prozeß ausfällt, können die anderen die Kontrolle über ihr Board weiter ausüben und nur dieser eine muß gegebenenfalls neu gestartet werden.

Der Datenaustausch zwischen den Boards des FLT wird von dem Message Transfer System übernommen. Für eine zusätzliche Kommunikation der Boards über den VME-Bus gibt es keine Notwendigkeit[4]. Eine derartige Kommunikation, wie sie bei Nerv/Synapse eingesetzt wird, ist daher bei der FLT-Software nicht vorgesehen.

## 5.2 Das Programm trun

Das Programm `trun` wird als Service-Prozeß auf dem Unix-Host gestartet. Es stellt dem Benutzer folgende Möglichkeiten zur Verfügung:

1. Lauffähige (cross-compilierte und -gelinkte) Programme können auf Boards geladen und gestartet werden.

2. Ein- und Ausgabefunktionen sowie andere Systemfunktionen eines Board-Prozesses werden ausgeführt.

3. Mit Hilfe eines interaktiven Modus kann der Status des Boards abgefragt werden. Außerdem wird eine Reihe von Testmöglichkeiten zur Verfügung gestellt.

4. Es kann eine Netzwerkverbindung zur Verfügung gestellt werden. Board-Prozesse können dann via Ethernet z.B. mit einem zentralen Steuerprogramm Daten austauschen.

Den Datenfluß bei Start und Ablauf eines Benutzerprogramms in einem System aus Unix-Host und einem Prozessorboard, hier eine TFU, zeigt Abbildung 5.3. Auf der Unix Seite gibt es `trun` als Service Prozeß, der Zugriff auf Peripherie wie z.B. Festplatte und Netzwerk hat. Auf dem Board gibt es zum einen den eigentlichen Hardware-Prozessor, der den Spursuche Algorithmus abarbeitet. Er hat als Eingangsdaten FLT-Messages und Detektordaten und als Ergebnis eine FLT-Message mit genaueren Spurinformationen. Weiterhin sind zwei Prozesse abgebildet, die (nacheinender) auf dem boardeigenen Mikroprozessor ablaufen und unabhängig parallel zu dem eigentlichen Triggeralgorithmus arbeiten.

Nach dem Einschalten des Boards wird aus dem EPROM das Programm root gestartet. Es verzweigt entweder in das boardeigene Monitorprogramm (nicht abgebildet) oder ein geladenes Benutzerprogramm. `Trun` lädt das Benutzerprogramm von der

[4]Es wäre ohnehin nur mit Einschränkungen möglich, da die Boards auf sechs Crates verteilt sind.

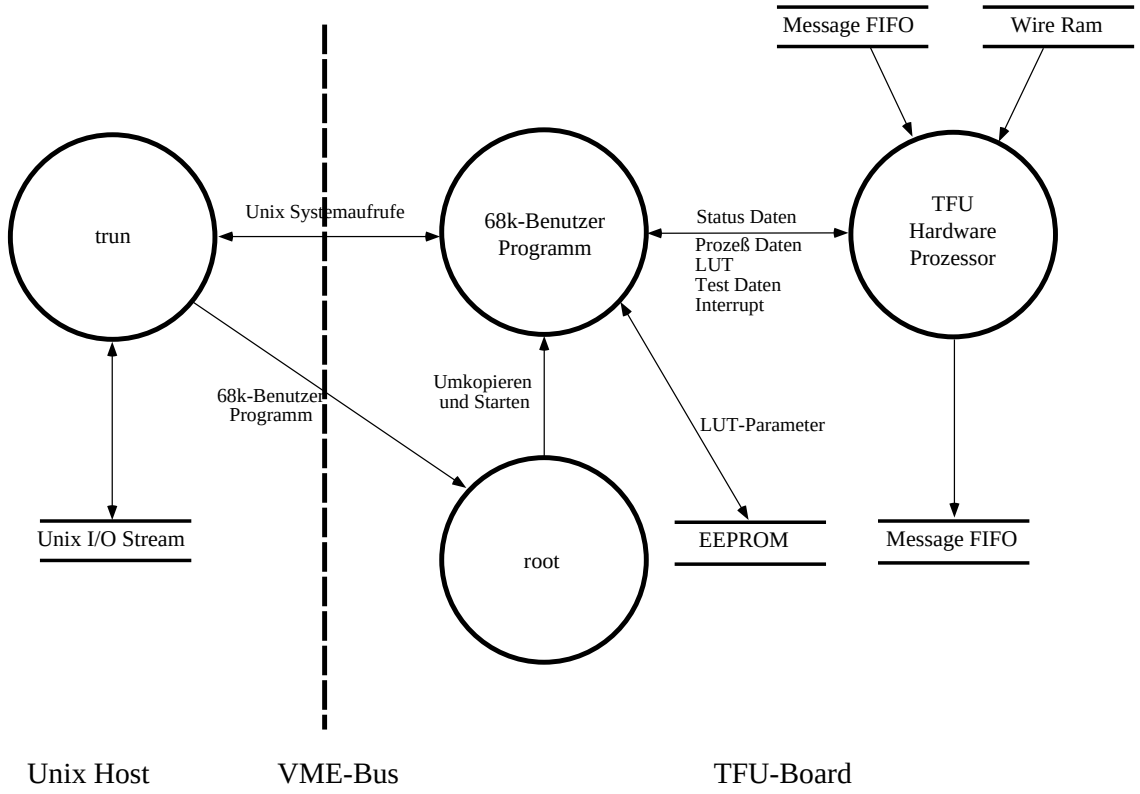


Abbildung 5.3: *Der Datenfluß zwischen Unix-Host und TFU-Prozessorboard.*

Festplatte über den VME-Bus in den Arbeitsspeicher des Boards. Dann wird root mitgeteilt, daß ein Programm geladen wurde. Root übernimmt das Programm und kann es, falls gewünscht, an eine andere Stelle im Arbeitsspeicher verschieben, zum Beispiel in einen Bereich, der über den VME-Bus nicht zugänglich ist. Befindet sich das Benutzerprogramm an der gewünschten Zieladresse, so springt root an dessen Anfang und startet damit das Programm. Nach Beenden des Programms wird die Kontrolle wieder von root übernommen.

Das laufende Benutzerprogramm kann, je nach Anwendung, schreibend und lesend auf die Register des Hardwareprozessors zugreifen. Beispielsweise können Lookup-Tables geladen, Message-Raten gelesen oder auch der Prozessor angehalten und einzelne Pipelinestufen ausgelesen oder beschrieben werden. Die Parameter für die LUT-Berechnung können lokal in einem EEPROM abgelegt werden, damit jedes Board die Tabellen berechnen kann ohne die Parameter von einer Festplatte laden zu müssen. Der Hardwareprozessor kann auch einen Interrupt auf dem 68k-Prozessor auslösen. Damit kann zum Beispiel auf Fehler reagiert werden oder können in festen Zeitabständen Statusinformationen an die Zentrale verschickt werden.

Im Folgenden wird näher beschrieben, wie ein Programm geladen und gestartet wird und wie die Kommunikation zwischen Unix-Prozeß und Board-Prozeß abläuft.

### 5.2.1 Laden und Starten eines Programms

Die C-Programme für die FLT-Boards werden mit den Cross- Entwicklungswerkzeugen von GNU unter Linux übersetzt und gebunden. Von dem Cross-Linker wird das ausführbare Programm als 68k-Binärdatei in dem a.out-Format auf Festplatte geschrieben und steht unter LynxOS dann zur Verfügung. Um ein Programm korrekt laden und ablaufen lassen zu können, muß `trun` neben dem Namen und den Parametern des Board-Programms auch mitgeteilt werden, in welchem Modus es selbst arbeiten soll und welcher Boardtyp bedient wird. Die Informationen, die zum Ablaufen eines Board-Programms nötig sind, werden `trun` auf zwei Arten zur Verfügung gestellt:

- Parameterliste
  In der Parameterliste von `trun` wird das Programm mit Flags an verschiedene Bedingungen angepaßt. Zusätzlich muß der Name einer Konfigurationsdatei angegeben werden, die Daten über das FLT-Board enthält. Darauf folgt dann der Name des zu startenden Board-Programms mit seinen Parametern.

- Konfigurationsdatei
  In der Konfigurationsdatei werden die Hardware des jeweiligen Boards und seine Betriebsparameter spezifiziert, beispielsweise die Adressen von Kontrollregistern, die Ausbaustufe des RAMs und die Positionen von Stack und Heap im Adreßraum des Boards.

Die Ausführung eines FLT-Programms `m68k_prog` wird auf dem Unix-Host durch

```
trun [Flags] config_file m68k_prog [parameterliste_des_m68k_prog]
```

gestartet. Dabei sind die `Flags` und die Parameter des `m68k_prog` optional. Das Laden und Starten des Programms läuft dann in folgenden Schritten ab:

1. Das Board wird zurückgesetzt.
   Dabei wird der Reset „gehalten“, das heißt die Reset Leitung bleibt auf 0 und der Prozessor wartet.

2. Das Programm wird an die durch die Konfigurationsdatei definierte Stelle kopiert.
   Hierzu wird die Binärdatei im a.out-Format von der Festplatte eingelesen. Der Header der Datei wird interpretiert und abhängig von der Header-Information der Code und das initialisierte Datensegment auf das Board kopiert.

3. Die Board-Programm-Parameter werden auf das Board kopiert.
   Zusätzlich berechnet `trun` die Zeigertabelle mit den lokalen Adressen für diese Parameter und legt sie an definierter Stelle ab. Das Board-Programm kann dann wie üblich über `argc` und `argv` darauf zugreifen.

4. Die Systemstruktur wird auf das Board kopiert.
   Die Systemstruktur ist an einer festen Adresse am Beginn des RAM-Speichers der Boards untergebracht. Sie enthält alle Informationen über die Konfiguration des Systems (im wesentlichen Adressen). Zudem wird die Kommunikation zwischen `trun` und dem Board-Prozeß über sie abgewickelt.

5. Die `MagicNumber` wird gesetzt.
   Sie dient als Flag, mit dem `root` (vgl. Abbildung 5.3), dem Startprogramm des FLT-Boards, mitgeteilt wird, daß sich ein Programm im Speicher befindet, das gestartet werden soll.

6. Der Reset wird weggenommen.
   Dann läuft das Startprogramm (`root`) an. Es überprüft das Vorhandensein der `MagicNumber`. Falls die Nummer vorhanden ist, wird das geladene Board-Benutzerprogramm aufgerufen. Im anderen Fall verzweigt der Prozessor in das Monitorprogramm, mit dem die serielle Schnittstelle bedient wird.

7. Die `MagicNumber` wird gelöscht.

Anschließend geht `trun` in eine Endlosschleife, und die Arbeit der main-Funktion von `trun` ist damit beendet. Kommunikationsanfragen an `trun` werden dann in Interruptroutinen oder Signalroutinen abgearbeitet. Der interaktive Modus beispielsweise läuft gegebenenfalls in einer Signalroutine ab, während Board-Anfragen in einer Interruptroutine bearbeitet werden.

### 5.2.2 Die Kommunikation zwischen dem Unix-Prozeß und dem Board-Prozeß

Da `trun` nur eine Lage zwischen Board-Prozeß und dem Unix Betriebssystem ist, sind die Informationen, die zwischen `trun` und dem Board-Prozeß ausgetauscht werden, eine Teilmenge dessen, was zwischen einem normalen Unix Prozeß und dem Betriebssystem ausgetauscht wird.

Ein Unix Prozeß nimmt über die Schnittstelle der Systemaufrufe Kontakt mit dem Betriebssystem auf. Dem Board-Prozeß wird eine Teilmenge dieser Schnittstelle über den VME-Bus zur Verfügung gestellt. Dabei macht es keinen Sinn, Befehle wie den Systemaufruf für das Anfordern von Speicher (`sbrk`) zu exportieren, da sich der Speicher des Board-Prozesses ja physikalisch auf dem Board befindet. Die Speicherverwaltung für das Board ist daher komplett in der FLT-C-Bibliothek enthalten. Ebenso macht es keinen Sinn, Systemaufrufe zur Prozeßsteuerung zu exportieren (zum Beispiel `fork`). Es hat sich gezeigt, daß es ausreicht, im wesentlichen die I/O bezogenen Systemaufrufe zu exportieren, wodurch dem Board-Prozeß die Peripherie des Unix-Systems zugänglich wird.

Das Betriebssystem wiederum kann über Signale einen Prozeß ansprechen. Signale sind Software Interrupts und ermöglichen die Behandlung asynchroner Ereignisse durch einen Prozeß. `Trun` kann Signale an ein Board weitergeben, indem es in seiner Signalroutine einen Interrupt des Board-Prozesses auslöst, so daß dieser in seiner Interruptroutine dann auf das Signal reagieren kann.

Die Einbindung eines Board-Prozesses durch `trun` in diese beiden Kommunikationssysteme wird im folgenden beschrieben. Für den normalen Benutzer und Entwickler von Board-Programmen ist die hier beschriebene Kommunikation völlig transparent.

### Unix Ein-/Ausgabe

Unix besitzt ein einheitliches Modell für Ein-/Ausgabe Operationen. Alle Arten von Ein-/Ausgabe werden über Dateien abgewickelt. Egal was für ein Gerät angesprochen wird, können daher einheitlich immer die gewöhnlichen Systemaufrufe wie zum Beispiel `read` und `write` verwendet werden ([52], [54]). Alle diese Funktionen arbeiten mit Dateidescriptoren, die der Systemkern für jede geöffnete Datei vergibt. Man bezeichnet diese Funktionen auch als Funktionen zur ungepufferten Ein-/Ausgabe, da jeder Aufruf direkt einen Systemaufruf in den Kern auslöst.

In der C-Standardbibliothek ist eine gepufferte Ein-/Ausgabe realisiert, die auf den oben genannten Systemfunktionen basiert. Die Bibliothek stellt für jede Datei einen Puffer zur Verfügung. Ausgaben in eine Datei werden solange zwischengespeichert, bis ein Befehl zum Leeren des Puffers erfolgt, der dann eine Ausgabe des gesamten Pufferinhalts mit einem `write`-Aufruf zur Folge hat. Ziel der Pufferung ist, die Anzahl der erforderlichen `read-` und `write-`Aufrufe so gering wie möglich zu halten, da ein Systemaufruf verhältnismäßig viel Rechenzeit in Anspruch nimmt.

Dies gilt um so mehr für die Board-Laufzeitumgebung, da hier jeder Systemaufruf noch zusätzlich mit einer VME-Bus Kommunikation verbunden ist. Die Pufferung der Standardbibliothek korrespondiert daher sehr gut mit den Erfordernissen der Laufzeitumgebung. Hier wird klar, daß es am effizientesten ist, die Systemaufrufe auf das Board zu exportieren, da man dann automatisch von den Eigenschaften der Bibliothek profitiert, teure Systemaufrufe zu minimieren.

### Die Schnittstelle für Systemaufrufe

Im einzelnen sind für die Ein-/Ausgabe folgende Systemaufrufe in die Schnittstelle zwischen `trun` und einem Board-Prozeß aufgenommen worden:

| | |
|---|---|
| `creat` | Erzeugen einer Datei |
| `open` | Öffnen und gegebenenfalls Erzeugen einer Datei |
| `close` | Schließen einer Datei |
| `read` | Lesen aus einer Datei |
| `write` | Schreiben in eine Datei |

| | |
|---|---|
| `lseek` | Positionierung des Dateidescriptors |
| `rename` | Umbenennen einer Datei |
| `unlink` | Löschen einer Datei |
| `stat` | Status-Informationen über eine Datei lesen |

Hinzu kommen weitere, nicht I/O bezogene Funktionen:

| | |
|---|---|
| `exit` | Beende `trun` normal |
| `abort` | Anormale Beendigung von `trun` |
| `getenv` | Zugriff auf Unix Umgebungsvariablen |
| `system` | Ausführen eines Unix Kommandos |
| `time` | Liefere die aktuelle Zeit in Sekunden |
| `strerror` | Liefere Zeiger auf Fehlerzeichenkette |
| `irAccept` | Informiere `trun` über einen Board-Interrupt |
| `irDone` | Informiere `trun` über Beendigung eines Board-Interrupts |
| `raise` | Sende Signal an das Programm |
| `exeptionHandler` | Übergebe Exeption-Informationen von Board zu `trun` |

Da auf dem Board kein vom Programm getrenntes Betriebssystem vorliegt, sind alle oben genannten Funktionen dort als Bibliotheksfunktionen implementiert und in die FLT-Version der C-Standardbibliothek integriert. Wird eine solche Funktion aufgerufen, so ruft sie ihrerseits den Systemcall-Handler auf (siehe auch Abbildung 5.2). Der Systemcall-Handler löst einen Interrupt von `trun` aus und wickelt dann das synchrone Kommunikationsprotokoll (Abbildung 5.4) mit `trun` ab. Die Funktionen werden teilweise in 5.3 näher beschrieben.

### Die Systemstruktur

Die Systemstruktur ist so im Arbeitsspeicher des Boards untergebracht, daß sie für beide Seiten erreichbar ist (vergleiche Abbildung 5.7). Sie stellt also eine Art Shared Memory zwischen Board-Prozeß und `trun` dar und hat drei Aufgaben:

Einmal wird in ihr die Konfiguration der Laufzeitumgebung abgelegt. Hierfür werden ein Großteil der Informationen aus der Konfigurationsdatei in die Systemstruktur kopiert. Das Schreiben der Systeminformation vor dem Start des Board-Prozesses legt die Speichereinteilung, die Adressen des Heaps, Stacks etc., auf dem Board fest.

Zum zweiten enthält sie die von den jeweiligen Applikationen benötigten Systemvariablen zum Ablegen von Statusinformationen zur Laufzeit und für das Speicher-Management.

Weiterhin dient die Struktur als Schnittstelle für das Kommunikationsprotokoll. Das Schreiben der Systeminformation vor dem Start des Board-Prozesses initialisiert die Kommunikation. Anschließend benutzt der Systemcall-Handler die Struktur für die

Ablage der Parameter und Rückgabewerte von Systemaufrufen und zur Abwicklung der Kommunikationprotokolle mit `trun`, das die Parameter liest und die Kommandos ausführt. Die Rolle der Struktur kommt in den folgenden Protokollbeschreibungen noch deutlicher zum Ausdruck.

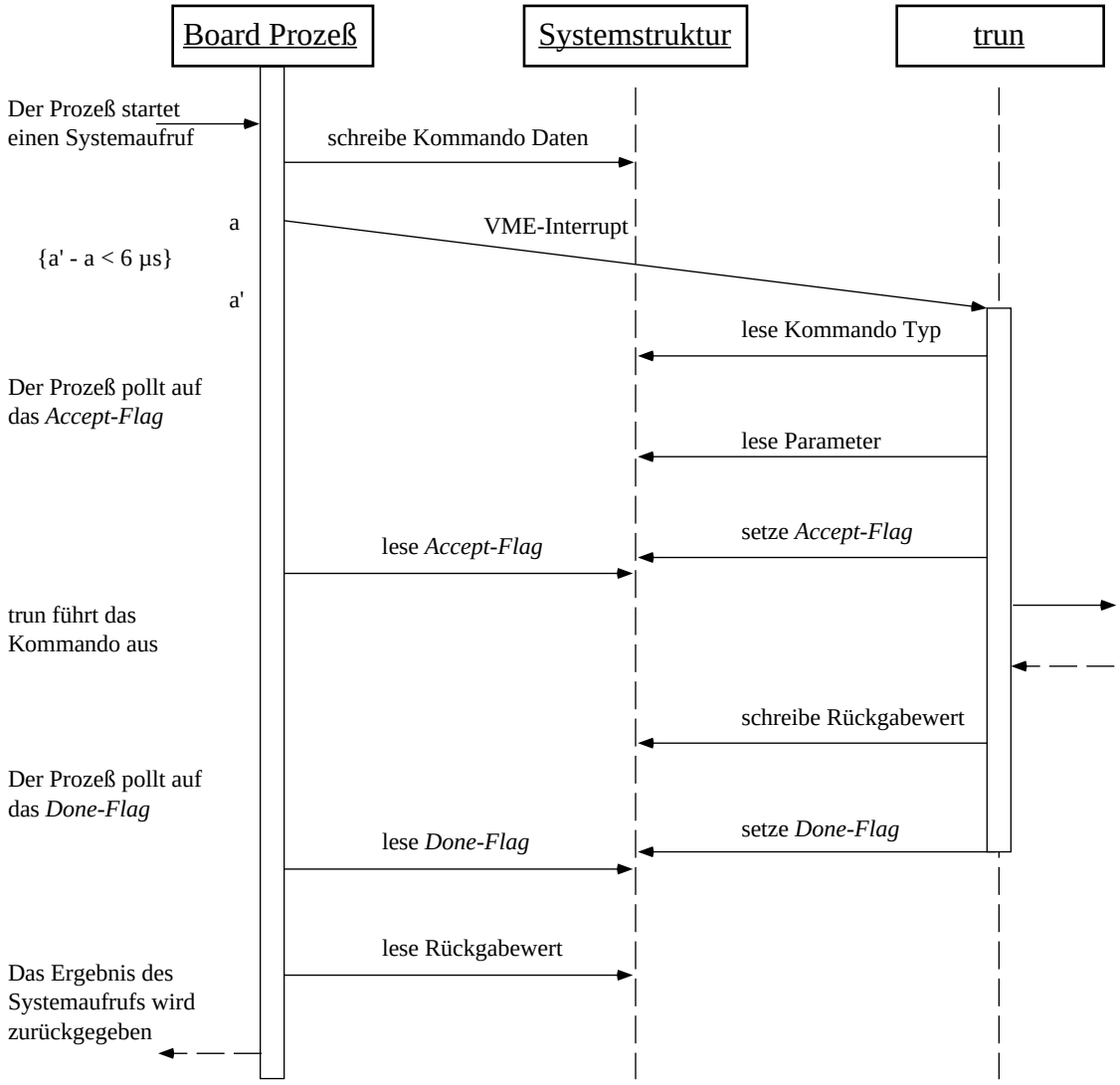


Abbildung 5.4: *Sequenzdiagramm des Kommunikationsprotokolls zwischen einem Prozeß auf einem Prozessorboard und* `trun`.

### Das Protokoll

Immer dann, wenn ein Systemaufruf erfolgt, ist von der Board Seite eine Kommunikation mit `trun` erforderlich. Der zeitliche Verlauf des zugehörigen Kommunikationsprotokolls ist in dem Sequenzdiagramm in Abbildung 5.4 dargestellt. Die

beiden Prozesse und die Struktur, über die sie ihr Protokoll abwickeln, werden dabei als Objekte aufgefaßt, die untereinander Nachrichten austauschen. Die Zeitachse verläuft nach unten und die Nachrichten zwischen den drei Beteiligten sind durch Pfeile dargestellt.

Ein erfolgter Systemaufruf des Board-Prozesses führt als erstes eine Initialisierung durch, indem der Typ und die Parameter des Aufrufs in die Systemstruktur eingetragen werden. Dann wird ein VME-Interrupt ausgelöst, der von `trun` in einem Thread abgearbeitet wird. Dieser liest, um welchen Systemaufruf es sich handelt, liest dessen Parameter, setzt daraufhin das *Accept-Flag* und führt den Aufruf aus. Wenn er von dem Aufruf in den Systemkern zurückkehrt, wird der Rückgabewert in die Struktur geschrieben und mit dem Setzen des *Done-Flag* die Beendigung dem Board-Prozeß mitgeteilt. Der Board-Prozeß pollt zwischenzeitlich, nachdem er zuerst auf das *Accept-Flag* gepollt hat, auf das *Done-Flag*. Wenn es gesetzt wird, liest er den Rückgabewert seines Systemaufrufs aus der Struktur. Dieser wird dann von dem lokalen Systemaufruf zurückgegeben, der damit dann beendet ist.

### Die Host initiierte Kommunikation

Die Initiative für eine Kommunikation zwischen `trun` und dem Board-Prozeß kann auch von `trun` ausgehen, indem es auf dem Board einen Interrupt auslöst. Die Kommunikation wird ebenfalls über die Systemstruktur zwischen der Board-Interruptroutine und `trun` abgewickelt, natürlich mit einem anderen Protokoll. Momentan wird die Host initiierte Kommunikation für die Weiterleitung von Signalen und den interaktiven Modus (siehe Abschnitt 5.2.3) benutzt.

Der Board-Prozeß kann auf zwei Arten unterbrochen werden: Bei der asynchronen Unterbrechung wird ohne Rücksicht auf den Zustand des Board-Prozesses sofort ein Interrupt ausgelöst. Falls sich der Board-Prozeß gerade in einer kritischen Programmregion befindet und die Interrupt Routine nun ebenfalls solche kritischen Programmregionen aufruft, so kann dies zu dem Absturz des Systems führen. Kritische Programmregionen sind solche, in denen der Prozeß gerade auf Systemresourcen zugreift. Die Interrupt Routine einer asynchronen Unterbrechung darf deshalb keine Aufrufe von Funktionen für Ein-/Ausgabe, Speicheranforderung, Kommunikation und Signalbehandlung enthalten. Dies ist verboten, weil der Board-Prozeß vor der Unterbrechung unter Umständen ebenfalls gerade eine solche Funktion in Anspruch nimmt. Bei Mißachtung könnte zum Beispiel die VME-Bus Kommunikation durcheinandergeraten oder Speicher doppelt angefordert werden. Zusätzlich muß beachtet werden, daß der Board-Prozeß sich nicht bereits in einer Interruptroutine befindet, da dies zu den gleichen Problemen führen würde. Das heißt, Interrupts dürfen auch nicht geschachtelt werden.

Die synchrone Unterbrechung berücksichtigt diese Problematik, indem sie den Zustand des Board-Prozesses abfrägt und gegebenenfalls wartet, bis er sich in einem nicht kritischen Zustand befindet. In der Systemstruktur befinden sich hierzu zwei

Statusflags, die jeweils von dem Board-Prozeß gesetzt werden, wenn er auf Systemresourcen zugreift oder sich in einer Interruptroutine befindet. In der Interruptroutine einer synchronen Unterbrechung können daher fast alle Systemfunktionen verwendet werden.

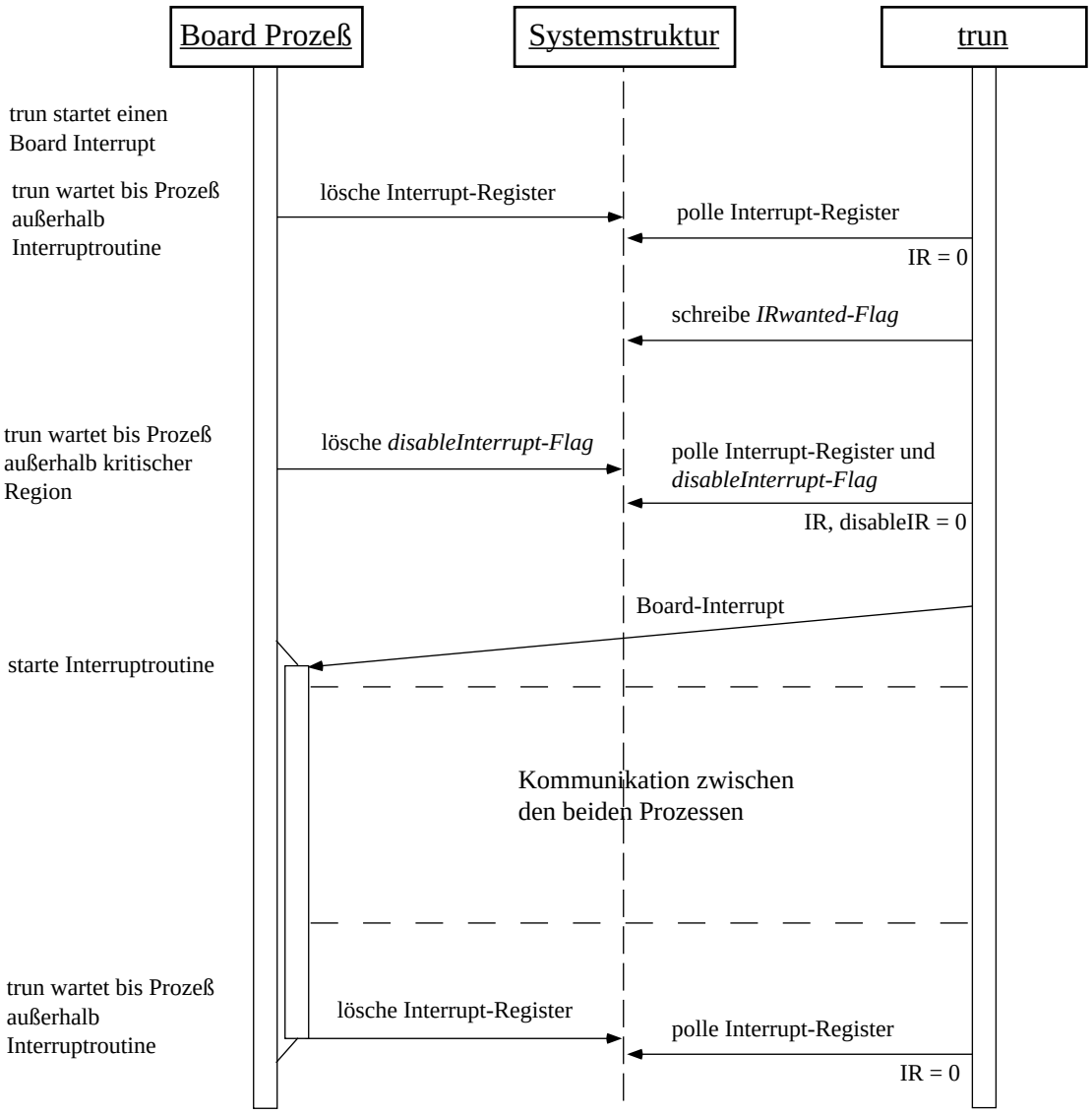


Abbildung 5.5: *Sequenzdiagramm des Kommunikationsprotokolls bei einer synchronen Unterbrechung des Board-Prozesses durch* `trun`.

Abbildung 5.5 gibt das Protokoll einer synchronen Unterbrechung wieder. `Trun` wartet zuerst ab, bis sich der Board-Prozeß außerhalb einer Interrupt-Routine befindet, indem es auf das Interrupt-Register pollt. Dann setzt es ein Flag, das dem Board-Prozeß signalisiert, daß ein Interrupt gewünscht wird. Der Board-Prozeß geht dann nicht mehr in kritische Programmregionen, sondern wartet vorher auf das Löschen

dieses Flags. `Trun` wartet nun ab, bis sich der Board-Prozeß nicht mehr in einer Interruptroutine und auch nicht mehr in einer kritischen Programmregion befindet. Wenn dies der Fall ist, löst `trun` einen Interrupt auf dem Board aus. Da `trun` vorher sichergestellt hat, daß sich der Board-Prozeß in einem definierten Zustand befindet, kann nun zwischen den beiden eine beliebiges Protokoll, zum Beispiel zur Signalbehandlung, abgewickelt werden. Während des Protokollablaufs kann problemlos auf Systemresourcen zugegriffen werden. Bei Beendigung der Interruptroutine wird das Flag, das den Interrupt-Modus des Boards signalisiert, gelöscht, und beide Prozesse können nun ihren Ablauf normal fortsetzen.

### 5.2.3 Der interaktive Modus

Das `trun` Programm besitzt einen zweiten Betriebsmodus, den sogenannten interaktiven Modus, der eine einfache kommandozeilenorientierte Schnittstelle zur Verfügung stellt. Er ist weniger für den normalen Benutzer gedacht, als für das Testen des Systems in einfachen Fällen. Man gelangt in diesen Modus entweder direkt durch Start von `trun` ohne Angabe eines Programmnamens oder durch Drücken der Control-C Taste während des normalen Programmablaufs.

Um `trun` während des Ablaufs in den interaktiven Modus versetzen zu können, muß zusätzlich das `trun`-Flag '-t' („trace“) gesetzt werden. Je nachdem ob das Flag gesetzt wurde oder nicht, wird ein unterschiedlicher Signal-Handler für `SIGINT` geladen. Das Drücken von Control-C löst unter Unix das Signal `SIGINT` aus. Je nach Flag Status wird jetzt dieses Signal

- zum Übergang in den interaktiven Modus genutzt oder
- zum Board-Programm weitergesendet.

Im interaktiven Modus wird ein Kommandozeileninterpreter aktiviert. Er zeigt einen Prompt mit dem Namen des Boards an. Hier werden verschiedene Kommandos als Strings von der Standardeingabe eingelesen.

Das folgende Beispiel gibt von einem Board, das den Namen TFU102 trägt, einen Hexdump mit einer bestimmten Länge und ab einer gewünschten Adresse aus:

```
TFU102> read <Adresse>  <Länge>
```

Der Modus wird durch den Befehl `exit` verlassen. Der FLT-Prozeß läuft danach weiter. Wird erneut Control-C gedrückt, geht der Prozeß wieder in den interaktiven Modus. Durch das einfache Hin- und Herschalten läßt sich bei Störungen jederzeit der Systemzustand feststellen.

## 5.3 Die FLT-C-Bibliothek

Jedes Programm, das auf einem FLT-Board ablaufen soll, muß die FLT-C-Bibliothek einbinden. Die Bibliothek enthält eine fast vollständige Version der C-Standardbibliothek. Aber auch wenn man deren Funktionen nicht benutzt, muß die FLT-C-Bibliothek hinzugelinkt werden, da sie noch weitere Funktionalität enthält. Wie in 5.2.1 beschrieben, wird beim Starten eines Programms zuerst die Laufzeitumgebung in Abhängigkeit von der Konfigurationsdatei und den `trun`-Parametern initialisiert. Anschließend wird die Board-Applikation geladen und gestartet. Da es auf dem Board kein Betriebssystem gibt, muß der geladene Programmcode neben der benutzerdefinierten Funktionalität auch alle notwendigen Laufzeitfunktionen enthalten. Sie müssen nach dem Übersetzen des Benutzerprogramms hinzugelinkt werden. Diese Funktionen, die betriebssystemähnliche Aufgaben erfüllen, sind ebenfalls Bestandteil der an die Board-Hardware angepaßten FLT-C-Bibliothek. Hinzu kommt noch Code, der vor und nach dem Ablauf des eigentlichen Benutzerprogramms abgearbeitet werden muß. Man kann daher zwei Funktionsgruppen in der FLT-C-Bibliothek unterscheiden, auf die im folgenden eingegangen wird:

1. Funktionen der C-Standardbibliothek, die vom ANSI-C-Standard [5] definiert werden.
2. Funktionen, die die Laufzeitumgebung benötigt.

Im folgenden werden die Standard-Funktionen und die Funktionen der Laufzeitumgebung angesprochen. Anschließend wird auf die Abläufe vor und nach dem Aufruf der `main`-Funktion des Benutzerprogramms und auf seine Speicherverwaltung näher eingegangen.

### 5.3.1 Funktionen der Standardbibliothek

Die FLT-C-Standardfunktionen [26] sind in folgenden Header-Dateien deklariert:

```
assert.h    ctype.h    error.h    setjmp.h    signal.h
stdarg.h    stdio.h    stdlib.h   string.h    time.h
```

Ein großer Teil der Funktionen ist nicht von der Hardware oder der Programmumgebung abhängig und konnte daher unverändert in die FLT Version übernommen werden. Hierzu zählen zum Beispiel alle Stringbearbeitungsfunktionen in `string.h`.

Die FLT-C-Bibliothek deckt den größten Teil der ANSI-C-Standard Bibliothek ab. Nicht implementiert wurden multi-byte Funktionen und Funktionen, die von lokalen Sprachen, Nationalität oder Kultur abhängen. Auch die `clock`-Funktion und einige Mathematik Funktionen sind nicht verfügbar. Die wesentlichen Zeit- und Signal-Funktionen werden unterstützt. Im Vergleich zur Nerv/Synapse-Implementation ist

[5] American National Standard X3.159-1989 [3]

der Umfang der verfügbaren Funktionen erheblich erweitert worden. Hier soll nur auf einige implementationsabhängige Funktionen näher eingegangen werden.

### Funktionen mit Bezug zu der Prozeßumgebung

Als Prozeßumgebung eines Board-Prozesses wird die Umgebung seines `trun`-Prozesses auf dem Host angesehen. Die beiden Funktionen, die mit der Umgebung kommunizieren, sind daher Bestandteil der Kommunikationsschnittstelle (siehe Abschnitt 5.2.2) und werden von `trun` ausgeführt.

Die Funktion `getenv`, von dem Board-Prozess aufgerufen, nimmt daher Bezug auf die Umgebungsvariablen von `trun` auf dem Host. Der Umgebungsstring wird hierzu auf das Board kopiert und `getenv` liefert dann einen lokalen Zeiger auf das erste Zeichen zurück.

Dementsprechend führt die Funktion `system` ihr übergebene Kommandos auf dem Host aus.

### Globale Sprünge

In der Header-Datei `setjmp.h` werden die Funktionen für nicht-lokale Sprünge definiert. Mit deren Vereinbarungen kann man in C die normale Folge von Funktionsaufruf und -ende umgehen. Die Funktion `setjmp` wurde so implementiert, daß sie die Daten- und Adreßregister und den Programmzähler der M680x0-CPU sichert.

Bei Aufruf der `longjmp`-Funktion wird der mit `setjmp` gespeicherte Prozessorzustand wieder geladen, insbesondere wird der Stack rekonstruiert. Das Programm wird hinter dem Aufruf von `setjmp` fortgesetzt.

### Zeit- und Datums-Funktionen

In der Header-Datei `time.h` sind Zeit- und Datumsfunktionen und deren Konvertierung in verschiedene Formate definiert. Die Bibliotheksfunktion `time` wurde ebenfalls in die Kommunikationsschnittstelle zu `trun` aufgenommen. Sie liefert die Zeitangabe `time_t`, die die Zeit in Sekunden seit dem 1. Januar 1970 in einem 32-Bit breiten Langwort angibt. Die Struktur `struct tm` beinhaltet die Sekunden, die Minuten, die Stunden, den Wochentag, den Tag des Monats, den Monat selbst, die Anzahl der Jahre (seit 1900) und die Anzahl der Tage im Kalenderjahr. Die Anpassung der lokalen Zeit wird durch zwei Umgebungsvariablen von `trun` definiert:

`LOCALTIME` gibt die Verschiebung der lokalen Zeit zur Coordinate Universal Time (UTC bzw. GMT) in Minuten an. Beispielsweise ist die Mitteleuropäische Zeit (CET) der UTC um eine Stunde voraus. Somit muß der Umgebungsvariablen ein Wert von +60 zugewiesen werden.

`LOCALTIMEZONE` stellt eine Zeichenkette dar, die die lokale Zeitzone beschreibt (beispielsweise „CET“).

### 5.3.2 Funktionen der Laufzeitumgebung

#### Systemaufrufe

Die Systemaufrufe, die den Zugriff des Board-Programms auf die Betriebssystemfunktionen des Host ermöglichen, bilden die wichtigste Gruppe der hardwareabhängigen Funktionen. Für jeden Systemaufruf ist eine gleichnamige Funktion in der FLT-C-Standard-Bibliothek definiert. Hinter diesen Funktionen verbirgt sich im Fall der FLT-Bibliothek aber nicht das Betriebssystem, sondern die VME-Bus Kommunikation. Die Systemaufrufe wurden bereits in Abschnitt 5.2.2 in Zusammenhang mit der Kommunikationsschnittstelle aufgelistet.

#### Die Ausnahme Behandlung

Wenn im Programmablauf ein nicht zu lösendes Problem, zum Beispiel eine Division durch Null oder ein Zugriff auf eine nicht vorhandene Adresse, auftritt, so reagiert der Prozessor in vordefinierter Weise auf diese Ausnahme. Diese Ausnahme Behandlung (Exception Handling) wird normalerweise von dem Betriebssystem übernommen. In unserem Fall, in dem kein Betriebssystem vorhanden ist, muß sie daher als Teil der FLT-C-Bibliothek implementiert werden. Vor dem Start des Benutzerprogramms (siehe Abschnitt 5.3.3) wird eine Exception-Tabelle angelegt, auf deren Adresse der Exception-Vektor des Prozessors zeigt. Die Tabelle enthält als Einträge Zeiger auf Funktionen, die beim Auftreten von Exceptions aufgerufen werden sollen. Im Fall einer Exception liest der Prozessor dann den zu der jeweiligen Exception-Nummer gehörenden Eintrag aus der Tabelle und das Programm springt an die angegebene Adresse, um auf den Fehler zu reagieren.

Der Benutzer hat die Möglichkeit, eigene Funktionen in die Exception-Tabelle einzutragen. Wird für eine Exception-Nummer keine benutzerspezifische Funktion eingetragen, so wird eine Standard Handler Funktion aufgerufen. Diese informiert den Unix-Host über die aufgetretene Exception. Der Board-Prozess gibt die Art des Fehlers, den Exception-Frame und den Inhalt der Prozessorregister aus und geht anschließend in eine Endlosschleife. Hierzu wurde die Funktion `exeptionHandler` in die Kommunikationsschnittstelle zu `trun` aufgenommen. Sie dient der Übergabe der Exception-Informationen und beendet `trun` nach deren Ausgabe.

Auf Funktionen der Programm-Umgebung und der Speicherverwaltung, die ebenfalls Bestandteil der Laufzeitumgebung sind, wird in den folgenden beiden Abschnitten gesondert eingegangen.

### 5.3.3 Die Umgebung eines C-Programms

Die Umgebung eines C-Programms beinhaltet Funktionen, die generell vor dem Start des eigentlichen Benutzerprogramms zu Initialisierungszwecken und nach seiner Beendigung ausgeführt werden. Anschließend beginnt die Ausführung eines C-Programms mit dem Aufruf der Funktion

```
int main(int argc, char** argv);
```

Auch bei dem Aufruf von C-Programmen für die Boards ist eine normale Übergabe von Befehlszeilenargumenten an `main` möglich (siehe 5.2.1). `Trun` kopiert hierzu den Inhalt seiner Befehlszeile, der sich auf den Board-Prozess bezieht, in den Arbeitsspeicher des Boards. Anschließend legt es das Array mit Zeigern auf die Argumentzeichenketten neu an und stellt einen Zeiger auf dieses Array (`argv`) und die Anzahl der Argumente (`argc`) für den Board-Prozeß bereit. Erst dann wird dieser gestartet.

Beim Start eines C-Programms durch das Betriebssystem wird vor dem Aufruf der Funktion `main` eine spezielle Startroutine abgearbeitet. Die Funktion `main` wird also nie direkt von dem Betriebssystem aufgerufen, sondern in einer ausführbaren Programmdatei ist immer die Adresse der Startroutine als Startadresse des Programms verzeichnet.

Diese Startroutine ist natürlich stark systemabhängig und daher, wie auch bei der FLT-Laufzeitumgebung, oft in Assembler geschrieben. Zu der FLT-C-Bibliothek wird hierzu das 68k-Assemblerprogramm `crt0.s` hinzugelinkt. Die Programmstartadresse wird dabei dem Linker übergeben, der sie in die ausführbare Programmdatei einträgt. Die Startroutine der Board-Prozesse muß vor dem Aufruf von `main` folgende Initialisierungsaufgaben (vergleiche Abbildung 5.6) ausführen :

- Den Zustand des Prozessors vor dem Programmaufruf speichern, um ihn nach dessen Beendigung wieder in den vorgefundenen Zustand zu versetzen.
- Tabellen für Exception- und Signal-Handling initialisieren.
- Den Stack und den Heap an die Adressen aus der Konfigurationsdatei setzen. Die Heap-Liste für dynamische Speicheranforderungen initialisieren.
- Die Parameter für `main`, die von `trun` bereitgestellt wurden, auf den Stack legen.

Anschließend wird an den Anfang der `main`-Funktion verzweigt und damit das Benutzerprogramm gestartet.

Bei der Beendigung des Programms spielt erneut `crt0.s` eine Rolle. Ein Benutzerprogramm kann auf drei verschiedene Arten normal beendet werden: Mit der Rückkehr von `main`, dem Aufruf der Funktion `exit` oder dem Aufruf der Funktion `_exit`. Die Rückkehr der Funktion `main` durch einen Aufruf von `return` ist

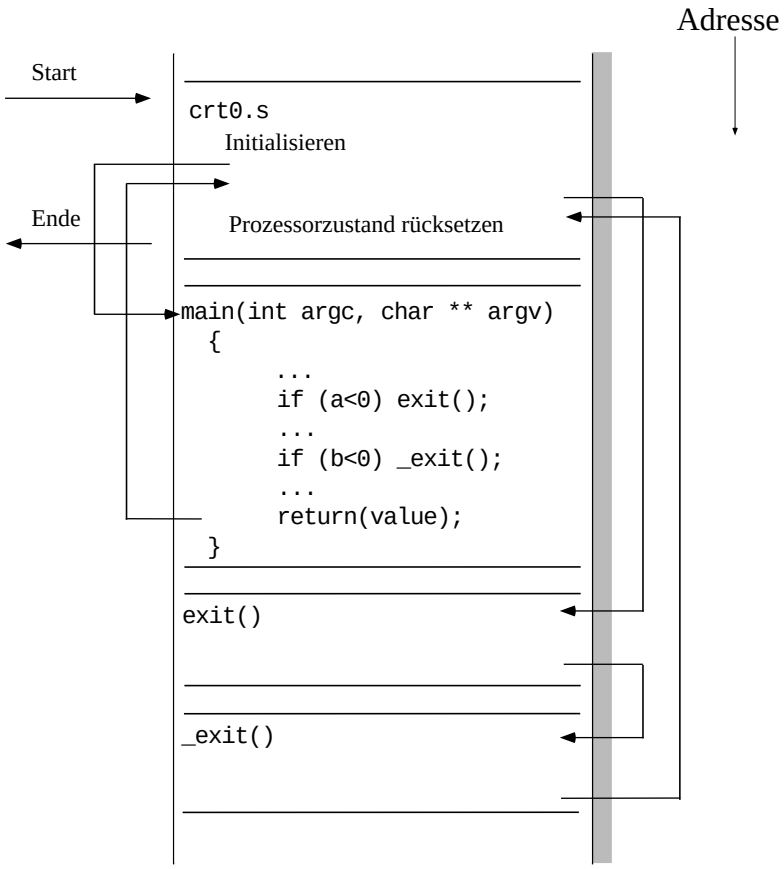


Abbildung 5.6: *Die Umgebung eines Benutzerprogramms im Arbeitsspeicher eines Boards. Die Pfeile geben den Ablauf eines Programms wieder, das mit dem Aufruf von* `return` *normal beendet wird. Bei einem direkten Aufruf von* `exit` *oder* `_exit` *in* `main` *springt das Programm direkt an die entsprechenden Stellen und läuft dann weiter wie dargestellt.*

äquivalent zu dem Aufruf von `exit`. Die Startroutine ist so geschrieben, daß die Funktion `exit` aufgerufen wird, sobald `main` zurückkehrt (siehe Abbildung 5.6). Der Aufruf von `_exit` beendet ein Board-Programm sofort. `Exit` hingegen führt vorher noch einige Aufräumarbeiten durch. Es ruft Funktionen, die mit `atexit` registriert wurden auf, schließt die offenen Dateien auf dem Unix-Host und unterrichtet den `trun`-Prozeß von der gewünschten Beendigung des Programms. Der exit-Status des Board-Prozesses wird dabei an `trun` weitergegeben, das seinerseits dann `exit` aufruft. `Exit` ist daher ebenfalls Bestandteil der Kommunikationsschnittstelle zu `trun` (Abschnitt 5.2.2). Anschließend wird dann ebenfalls die Funktion `_exit` aufgerufen. Zuallerletzt wird der Prozessor wieder in den ursprünglichen Zustand vor dem Aufruf von `main` versetzt.

Der Board-Prozeß kann auch auf drei anormale Arten beendet werden: Mit dem Aufruf von `abort`, durch ein Signal von dem Unix-Host und durch eine Exception. `Abort` ist Bestandteil der Kommunikationsschnittstelle (Abschnitt 5.2.2) zu `trun` und führt zu der sofortigen anormalen Beendigung des Board-Prozesses und des zugehörigen `trun` Prozesses. Das Übermitteln von Signalen kann ebenfalls das Programm beenden, zum Beispiel, bei Verwendung des Standard-Signal-Handlers, durch Drücken der *Control-C* Taste. Die Beendigung des Programms durch eine Exception, die nicht abgefangen wird, wurde bereits in Abschnitt 5.3.2 angesprochen.

### 5.3.4 Verwaltung des Speichers

Insgesamt stehen für die Laufzeitumgebung und das Benutzerprogramm auf den Prozessorboards 4 MB Arbeitsspeicher zur Verfügung. Das Speicherlayout des Arbeitsspeichers zeigt Abbildung 5.7. Es entspricht einer typischen Aufteilung des Speichers in einzelne Segmente [52]. Der untere Teil des Speichers ist für betriebssystemähnliche Funktionen der Laufzeitumgebung reserviert. Er enthält die Systemstruktur und die Ablage für Programmparameter und Exception-Informationen. Daran schließt sich das C-Programm an, das in Textsegment, initialisiertes- und nichtinitialisiertes Datensegment unterteilt ist. Das Textsegment und das initialisierte Datensegment werden direkt von der Binärdatei auf der Festplatte in den Arbeitsspeicher geladen. Das Textsegment enthält dabei die Maschinenbefehle, die von der CPU ausgeführt werden. Das initialisierte Datensegment enthält Variablen, die außerhalb von Funktionen stehen und vom Programm mit einem Anfangswert initialisiert werden. Das nichtinitialisierte Datensegment wird vor der Programmausführung mit Nullen initialisiert. Es enthält außerhalb von Funktionen deklarierte Variablen, denen im Programm kein Anfangswert zugewiesen wird.

Oberhalb des Programms befindet sich der Heap, der nach oben hin wächst. Der Stack befindet sich am oberen Ende des Arbeitsspeichers und wächst nach unten.

Das Speicherlayout ist weitgehend über eine Konfigurationsdatei einstellbar, um die Laufzeitumgebung an verschiedene Hardware oder spezielle Benutzerwünsche anpassen zu können. Zum Beispiel wird bei dem Test-Board der Stack in den Bereich des SRAMs gelegt, um die Ausführungsgeschwindigkeit zu erhöhen.

Der Heap dient der temporären Bereitstellung von Speicher zur Laufzeit des Programms. Seine Verwaltung ist eine der zentralen Anforderungen an die FLT-C-Bibliothek. Im Rahmen der Systemsoftware ist eine Verwaltung dieses Speicherbereichs über eine doppelt verkettete Liste implementiert worden. Sie ist ausführlich in [14] beschrieben. Als Schnittstelle zu dem Benutzer sind für das dynamische Verwenden von Speicherblöcken die C-Standardfunktionen `malloc`, `calloc`, `realloc` und `free` verfügbar. Damit kann auf gewohnte Weise Speicher angefordert und wieder frei gegeben werden.

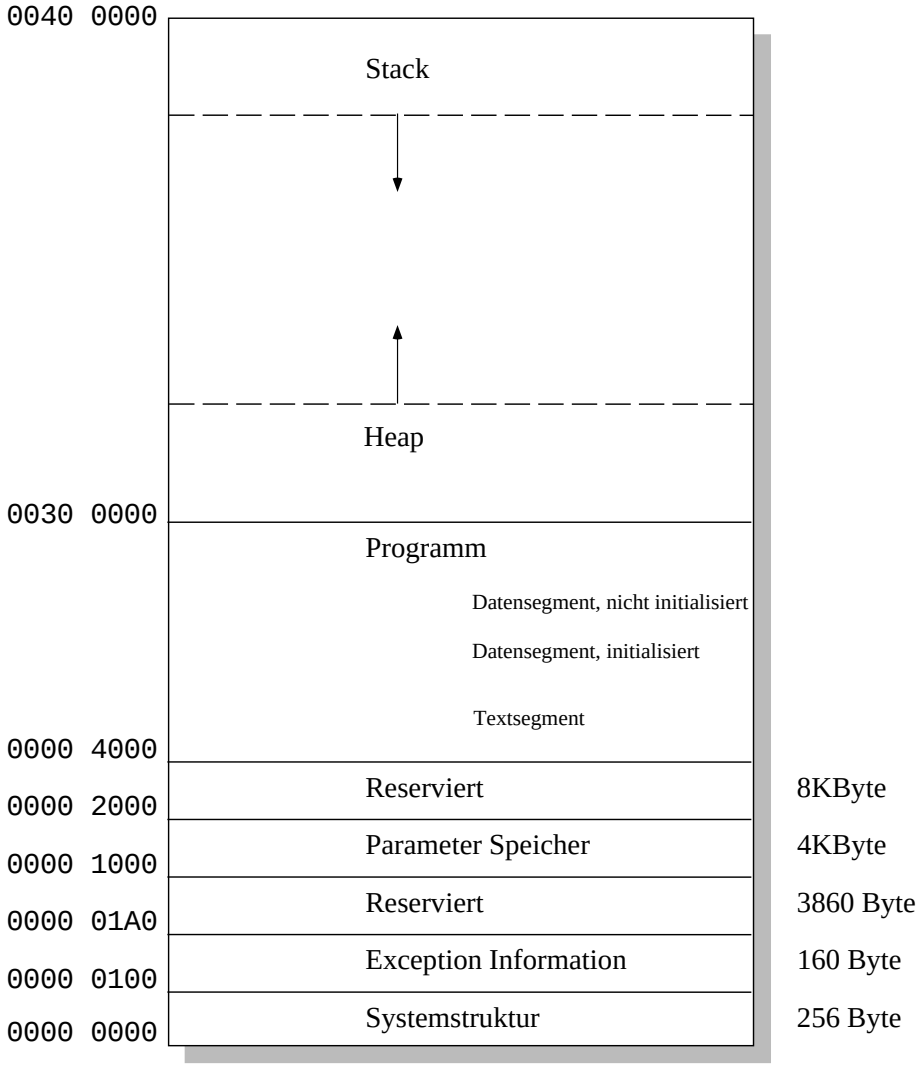


Abbildung 5.7: *Die Einteilung des 4MB großen Arbeitsspeichers von TFU, TPU und TDU.*

## 5.4 Die Netzwerkanbindung der Board-Programme

Während des Betriebs des FLT werden alle Boards von einer Workstation aus gesteuert und überwacht. Um mit diesem zentralen Steuerprogramm Informationen austauschen zu können, müssen die Board-Programme Daten über das Netzwerk empfangen und verschicken können.

Dem Board-Prozeß, und damit `trun`, muß hierfür eine Interprozeßkommunikation mit dem Steuerprozeß ermöglicht werden. Für die Interprozeßkommunikation zwischen Prozessen auf unterschiedlichen Hosts bietet Unix zwei Arten an: Datenströme (Streams) und Sockets. Dabei werden nur die Sockets von allen Unix Systemen un-

terstützt, weshalb sie auch für die FLT-Software mit ihrem heterogenen Umfeld eingesetzt werden. (Zur Diskussion der Netzwerksoftware siehe auch Abschnitt 4.5.) Bei einer Fortführung des bisherigen Konzepts ist es daher erforderlich, das Socket Paket in die FLT-C-Bibliothek zu integrieren und die Systemaufrufe, die dem Paket zugrunde liegen, in die Schnittstelle von `trun` zu integrieren. Damit könnte dann eine eigene Socketkommunikation für die Programme implementiert werden.

Nun ist die Interprozeßkommunikation zwischen Prozessen auf unterschiedlichen Hosts aber eine weit verbreitete Anwendung. Es gibt daher Softwarepakete, die dem Programmierer bereits eine fertige, auf Sockets basierende Kommunikations-Bibliothek mit einfacher Schnittstelle bieten. Beispiele hierfür sind Java und Tcl. Man spart durch ihre Verwendung enorm an Implementierungsaufwand und verwendet außerdem eine getestete Software. Für die Sockets des zentralen Steuerprogramms werden bereits Tcl-Sockets verwendet, da dessen Oberfläche ohnehin in Tcl/Tk programmiert ist (siehe Abschnitt 4.5.2). Zudem läßt sich ein Tcl-Interpreter sehr einfach in ein C-Programm integrieren. Von daher bietet es sich an, auch auf der `trun`-Seite mit Tcl-Sockets zu arbeiten. Um einen Tcl-Interpreter in einem Board-Programm zu verwenden, müßte dann zusätzlich zu dem oben genannten Socket Paket [6] das gesamte Tcl-Paket cross-compiliert werden, und jedes Board-Programm, das Netzwerkzugriff benötigt, muß die Tcl-Bibliothek einbinden. Ein so umfangreiches Paket wie Tcl in die FLT-Laufzeitumgebung auf den Boards zu integrieren, ist als sehr problematisch anzusehen, da es praktisch der Portierung auf eine neue Plattform gleichkommt. Weiterhin ist mit Speicherplatzproblemen auf den Boards zu rechnen. Es ist fraglich, ob sich Tcl mit der FLT-C-Bibliothek überhaupt übersetzen läßt und, falls dies gelingt, dann auch zuverlässig funktioniert.

Um diese Probleme zu vermeiden, wird das Tcl-Paket statt in die Board Software in `trun` eingebunden. Hierzu muß Tcl unter LynxOS übersetzt werden [7]. Anschließend kann dann in `trun` ein Tcl-Interpreter eingebunden werden. `Trun` öffnet, falls erwünscht, beim Start mit Hilfe eines Tcl-Skripts automatisch eine Socket Verbindung zu dem Steuerprogramm. Bei Hinzufügen der Netzwerkkommunikation auf diese Art und Weise muß die Schnittstelle zwischen Board-Prozeß und `trun` nicht angepaßt werden, da Sockets als Datei angesprochen werden und Datei-I/O in der Schnittstelle bereits implementiert ist. Ein Board-Programm kann mit Datei-I/O lesend/schreibend auf einen Socket zugreifen, lediglich der Dateidescriptor des Sockets muß dem Board-Programm bekannt sein.

Insgesamt ergeben sich durch die Verwendung von Tcl-Sockets und deren Einbindung in `trun` statt in die Board-Programme mehrere Vorteile:

- Tcl ermöglicht eine sehr einfache Programmierung der Socket Kommunikation.

[6] Tcl ist ebenfalls in C geschrieben und benötigt daher für seine Übersetzung die entsprechende Socket Software.

[7] Auch dies ist nur mit erheblichen Schwierigkeiten möglich und erfordert Modifikationen in dem Tcl Quellcode.

- Die Tcl-Sockets sind getestet, enthalten weniger Fehler und erfordern damit weniger Debugging.
- Das Socket Paket muß nicht auf den Boards verfügbar sein.
- An der Schnittstelle zwischen Board-Prozeß und `trun` muß nichts hinzugefügt werden.

Insgesamt ergibt sich eine ganz erhebliche Verminderung des Implementierungsaufwandes und eine höhere Betriebssicherheit.
Als Nachteil ergibt sich die Einschränkung, daß mit einem Programm während der Laufzeit kein zusätzlicher Socket geöffnet werden kann. Wenn man dies möchte, dann muß hierfür `trun` modifiziert werden.

Auf der Seite der FLT-C-Bibliothek wird die Socketdatei über einen zusätzlichen Standarddescriptor gehandhabt. Gemäß Konvention wird von Unix-Shells der Dateidescriptor `0` für die Standardeingabe, der Dateidescriptor `1` für die Standardeingabe und der Dateidescriptor `2` für die Standardfehlerausgabe verwendet. Entsprechend sind in der C-Standardbibliothek drei Datenströme mit gepuffertem I/O vordefiniert und für Prozesse automatisch verfügbar. Diese Standard-I/O-Datenströme werden durch drei vordefinierte Dateizeiger `stdin`, `stdout` und `stderr` bezeichnet.

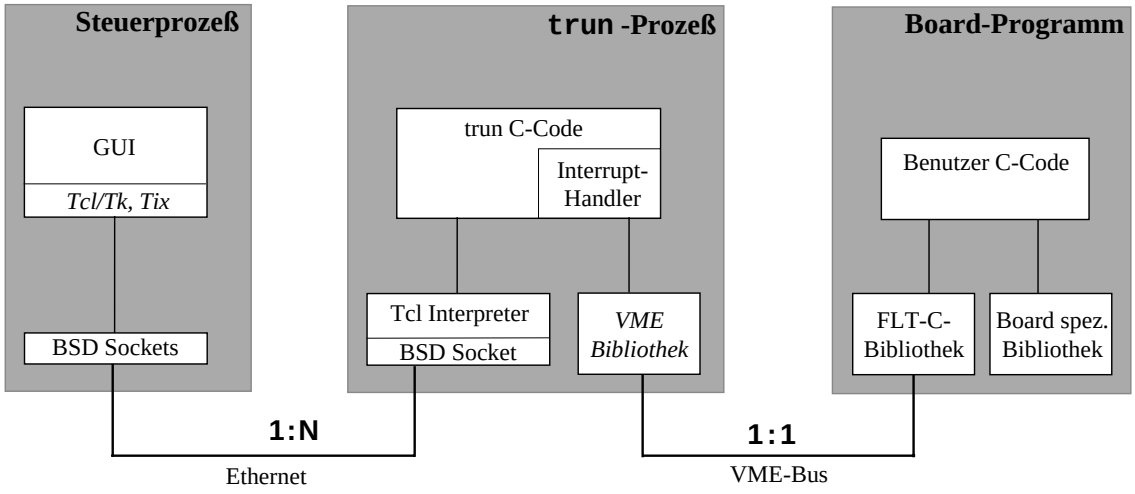


Abbildung 5.8: *Die FLT Steuerprozesse.*

Bei der FLT Version der Bibliothek wird nun für einen vierten Datenstrom ein vierter Dateizeiger `stdsocket` mit dem Dateidescriptor `3` hinzugefügt. Dies paßt sich nahtlos in das bestehende Konzept ein. Damit besitzt jedes Board-Programm automatisch eine vordefinierte Socket Netzwerkverbindung, die von `trun` zur Verfügung gestellt wird. Eine Fehlermeldung kann zum Beispiel auf folgende Weise über den Socket abgeschickt werden:

```
fprintf(stdsocket, ``ERROR: Message FIFO overflow.''),
```

Natürlich darf diese Datei nur angesprochen werden, wenn `trun` auch im Netzwerk Modus gestartet wurde.

Zwischen `trun` und dem Steuerprogramm findet eine Client-Server Kommunikation statt. Das Steuerprogramm als Server wartet auf Verbindungsanfragen der `trun`-Clients. Ein `trun`-Client initiiert jeweils den Aufbau der Verbindung, über die dann bidirektional Daten ausgetauscht werden können.

## 5.5 Zusammenfassung

Thema dieses Kapitels ist die VME-Systemsoftware. Sie stellt ihren Benutzern eine Programmierumgebung zur Verfügung, mit der die Mikroprozessoren auf den Triggerboards in C programmiert werden können. Der Benutzer hat bei der Programmierung und bei der Interaktion mit dem laufenden Programm den Eindruck, als würde der Board-Prozeß auf dem Unix System ablaufen. Alle Aspekte bezüglich Laden und Starten des Programms und der Kommunikation über den VME-Bus sind für den Benutzer, sowohl bei der Programmierung als auch bei der Benutzung seines Programms, völlig transparent.

Erreicht wurde dies durch eine an die spezielle Hardware der Boards angepaßte Version der C-Standardbibliothek, kombiniert mit einem auf dem Unix-Host laufenden Service-Prozeß (ein `trun`-Prozeß pro Board), der das Board-Programm startet und einen Teil der Unix-API auf das Board exportiert. Die VME-Bus Kommunikation ist darin vollständig gekapselt und es kann sowohl der Board-Prozeß das Unix System ansprechen, als auch umgekehrt das Betriebssystem den Board-Prozeß. Um dies zu ermöglichen, wurden zwischen Host und Prozessor-Boards mehrere Kommunikationsprotokolle implementiert.

Die Kommunikation ist vollständig ereignisgesteuert, beide Seiten können eine Kommunikation anstoßen, die grundsätzlich asynchron über Interrupts ausgelöst wird. Dies ist im Vergleich zu Polling sehr effizient, da der VME-Bus nur beansprucht wird, wenn tatsächlich Bedarf besteht. Zusätzlich geht `trun` nach dem Programmstart in den Schlaf-Modus über, und die Interrupt Routine wird im Bedarfsfall dann mit einstellbarer Priorität als Thread abgearbeitet. Ein Board-Prozeß, der keine Ein-/Ausgabe macht, nimmt zu diesem Zeitpunkt weder Rechenzeit des Unix-Systems, noch Resourcen des VME-Bus in Anspruch.

Die Anbindung eines Board-Prozesses an das Netzwerk erfolgt über die Einbindung des Tcl-Paketes in `trun`. Dem Board-Prozeß wird die Netzwerkverbindung über einen Standarddescriptor zur Verfügung gestellt.

# Kapitel 6

# Das Steuerungsprogramm mit Benutzeroberfläche

Der gesamte First Level Trigger soll mit einen Steuerungsprogramm von einer Workstation aus bedient werden können. Um eine einfache Bedienung zu ermöglichen, erhält dieser zentrale Kontrollprozeß eine grafische Oberfläche, über die als Benutzerschnittstelle das gesamte FLT-System überwacht und gesteuert wird. Nach einer Betrachtung der Anforderungen an das Steuerungsprogramm, werden im darauf folgenden Abschnitt die Erfordernisse der grafischen Oberfläche genauer spezifiziert. Besonderes Augenmerk gilt dabei der Auswahl der Entwicklungswerkzeuge mit deren Hilfe der Entwicklungsaufwand für die Oberfläche möglichst gering gehalten werden soll. Anschließend wird die Implementierung der Oberfläche in der Sprache Tcl/Tk und ihre Funktionalität besprochen. Am Schluß des Kapitels folgt die Realisierung der Netzwerkkommunikation des Kontrollprozesses mit den lokalen Prozessen auf den Boards.

## 6.1 Anforderungen an den Kontrollprozeß

Der FLT ist ein komplexes Multiprozessorystem, das aus etwa 80 Hardware-Prozessoren aufgebaut ist, die über ihre Message-Schnittstellen auf komplexe Weise untereinander verbunden sind. Hinzu kommt ein heterogenes Rechnersystem mit etwa der gleichen Anzahl von Rechnern, die der Kontrolle des Multiprozessors dienen und über VME-Bus und Ethernet miteinander kommunizieren.

Mit Hilfe eines zentralen Kontrollprozesses soll der gesamte First Level Trigger überwacht und gesteuert werden. Der Kontrollprozeß befindet sich auf einer Workstation und muß von dort aus über Netzwerk mit den VME-Host Computern kommunizieren. Diese wiederum stellen via VME-Bus den Kontakt zu den lokalen Prozessen auf den Board-Rechnern her. Von der Workstation aus muß der Kontrollprozeß das System starten können, alle Einstellungen vornehmen, beziehungsweise von den lokalen

Rechner vornehmen lassen, die Betriebsparameter überwachen und gegebenenfalls auf Fehlermeldungen reagieren. Das Triggersystem ist für diese Zwecke vollständig durch Software konfigurierbar und kontrollierbar.

Ein Problem stellen dabei die zahlreichen Einstellmöglichkeiten und Betriebsparameter dar, über die nur schwer der Überblick zu wahren ist. Jeder Hardware-Prozessor für sich hat bereits eine große Zahl von Parametern, die für den Betrieb gesetzt werden müssen, wie zum Beispiel die Informationen über die lokale Detektorgeometrie, die für die Berechnung der Lookup-Tabellen benötigt werden. Hinzu kommen etliche Parameter, wie Message-Raten oder Betriebsspannungen, die für die Überwachung des aktuellen Betriebszustandes herangezogen werden können.

Die Aufgaben des Kontrollprozesses lassen sich grob in Managment-, Kontroll- und Anzeigeaufgaben einteilen und sind in dieser Reihenfolge im folgenden aufgelistet:

- Run Management
  1. Verwalten der Konfigurationsdateien, Pfade und Run-Nummern.
  2. Archivierung der aktuellen Run-Parameter, log Dateien ...
- Automatisches Starten eines Trigger Runs
  1. Einlesen der aktuellen Systemkonfiguration.
  2. Starten der lokalen Prozesse auf den Boards.
  3. Startup Tests des Triggersystems.
- Überprüfen des Systemaufbaus und Funktionstests der Boards und des Gesamtsystems.
- Steuerung des Triggers, manuell und als Reaktion auf bestimmte Betriebszustände, zum Beispiel auf Fehlermeldungen.
- Anzeige des Systemaufbaus.
  1. Einteilung der TFUs auf die Detektorlagen.
  2. Verbindungen der Prozessoren untereinander.
  3. Zuordung zwischen den Bereichen einer Detektorlage und TFUs.
  4. Verteilung der Boards auf die VME-Crates.
- Kontinuierliche Anzeige der aktuellen Betriebsparameter.
- Ausgabe von Fehlermeldungen bei Störungen.
- Darstellung wichtiger Datenstrukturen, wie den Wire-Ram Inhalt.

Um diese Aufgaben wahrnemen zu können, muß der Kontrollprozeß in der Lage sein, über das Netzwerk mit den Prozessen auf den einzelnen Boards Informationen auszutauschen. Jedem Board ist hierfür ein `trun`-Kommunikationsprozeß auf seinem jeweiligen VME-Host Computer zugeordnet. Es gibt also etwa 80 `trun`-Prozesse, die zu Beginn von der zentralen Workstation aus gestartet werden und mit denen anschließend Daten ausgetauscht werden müssen.

Der Kontrollprozeß hat weiterhin die Aufgabe für den Benutzer die Komplexität soweit zu reduzieren und anschaulich darzustellen, daß das System möglichst intuitiv bedienbar wird und sein zuverlässiger Betrieb sichergestellt ist. Dies ist wichtig, um den Aufwand zur Einarbeitungung zu reduzieren und eine Bedienung ohne das Beisein von Experten zu ermöglichen. Aus diesen Gründen soll die Bedienung des Kontrollprozesses über eine grafische Benutzeroberfläche erfolgen.

Als Basisanforderungen für den Kontrollprozeß ergeben sich also die Fähigkeit zur Netzwerkkommunikation und die Verwendung einer grafischen Oberfläche. Damit können dann die oben genannten Aufgaben realisiert werden. Im folgenden wird nun zuerst die Benutzeroberfläche und anschließend die Kommunikation über Netzwerk betrachtet.

## 6.2 Die grafische Benutzeroberfläche

Die Entwicklung einer grafischen Benutzeroberfläche vereinfacht die Bedienung eines Programms, erhöht aber andererseits stark den Implementierungsaufwand. Der Aufwand für eine grafische Oberfläche ist in der Regel vergleichbar mit dem Aufwand zur Programmierung der eigentlichen Programmfunktionalität. Aus diesem Grund gibt es zahlreiche Entwicklungsumgebungen, die das Entwerfen von grafischen Oberflächen vereinfachen sollen und den Entwickler von immer wiederkehrenden Arbeiten entlasten. Daher spielt es eine wichtige Rolle, für die Oberflächenerstellung des FLT-Kontrollprozesses die richtige, also problemangepaßte, Umgebung auszuwählen.

### 6.2.1 Anforderungen an die Oberfläche und Entwicklungswerkzeuge

Neben den speziellen Anforderungen an den Kontrollprozeß gelten für die Oberflächenentwicklung folgende Anforderungen und Randbedingungen:

- Bereitstellung der wichtigsten Oberflächen-Elemente wie Menüs, Knöpfe, Regler usw.
- Möglichst einfache und effiziente Entwicklungsumgebung.
- Klar definierte Schnittstelle zwischen dem eigentlichen Programm und der Oberfläche.

- Erweiterbarkeit der Oberfläche um eigene Elemente.
- Verfügbarkeit der Entwicklungswerkzeuge im Rahmen des HERA-B Projekts.
- Die FLT-Oberfläche besitzt keine zeitkritischen Elemente.

Als Grundlage jeder Oberfläche dient eine Grafikschnittstelle zu dem jeweiligen Betriebssystem. Von dieser Schnittstelle werden in der Regel nur elementare Funktionen, wie das Öffnen von Fenstern, Ereignisverarbeitung von Maus oder Tastatur und einfache Maloperationen, zur Verfügung gestellt. Alle komplexeren Elemente und deren Verhalten müssen aus diesen Basisfunktionen aufgebaut werden.

Um nicht bei jeder Anwendung wieder das Rad neu erfinden zu müssen, gibt es zahlreiche grafische Oberflächen-Builder und -Bibliotheken mit einer Anzahl von vorgefertigten grafischen Objekten, sogenannte „Widget-Sets“.

Im folgenden wird daher eine Anzahl von Oberflächen-Buildern und Bibliotheken besprochen, die dem Stand der Technik entsprechen. Es wird geprüft inwieweit sie den Randbedingungen des Projekts genügen und dann eine geeignete Entwicklungsumgebung ausgewählt, mit deren Widgets die Oberfläche für den FLT aufgebaut werden kann.

### 6.2.2 Auswahl der Oberflächenwerkzeuge

Unix-Betriebssysteme verwenden das X-Window-System (X11) als Standard-Grafikschnittstelle. Aufsetzend auf der Basisfunktionalität von X11, sind unter Unix das frei verfügbare Xaw [1] und das kommerzielle Motif [2] die am weitesten verbreiteten Bibliotheken.

Trotz der, gegenüber der reinen X11 Programmierung, wesentlich bequemeren Programmierung, ist auch hier die Implementierung einer Oberfläche umständlich und zeitaufwendig. Man bewegt sich immer noch auf einem sehr niedrigen Abstraktionsniveau, es wird viel Know-How benötigt und bei jeder Änderung muß das gesamte Programm neu compiliert werden. Der Vorteil dieser relativ fundamentalen Implementierung liegt in der Schnelligkeit der Programme. Sie ist also bei zeitkritischen Anwendungen, wie zum Beispiel der Anzeige von Bildsequenzen, vorzuziehen, was aber bei dem FLT nicht der Fall ist. Ein weiterer Vorteil ist die hohe Portabilität der Programme, auch dies ist aber bei dem FLT kein entscheidendes Kriterium. Deswegen ist es besser, Bibliotheken zu verwenden, die ein noch höheres Abstraktionsniveau aufweisen.

[1] Athena Widget Set
[2] Motif Widget Set, nach dem Motif Standard der Open Software Foundation

### WxWindows

WxWindows ist ein frei verfügbares Paket zur Entwicklung grafischer Oberflächen. Der Kern von wxWindows ist eine umfassende C++ Klassenbibliothek, hinzu kommt eine Werkzeugsammlung zur Entwicklung grafischer Benutzerschnittstellen. Der Schwerpunkt des Pakets liegt in der plattformübergreifenden Oberflächenentwicklung für Unix, Windows und mit Einschränkungen MacOS. Die Oberfläche besitzt dabei das Look und Feel des jeweiligen Zielsystems.

Die Systemunabhängigkeit und Portierbarkeit ist zwar kein Kriterium für die FLT Software, aufgrund der objektorientierten Programmarchitektur und der Vielzahl zusätzlicher Utilities ist wxWindows aber für eine Verwendung bei dem Kontrollprozeß gut geeignet. Das verwendete Linux-Betriebssystem mit g++ Compiler gehört auch zu den getesteten Plattformen von wxWindows.

Neben den Bibliotheken gibt es Schnittstellen-Builder mit denen interaktiv am Bildschirm eine grafische Oberfläche erstellt werden kann. Diese Schnittstellen-Builder bieten den höchsten Abstraktiongrad, da man nicht mehr programmiert, sondern interaktiv ein Modell der Oberfläche erstellt. Ausgehend von diesem Modell wird dann der Programmcode für die Oberfläche automatisch generiert.

### XDesigner

X-Designer ist ein interaktives Werkzeug der Firma Imperial Software Technology zum Erstellen von Oberflächen. Es verwendet die Widgets der Motif Bibliothek als Bausteine. Die Oberfläche kann direkt am Bildschirm zusammengestellt werden. X-Designer stellt dabei auf dem Bildschirm die Hirarchie der Widgets und gleichzeitig die Oberfläche in ihrem Aussehen und Verhalten dar. Wenn das Design abgeschlossen ist kann direkt der benötigte C oder C++ Code für die konstruierte Benutzerschnittstelle generiert werden. In Testberichten wird XDesigner insgesamt als ausgereift, einfach und effizient nutzbar beschrieben.

X-Designer ist allerdings ein kommerzielles Produkt. Der Einsatz für die FLT-Software käme deswegen nur bei gravierenden Vorteilen gegenüber freien Paketen in Frage, da X-Designer dann auch von anderen Gruppen der Kollaboration gekauft werden müßte, was erfahrungsgemäß sehr schwierig durchzusetzten ist.

### XF und SpecTcl

Es gibt momentan vier frei verfügbare interaktive Oberflächen-Builder, die aus der entworfenen Oberfläche Tcl/Tk Code erzeugen. (Zu der Sprache Tcl/Tk siehe Abschnitt 4.4.) Die Programme Visual Tcl Project und GUIBuilder wurden nicht weiter betrachtet, weil das erstere sich noch im Anfangsstadium befindet und das Zweite vermutlich nicht weiterentwickelt wird, da es bereits die neuesten Tk Versionen nicht

mehr unterstützt. Damit verbleiben die Programme XF und SpecTcl, die beide auch selbst in Tcl/Tk geschrieben sind.

XF ist eine der ältesten und erfolgreichsten Applikationen in Tcl/Tk überhaupt. Es ist sehr umfangreich und auch für große Projekte geeignet. Bereits vorhandene Applikationen lassen sich mit XF weiterbearbeiten. Die Manipulation grafischer Objekte und die Verknüpfung von Aktionen mit Ereignissen wird direkt am Bildschirm vorgenommen und das Resultat dann als Tcl-Skript abgespeichert.

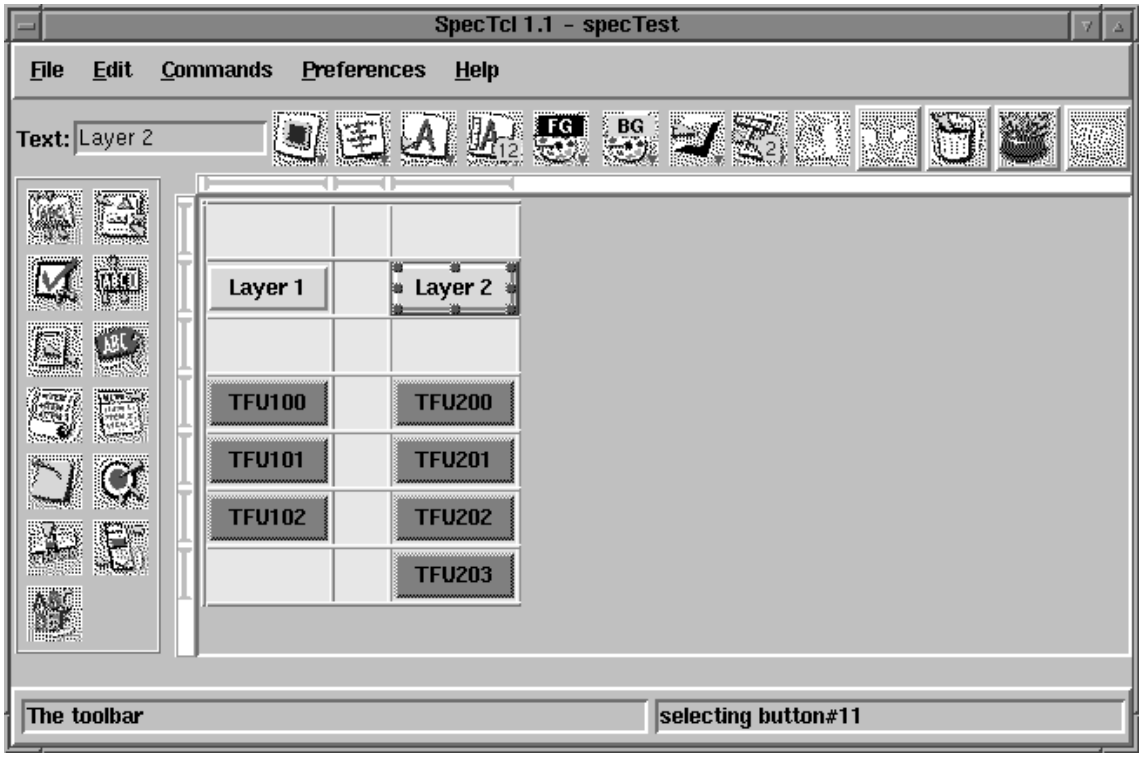


Abbildung 6.1: *Der Oberflächen-Builder SpecTcl.*

SpecTcl ist ein Tcl/Tk Oberflächen-Builder von Sun. Das 1.0 Release ließ leider sehr lange auf sich warten und vorherige Versionen waren nicht benutzbar. SpecTcl besitzt eine etwas spartanische Oberfläche und bereits vorhandene Tcl-Anwendungen lassen sich nicht in SpecTcl laden und weiterbearbeiten. Abbildung 6.1 zeigt SpecTcl als Beispiel für einen Oberflächen-Builder. Insgesamt macht XF einen ausgereifteren und besser verwendbaren Eindruck. Auch schien die Entwicklung von SpecTcl bei Sun nur halbherzig vorangetrieben zu werden [3]. Daher ist die Verwendung von XF auf jeden Fall vorzuziehen.

[3] Tatsächlich wurde die Weiterentwicklung von Sun mittlerweile eingestellt.

### Die Auswahl

Zusammenfassend ergibt sich also, daß die Verwendung von Basisbibliotheken wie Motif wegen des zu niedrigen Abstraktiongrades nicht in Frage kommt, geeignet hingegen sind Bibliotheken mit hohem Abstraktiongrades, wie das wxWindows Paket.

Ebenfalls nicht in Frage kommt ein kommerzieller Oberflächen-Builder wie XDesigner, da er im Rahmen des zentralen Kontrollprozesses keine entscheidenden Vorteile gegenüber freien Paketen bietet und seine Verwendung deshalb in der Kollaboration nicht begründet werden kann.

Es verbleiben noch XF und die direkte Programmierung in Tcl/Tk. Die grundlegende Frage, die sich dabei stellt ist, ob es für den Kontrollprozeß überhaupt sinnvoll ist einen Tcl/Tk-Generator einzusetzen, oder ob man nicht besser direkt in dieser Sprache entwickelt. Auch mit den Oberflächen-Buildern kann keine Oberfläche ohne grundlegende Kenntnisse von Tk erstellt werden, insbesondere wenn, wie in diesem Fall, zusätzliche Funktionalität zu der reinen Oberfläche eingebaut werden muß. Das Erlernen der Sprache ist also kein Kriterium für oder gegen eine der beiden Möglichkeiten. Die Verbindung des generierten Codes mit manuell erstelltem Code erfordert aber einen zusätzlichen Aufwand. Hier ist insbesondere der generierte Code von XF umfangreich, schlecht lesbar und damit schwierig nachzuvollziehen.

Bei komplexen Programmierschnittstellen, wie zum Beispiel von Motif, ist der Einsatz eines Oberflächen-Builders, der das mausgestützte Entwerfen erlaubt und den Quellcode generiert, sinnvoll. Er erspart die aufwendige Programmierung in einer Compilersprache und lange Turnaround-Zeiten. Die Programmierschnittstelle von Tcl/Tk bietet aber bereits ein so hohes Abstraktionsniveau, daß der Einsatz von Tcl/Tk-Generatoren keinen sehr großen Gewinn darstellt und nur bei wiederholtem Erstellen von Oberflächen und rapid prototyping sinnvoll ist. In dem Fall des Kontrollprozesses erfordert das notwendige Einarbeiten in den Oberflächen-Builder und das teilweise Nachvollziehen des jeweils generierten Codes, um zusätzliche, nicht oberflächenbezogene Funktionalität einzubauen, einen ähnlichen oder sogar höheren Aufwand, wie das direkte Entwickeln in Tcl/Tk.

Die Auswahl eines geeigneten Werkzeugs zur Entwicklung der Oberfläche des Kontrollprozesses ist also letztlich zwischen wxWindows und Tcl/Tk zu treffen. Der Kern dieser Fragestellung besteht darin, ob man für die Programmierung eine System-Programmiersprache oder besser eine Skriptsprache einsetzen will.

System-Programmiersprachen sind dazu entworfen um komplexe Datenstrukturen und Algorithmen von Grund auf neu zu implementieren. Sie sind für die gleichen Aufgabenstellungen wie Assemblersprachen konzipiert mit dem Unterschied, daß sie einen höheren Abstraktionsgrad bieten und streng typisiert sind.

Skriptsprachen unterscheiden sich davon grundlegend. Sie sind nicht dafür entworfen Applikationen von Grund auf zu entwickeln, sondern sie gehen von der Existenz einer Sammlung nützlicher Komponenten aus, die in der Regel in einer anderen

Sprache geschrieben sind. Komponenten können zum Beispiel Filterprogramme, die in einer Unix-Shell zu Pipelines verbunden werden, oder Elemente einer grafischen Oberfläche sein[37]. Skriptsprachen dienen primär dazu diese Komponenten zu verbinden. Sie sind in der Regel interpretierte Sprachen und tendieren dazu nur schwach oder gar nicht typisiert zu sein. Die fehlende Typisierung erleichtert um das Zusammenfügen und die Wiederverwendbarkeit von Komponenten.

Insgesamt ist bei Verwendung der Skriptsprache Tcl aufgrund ihres höheren Abstraktionsgrades mit fünf- bis zehnfach kürzeren Entwicklungszeiten im Vergleich zu einer Implementierung in C/C++ zu rechnen [37]. Dem steht eine etwa 10-fach langsamere Ausführungsgeschwindigkeit gegenüber (der Skript-Teile, nicht der Komponenten).

Der Kontrollprozeß implementiert keine komplexen Algorithmen oder Datenstrukturen und er ist nicht zeitkritisch. Er besitzt andererseits eine grafische Oberfläche, enthält häufig Manipulationen von Zeichenketten und benötigt zusätzliche Komponenten für Netzwerkkommunikation und das Editieren von Textdateien. Zudem muß er leicht erweiterbar sein. All dies spricht für die Verwendung einer Skriptsprache. Die Implementierung des Kontrollprozesses wird deswegen in Tcl/Tk vorgenommen.

Die Tcl/Tk Lösung erfüllt die Anforderungen, die an den zentralen Kontrollprozeß gestellt wurden. Im einzelnen sind folgende Möglichkeiten gegeben:

- Tcl/Tk stellt alle benötigten Widgets zur Verfügung.
- Die interpretierte Skriptsprache erlaubt eine interaktive, effiziente Implementierung der Oberfläche.
- Die C-Schnittstelle von Tcl erlaubt die Erweiterung der Sprache um eigene Befehle (Komponenten). Tk ermöglicht nach demselben Prinzip die Erweiterung um zusätzliche Widgets.
- Dies machen sich eine Vielzahl von Erweiterungspaketen zunutze, die meistens Erfordernisse für einen speziellen Anwendungdbereich abdecken.
- Die Software und auch die Erweiterungspakete sind frei verfügbar.

Hinzu kommt, daß Tcl auch für Anforderungen, die nicht direkt mit der Oberfläche zusammenhängen bereits fertige Lösungen enthält:

- Das Einlesen, Interpretieren und Editieren von Konfigurationsdateien ist in Tcl sehr elegant und einfach durchzuführen.
- Tcl enthält eine eingebaute Netzwerkkommunikation, die für das zentrale Steuerungsprogramm genutzt werden kann.

Insbesondere die letzten beiden Punkte, daß Tcl/Tk über die Oberfläche hinausgehend bereits fertige Komponenten enthält, die für den Kontrollprozeß benötigt werden, untermauert die Entscheidung Tcl/Tk einzusetzen.

Neueste Entwicklungen wie zum Beispiel Java und Qt konnten bei der Auswahl für den FLT nicht mehr in Betracht gezogen werden, weil sie zu Beginn der Entwicklung noch nicht verfügbar waren. Die C++ Klassenbibliothek Qt ist etwa als äquivalent zu wxWindows anzusehen. Mit der System-Programmiersprache Java wären die Anforderungen an einen Steuerprozeß ebenfalls zu erfüllen, sie enthält Bibliotheken für grafische Benutzeroberflächen und für Netzwerkkommunikation. Es sind aber für den Kontrollprozeß keine Vorteile gegenüber der Verwendung von Tcl/Tk zu erkennen.

## 6.3 Die Implementierung des Kontrollprozesses

Der Kontrollprozeß mit grafischer Benutzeroberfläche zur zentralen Steuerung und Überwachung des First Level Triggers ist vollständig in Tcl/Tk unter Zuhilfenahme des Zusatzpackets Tix implementiert. Das Programm mit dem Namen `tricon` [4] setzt sich aus mehreren funktionellen Einheiten zusammen (Abbildung 6.2).

An oberster Stelle des Programms liegt das Systemdisplay, von dem aus der Benutzer einen grafischen Überblick über die aktuelle Konfiguration und den Betriebszustand des Gesamtsystems bekommt. Darunter arbeitet der Konfigurations- und Run-Manager. Er arbeitet in der Regel automatisch, nur bei Änderungen wird direkt auf ihn zugegriffen.

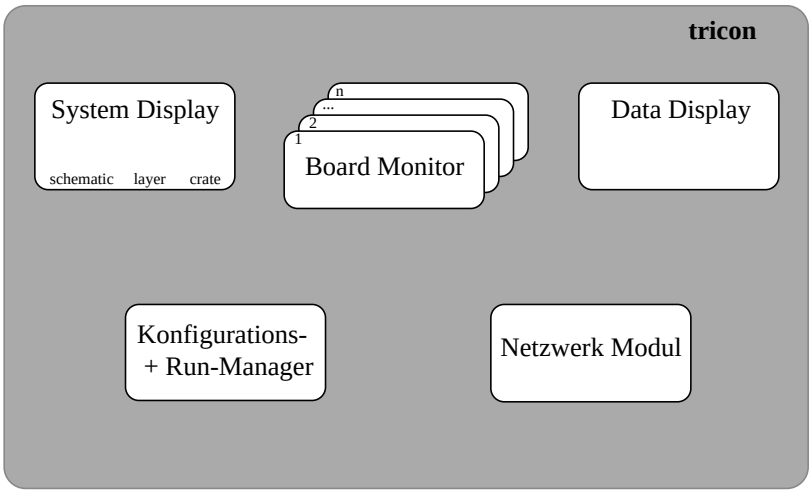


Abbildung 6.2: *Schematischer Aufbau des zentralen Kontrollprozesses.*

[4] TRIgger CONtrol.

Weiterhin gibt es den Board-Monitor, der einmal für jedes Prozessorboard aufgerufen werden kann. Er zeigt den Betriebszustand des jeweiligen Boards an und erlaubt das Absetzen prozessorbezogener Kommandos.

Das Daten-Display dient der Darstellung wichtiger Datenstrukturen wie der Message oder des Wire-Rams.

Die Kommunikation des Kontrollprozesses mit den `trun`-Prozessen wird von dem Netzwerk-Modul abgewickelt. Der Kontrollprozeß fungiert dabei als Server, der die Verbindungsanfragen der `trun`-clients erwartet.

### 6.3.1 Das Systemdisplay

Die Oberfläche von `tricon` stellt drei verschieden Sichten auf das Gesamtsystem zur Verfügung. Zwischen den drei Ansichten kann über ein Notebook-Widget umgeschaltet werden.

Durch anklicken des Notebook-Eintrags *Schematic View* gelangt man in die schematische Ansicht (Abbildung 6.3). Hier ist die Verteilung der TFUs auf die verschiedenen Lagen und die Verbindungen der Prozessoren untereinander gezeigt. Die Prozessoren sind nach Lagen geordnet als Buttons auf dem Bildschirm symbolisch dargestellt. Wenn man die Maus über ein Prozessorsymbol bewegt, werden die Messageverbindungen des jeweiligen Prozessors (Sender) zu den Prozessoren der nachfolgenden Lage (Empfänger) angezeigt. Die Verbindungen sind numeriert, so daß sich genau die Verkabelung des Message-Systems nachvollziehen läßt. Die Prozessorsymbole sind Menubuttons, die den Zustand ihres jeweiligen Prozessors farblich kodieren und durch Anwählen der Menüpunkte können prozessorspezifische Funktionen ausgelöst werden.

Mit einem Mausklick auf *Layer View* gelangt man in die Lagen-Ansicht, die die TFU-bestückten Detektorsuperlagen sortiert als Ebenen darstellt (Abbildung 6.4). Sie dient dazu, einen Überblick über die Zuordnung von TFUs zu den Detektorregionen zu gewinnen. In jeder Superlage ist ihre Aufteilung in TFU-Bereiche eingezeichnet. Die einzelne Ebene läßt sich mit der Maus auswählen und bewegen. Wählt man einen TFU-Bereich aus, so wird die zugeordnete TFU angezeigt und durch Klicken gelangt man in den Monitor für dieses Prozessorboard.

Unter dem Punkt *Crate-View* des Notebooks ist geplant die Verteilung der Prozessorboards auf die Steckplätze der neun VME-Crates wiederzugeben. Dieser Teil ist Gegenstand der Diskussion und daher momentan noch nicht implementiert.

### 6.3.2 Konfigurations- und Run-Manager

Der Konfigurations-Manager liest die Konfigurationsdateien ein, in denen unter anderem die aktuelle Konfiguration des Prozessorsystems abgelegt ist. Die Konfigurationsdateien sind in Tcl geschrieben, trotzdem aber auch ohne Tcl Kenntnisse

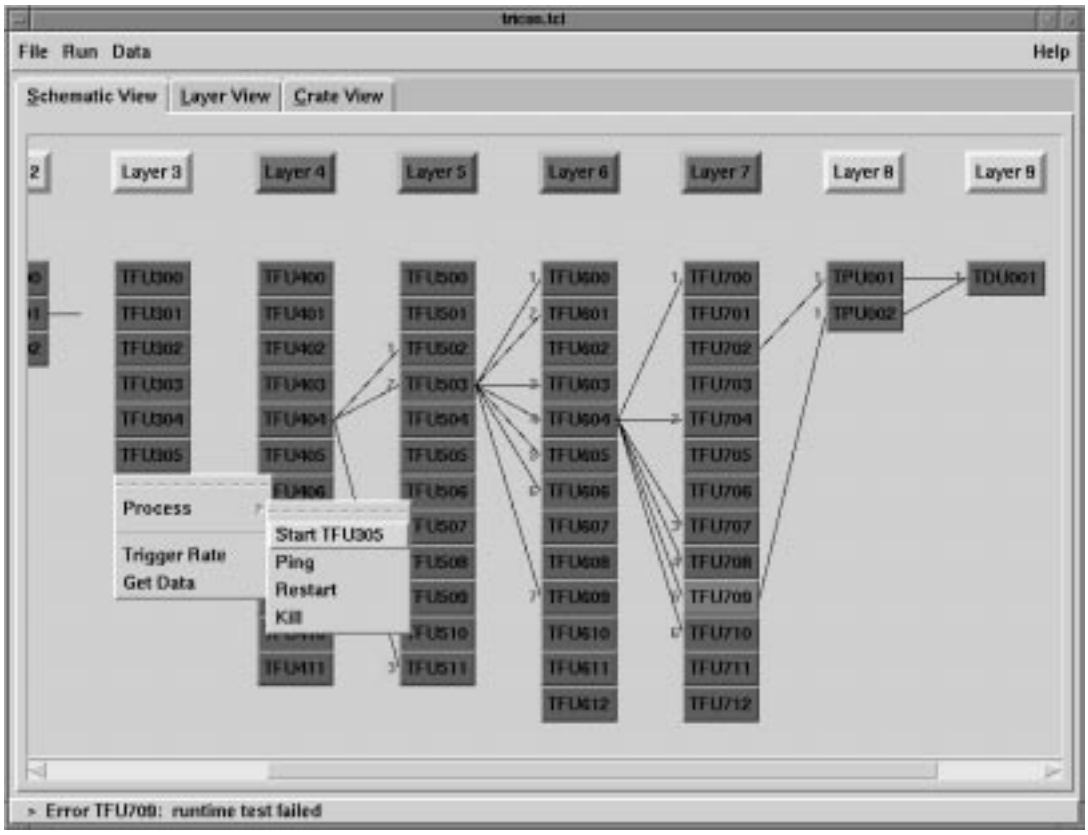


Abbildung 6.3: *Schematische Ansicht der Prozessoren und einer Auswahl ihrer Message-Verbindungen (vergleiche auch Abbildung 3.1). Bei TFU305 ist das Menü heruntergeklappt und TFU709 hat soeben eine Fehlermeldung erhalten, die auch in der unteren Statuszeile angezeigt wird.*

leicht lesbar, da im wesentlichen Variablen gesetzt werden. Die Dateien werden zur Laufzeit eingelesen und direkt von dem Tcl-Interpreter interpretiert, sie sind also im Prinzip Teil des Programms. Über ein Editorfenster hat man Zugriff auf alle Konfigurationsdateien, um gegebenenfalls Änderungen vorzunehmen.

Entsprechend den Angaben in der Konfiguration werden dann die `trun`-Prozesse auf den VME-Host Rechnern gestartet. Die zugeordneten Board-Programme melden ihren erfolgreichen Start an den Kontrollprozeß zurück. Wenn das gesamte Triggersystem betriebsbereit ist, kann die Messung beginnen oder zum Beispiel auch Tests des Gesamtsystems durchgeführt werden.

Das Run-Management verwaltet alle Informationen, die sich auf einen Run beziehen und während des Betriebs anfallen. Unter einem Run ist die Periode vom Neustart des gesamten First Level Triggers bis zum Auslösen eines Stops durch den zentralen Kontrollprozeß zu verstehen.

Das Run-Management vergibt eine Run-Nummer die automatisch hochzählt. Es archiviert unter dieser Nummmer die gesamte Konfiguration des Runs, Log-Dateien

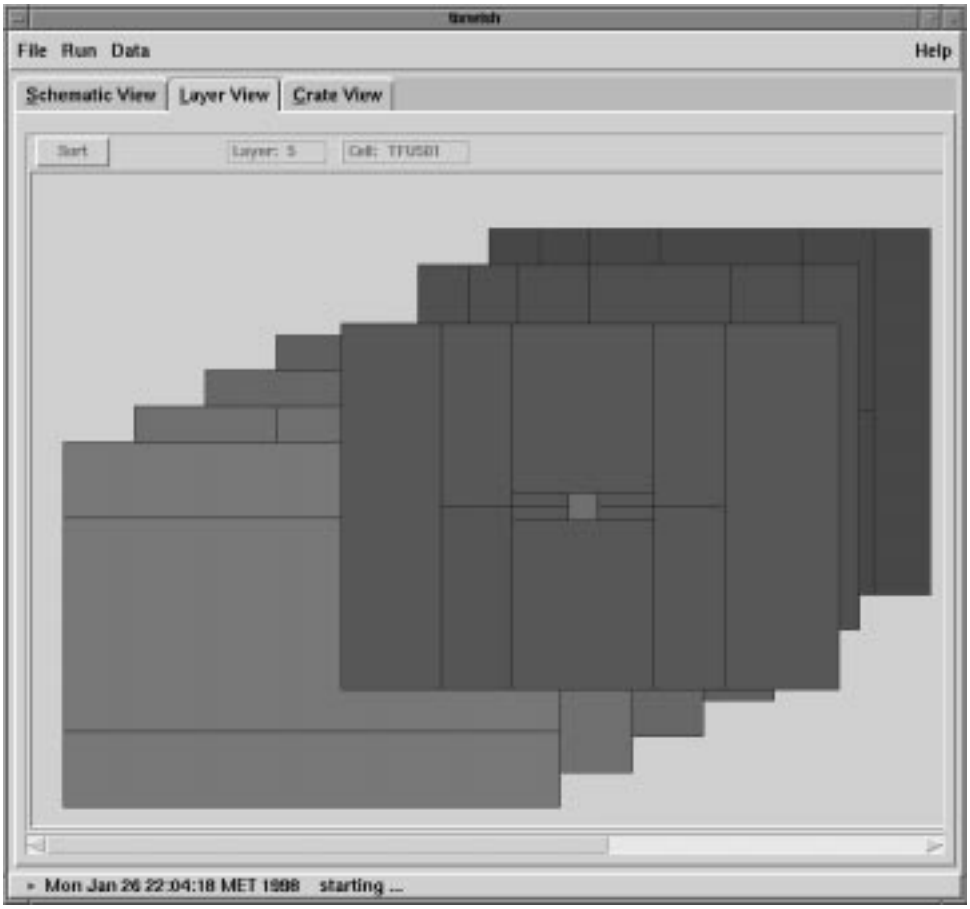


Abbildung 6.4: *Verteilung der Prozessoren auf die Detektorlagen. Die fünfte Lage ist durch anklicken in den Vordergrund gebracht.*

und Error-Dateien. Damit läßt sich später jederzeit rekonstruieren unter was für Bedingungen der Trigger betrieben wurde und was sich während des Betriebs ereignet hat.

### 6.3.3 Der Board-Monitor

Für jeden Prozessor kann aus den drei Ansichten des System Displays durch Klicken auf eines der Prozessor-Symbole ein Board-Monitor aufgerufen werden. Jeder Board-Monitor (Abbildung 6.5) öffnet ein neues Fenster (toplevel Window), in dem die Betriebsparameter des ausgewählten Prozessorboards angezeigt werden.

Diese sind in den beiden Hauptbereichen *Message Rates* und *Operating Data* angeordnet. In dem ersten Bereich werden die Eingangs- und Ausgangs-Message-Raten des Prozessors angezeigt. Der Bereich *Operating Data* beinhaltet Temperaturwerte an sechs Meßpunkten des Boards, acht Betriebsspannungen und die Überwachung der optischen Eingangsleistung der vierundzwanzig Datenübertragungskanäle. Die

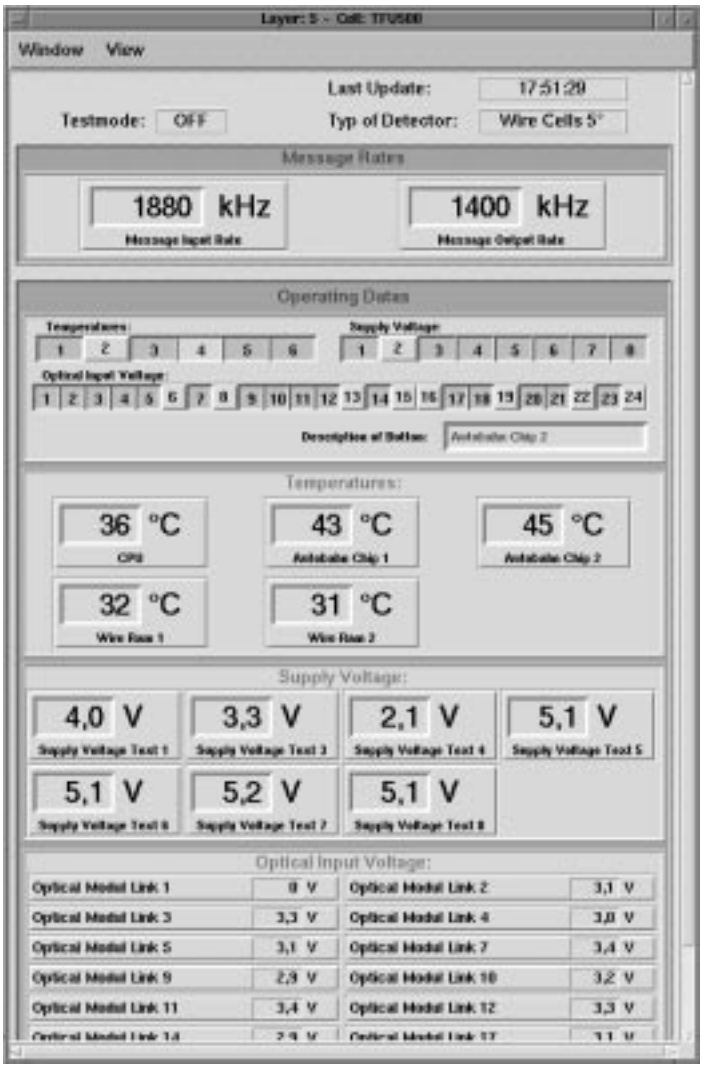


Abbildung 6.5: *Anzeige der Betriebsparameter einer TFU mit dem Board-Monitor.*

Anzeige aller Meßwerte kann selektiv über die darüberliegende Leiste mit Buttons ein oder ausgeschaltet werden.

### 6.3.4 Daten Displays

Mit dem *Data Display* können verschiedene Datenstrukturen, die von den Boards verwendet werden, auf dem Bildschirm dargestellt werden. Damit können auf elementarem Niveau, zum Beispiel um ein Board zu Überprüfen, Daten angezeigt werden.

Der Inhalt des Wire-Rams wird grafisch anschaulich wiedergegeben, da eine direkte Interpretation der Zahlenwerte in den Hit-Daten durch den Benutzer nur sehr

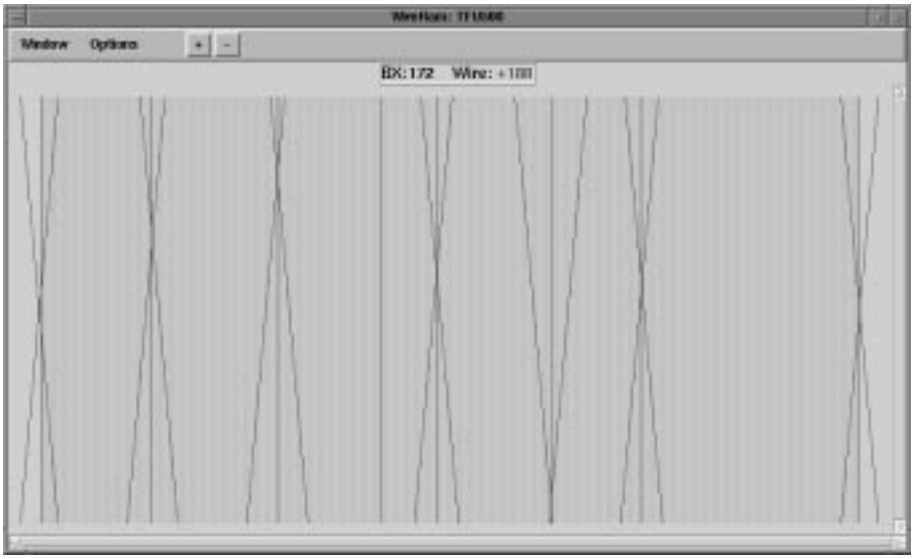


Abbildung 6.6: *Anschauliche Darstellung des Inhalts eines Wire-Rams bei einer TFU, die einer Detektor-Superlage mit 5 Grad gekippten Drahtlagen zugeordnet ist.*


Abbildung 6.7: *Darstellung des Inhalts eines Wire-Rams bei einer TFU, die einem Pad-Detektor zugeordnet ist.*

schwierig oder gar nicht möglich ist. Dabei werden die Daten je nach Detektortyp verschieden dargestellt.

Bei den Drahtkammerdetektoren sind die drei Lagen einer Superlage um einen bestimmten Winkel zueinander gekippt (Abbildung 6.6). Entsprechend ist der Inhalt der drei Wire-Ram Datenstrukturen als zueinander gekippte Drahtfolgen dargestellt. Von den beiden gekippten Lagen sind nur die gesetzten Drähte gezeichnet, während bei der vertikalen Lage zusätzlich die nicht gesetzten Drähte grau eingezeichnet sind.

Ein Teil der TFUs empfangen ihre Daten von Pad Detektoren. Der Wire-Ram Inhalt wird, wie in Abbildung 6.7, dann entsprechend als Pads angezeigt. Die gesetzten Pads sind farblich hervorgehoben.

Bei der Message Datenstruktur (Abbildung 6.8) werden die Zahlenwerte direkt angezeigt. Dabei kann für jeden Wert zwischen binärer, hexadezimaler oder dezimaler Schreibweise gewählt werden. Außerdem können neue Zahlenwerte eingegeben werden. Dabei überprüft das Programm jeweils, ob der, durch die Anzahl der Bits einer Variable, festgelegte Maximalwert nicht überschritten wurde.

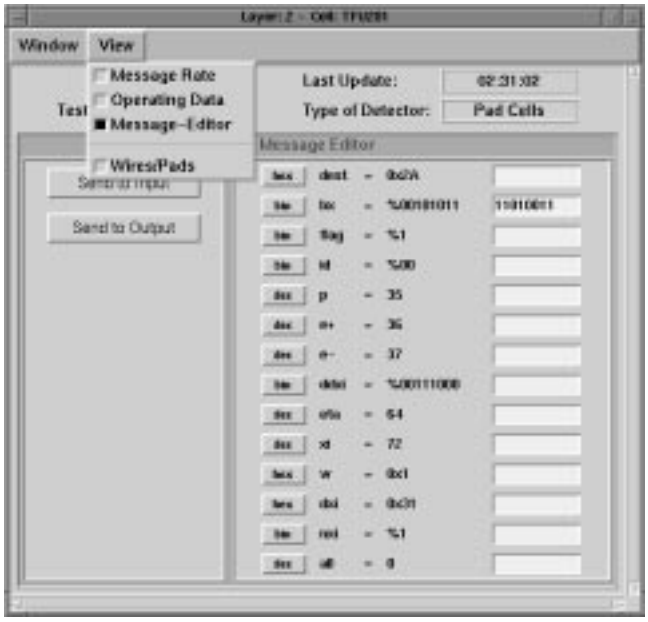


Abbildung 6.8: *Anzeige- und Eingabemaske der Message Datenstruktur. Das Zahlensystem der Einträge kann gewählt werden.*

## 6.4 Netzwerkkommunikation

### 6.4.1 Anforderungen an die Netzwerkkommunikation

Beim Start des Triggersystems muß der Kontrollprozeß als erstes via Netzwerk die benötigten Prozesse auf den LynxOS Rechnern starten können. Dies ist problemlos über eine Unix „remote shell“ möglich. Sie wird mit dem Unix Kommando `rsh` aufgerufen und erhält den Namen des Hosts und das dort zu startende `trun`-Programm als Parameter.

Mit den gestarteten `trun`-Prozessen werden anschließend über das Netzwerk Daten ausgetauscht. Dabei sind folgende Anforderungen und Randbedingungen zu beachten:

- Eine zuverlässige Übertragung der Daten.
- Eine effiziente Programmierung der Kommunikation.
- Es werden nur kleine Datenmengen bei kleiner Datenrate übertragen. Im wesentlichen handelt es sich um Steuerkommandos und Statusinformationen des Multiprozessors.
- Die Übertragung ist nicht zeitkritisch.

Die Netzwerkkommunikation mit Sockets, und im speziellen die Tcl Sockets, wurde bereits in Abschnitt 4.5 besprochen. Das von den Tcl Sockets verwendete TCP Netzwerk Protokoll mit verbindungsorientiertem Bytestrom geht vollständig konform mit den oben genannten Anforderungen. Im folgenden wird nun besprochen, wie die beteiligten Prozesse miteinander in Verbindung treten und dann Daten austauschen können.

### 6.4.2 Client/Server

Bei der Netzwerkprogrammierung wird zwischen Client und Server unterschieden. In dem vorliegenden Fall tritt `tricon` als Serverprozeß auf, der mit vielen `trun`-Clients verbunden sein kann (Abbildung 6.9) .

Beim Start erzeugt der Server als erstes einen Anfrage-Socket. Dieser Socket ist einem bestimmten Port zugeordnet, dessen Nummer, neben der IP-Nummer des Servers, den Clients bekannt sein muß. Ein Client kann dann über diesen Anfrageport mit dem Server Kontakt aufnehmen. Der Anfrage-Port kann nicht für schreiben oder lesen genutzt werden, seine einzige Funktion ist das Entgegennehmen der Verbindungsanfragen von Clients. In Abbildung 6.9 ist er als Port a bezeichnet.

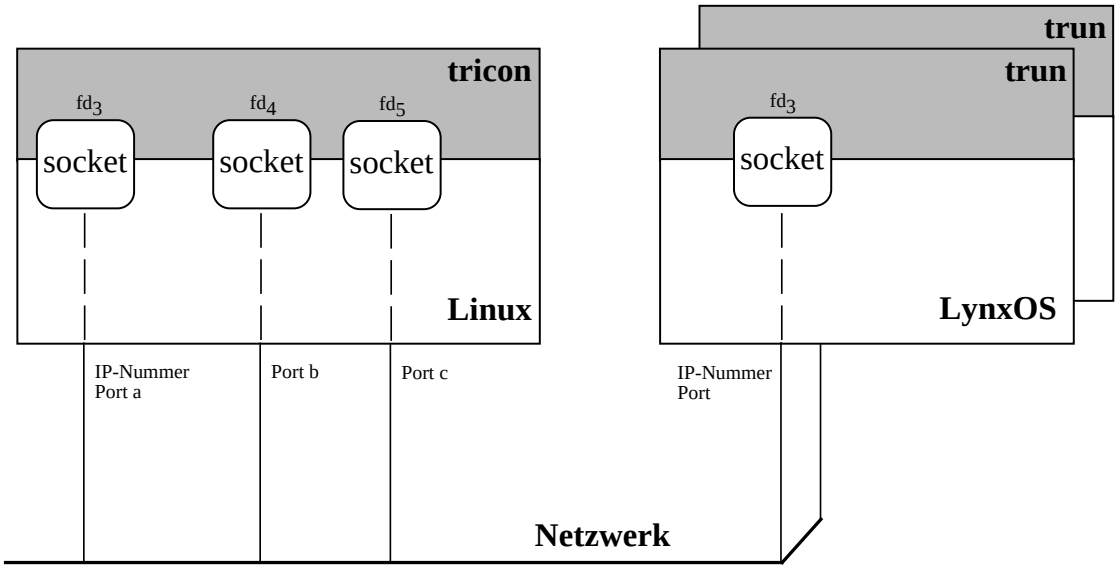


Abbildung 6.9: *Clients nehmen über den Anfrage-Port a Verbindung mit dem Serverprozeß* `tricon` *auf. Anschließend erzeugt* `tricon` *einen Socket, über den dann die Verbindung zu dem* `trun`*-Client gehalten wird. Von den Programmen aus werden die Sockets über Dateideskriptoren angesprochen.*

Wenn der Serverprozeß bereit ist, seinen Port auf Anfragen von Clients abzuhöhren, geht er in die Tcl Event-Loop. Das Ereignis einer Client-Anfrage wird zuvor an eine Ereignis-Routine gebunden. Nimmt ein Client Verbindung auf, so verläßt der Server

den Event-Loop und springt in die Ereignis-Routine. Dort erzeugt er einen neuen Socket, der über einen anderen Port, dessen Nummer dem Client mitgeteilt wird, dann die Verbindung zu diesem Client hält (vergleiche Abbildung 6.9 Port b und c). Damit bleibt der Server-Port für Anfragen von weiteren Clients offen. `Tricon` kann auf diese Weise beliebig viele Verbindungen zu `trun`-Prozessen aufbauen. Die einzige Beschränkung stellt die maximal erlaubte Anzahl geöffneter Dateien dar, die einem Prozeß zur Verfügung stehen, bei Linux sind dies 256.

Die Netzwerk-Schnittstelle der `trun`-Clients, die unter LynxOS betrieben werden, ist durch die Einbindung eines Tcl-Interpreters in das C-Programm `trun` realisiert. Siehe hierzu Abschnitt 5.4.

## 6.5 Zusammenfassung

Im Mittelpunkt dieses Kapitels steht der zentrale Kontrollprozeß, mit dem der gesamte First Level Trigger während des Betriebs gesteuert werden soll. Der Kontrollprozeß muß eine einfache Sicht auf das komplexe Triggersystem liefern, um dessen Bedienbarkeit zu erleichtern. Aus diesem Grund ist er mit einer grafischen Benutzeroberfläche versehen. Das Entwicklungswerkzeug der Wahl für die Oberfläche ist Tcl/Tk, weil es als Skriptsprache einen hohen Abstraktionsgrad liefert und über die Oberflächenentwicklung hinaus, sehr gut für das Handling von Konfigurationsdateien und für Netzwerkkommunikation eingesetzt werden kann.

Die Oberfläche präsentiert momentan zwei verschiedene Ansichten des Gesamtsystems. Über die grafischen Prozessorsymbole gelangt man jeweils zu der Anzeige der Betriebsparameter eines Prozessorboards. Weiterhin lassen sich von dem FLT verwendete Datenstrukturen wie Wire-RAM und Message darstellen.

Das Tcl Socket Paket, daß das TCP-Protokoll und verbindungsorientierten Bytestrom verwendet, wird von dem Kontrollprozeß für die Kommunikation mit den `trun`-Prozessen auf den VME-Host Rechnern eingesetzt. Der Kontrollprozeß fungiert als Server, über dessen Anfrage-Port `trun`-Clients eine Verbindung aufbauen können.

# Kapitel 7

# Ein objektorientiertes Framework zur Simulation der Triggersysteme

Für Funktionstests der Triggerboards und die Inbetriebnahme des Triggersystems werden Simulationen der einzelnen Hardwarekomponenten und ihres Zusammenwirkens benötigt. Auch für physikalische Fragestellungen braucht man eine Simulation des Gesamtsystems. Sie betreffen die Bestimmung von Effizienzen zur Interpretation der Messungen und die diesbezügliche Optimierung des Triggersystem. Die Simulation des First Level Triggers und der drei vorgeschalteten Pretriggersysteme soll mit einem gemeinsamen Simulationsprogramm des Gesamtsystems möglich sein. Dabei ist man sowohl an den Prozeß Ergebnissen, als auch an dem zeitlichen Verhalten des asynchronen Multiprozessorsystems interessiert. Diese Gesamtsimulation soll auf den Simulationen der einzelnen Komponenten, die durch die Hardwaretests mit den realen Komponenten abgeglichen werden, aufbauen. Daher muß sowohl im Detail der Zustand einzelner Boards, als zum Beispiel auch das Zeitverhalten des asynchronen Gesamtsystems simuliert werden.

Die besondere Herausforderung dieser Simulation ergibt sich, neben der Größe des zu simulierenden Systems, daraus, daß viele Personen, die auf mehrere Institute verteilt sind, gemeinsam an einem Programm arbeiten. Weiterhin sollen auch Entwickler ohne spezielle Hardwarekenntnisse, die simulierten Komponenten (Triggerboards) als Bausteine verwenden können, um damit spezielle Triggerszenarien zu simulieren.

Um diesen Anforderungen gerecht zu werden, wurde ein objektorientiertes Framework entwickelt. Mit Hilfe des Frameworks, das die generelle Architektur der Anwendungen festlegt, entwickelt jede Gruppe für sich eine Simulation ihres speziellen Subsystems.

Frameworks bieten den höchsten Grad an Wiederverwertung von Programmcode, was in einer erwünschten Reduzierung des Entwicklungsaufwands resultiert. Mit der wichtigste Aspekt des Frameworks ist aber, daß seine Verwendung die Entwurfsfrei-

heiten der einzelnen Entwicklergruppen hinreichend einschränkt, damit die getrennt entwickelte Software tatsächlich in einer einzigen Applikation zusammengefügt werden kann. Die größte Schwierigkeit ist dabei, die Architektur des Frameworks flexibel genug zu halten, um den Anforderungen (auch zukünftigen) aller Subsysteme gerecht zu werden.

Der Entwurf und die Implementierung des Frameworks sind der Inhalt dieses Kapitels. Nachdem zu Beginn die Anforderungen an die Simulation(en) betrachtet werden, gliedert sich der Rest in drei Abschnitte: Modellentwurf, Auswahl der geeigneten Softwaretechnologie und Implementierung. Im Modellentwurf wird eine Systemanalyse durchgeführt und aus den verschiedenen Ansätzen für Simulationsmodelle ein geeigneter Modelltyp ermittelt, mit dem das reale System abgebildet werden kann. Anschließend wird eine geeignete Softwaretechnologie ausgesucht. Bei der Implementierung wird, unter Berücksichtigung der Anforderungen, die gewählte Softwaretechnologie eingesetzt, um das Modell in ein Computerprogramm - das Framework - umzusetzen.

## 7.1 Anforderungen an die Simulationen

Nach einem generellen Überblick über den Bedarf an Simulationen im Rahmen von Entwicklung und Betrieb des Triggersystems wird die Ausgangssituation beschrieben, aus der heraus die Durchführung des Simulationsprojektes FLTSIM beschlossen worden ist. Daran schließen sich die generellen Anforderungen an die Simulation an.

### 7.1.1 Der Simulationsbedarf

Simulationen werden in drei Bereichen benötigt:

1. *Hardware Tests.* Für den Test von TFU, TPU und TDU wird eine Simulation dieser Boards benötigt. Aufgrund ihrer Komplexität ist es nicht möglich diese Schaltungen „von Hand“ zu testen. Es müssen Dauertests mit einer großen Anzahl von Testdaten durchgeführt werden. Diese Testdaten werden auch von der Simulation verarbeitet und ihre Ergebnisse mit den Resultaten des realen Systems verglichen. Insbesondere für die TFU, die das komplexeste Board ist und von der mindestens 75 Exemplare produziert werden, muß eine komfortable Testumgebung entwickelt werden.

   Simulationen für Hardwaretests ist eine Anforderung, die nur von der Mannheimer Gruppe gestellt wird, die Pretriggergruppen testen ihre Hardware mit eigenen Methoden.

2. *Simulation des Gesamtsystems.* Um die Meßergebnisse des Experiments zu interpretieren, ist die Kenntnis der Effizienz des Detektors, und damit auch des

Triggersystems, unbedingt notwendig. Da der Detektor nicht mit einem Referenzsignal geeicht werden kann, muß die Effizienz über Simulationen bestimmt werden.

Weiterhin muß das Verhalten des Gesamtsystems studiert werden, zum Beispiel die Latenzzeiten, Szenarien mit nur einem Teil der Detektoren und neue Trigger.

Mit Hilfe der Simulation sollen Subprojekte in einen brauchbaren Status gebracht werden noch bevor die Hardware existiert. Beispielsweise können die LUTs oder die Verteilung der TFUs auf die Detektorlagen entwickelt und überprüft werden.

3. *Inbetriebnahme des Triggers.* Hier wird die Simulation als Diagnoseinstrument für die Inbetriebnahme verwendet, um ein gegebenes Systemverhalten unter bestimmten Betriebsbedingungen zu verstehen und Fehler zu entdecken. Anhand der Simulation kann man dann untersuchen wie einzelne Betriebsparameter verändert werden müssen, um den Betrieb des Triggersystems zu optimieren.

### 7.1.2 Die Ausgangssituation und Grundsatzentscheidungen

Bereits vor Beginn des FLTSIM-Projektes lagen Erfahrungen mit Simulationen des Triggers vor. Zum einen gibt es das Programm „l1simu“ am DESY und zum zweiten eine in Mannheim entwickelte TFU-Simulation.

L1simu ist eine Simulation des gesamten First Level Triggers, die am DESY in Hamburg als Fortran-Programm entwickelt wurde. Die Simulation ist datenorientiert und berücksichtigt nicht das Zeitverhalten des Triggersystems. L1simu wurde zum Design des First Level Triggers benutzt und sukzessive mit dem Design weiterentwickelt. Es wurde über mehrere Jahre hinweg immer wieder verändert, da neue Spezifikationen getestet wurden und neue Anforderungen hinzukamen. Die Schwerpunkte lagen dabei eher auf physikalischen Fragestellungen, wie zum Beispiel Effizienzen für verschiedene Zerfallskanäle und Einfluß von Topologie und Auflösung des Detektors. Das Programm hatte einen derart schlechten Zustand erreicht, daß von Mitgliedern der DESY-Gruppe eine Neuentwicklung gefordert wurde. Vor allem auch im Hinblick darauf, daß ein Simulationsprogramm während der gesamten Laufzeit des Experiments (ca. 5 Jahre) immer wieder benötigt wird.

Im einzelnen sind folgende Probleme vorhanden:

- Außer dem Hauptentwickler ist niemand mehr in der Lage mit dem Programm richtig umzugehen, geschweige denn es weiterzuentwickeln. Verschiedene Benutzer kommen mit der Simulation zu unterschiedlichen Resultaten.

- Es existiert keinerlei Dokumentation, selbst Kommentarzeilen im Code beschreiben teilweise irgendwelche Vorversionen und nicht die aktuelle Programmversion.

- Der Triggeralgorithmus ist sehr hardwarefern implementiert, was schon während der Designphase der Triggerhardware immer wieder zu Problemen geführt hat, da Simulation und simuliertes System nicht übereinstimmen.

- Das für eine genauere Simulation benötigte Zeitverhalten in l1simu einzubauen käme einer Neuentwicklung gleich, ist aber entscheidend für Effizienzbestimmungen.

Die Mannheimer TFU-Simulation wurde im Rahmen dieser Arbeit in C++ geschrieben und ist ebenfalls datenorientiert ohne Simulation des Zeitverhaltens. Außerdem wird hier das Verhalten des Multiprozessorsystems mit einer Multi-Prozeß Simulation nachgebildet, indem für jedes zu simulierende Board ein eigener Unix-Prozeß gestartet wird. Die Messages werden dann über Interprozeßkommunikation zwischen den Prozessen verschickt. Für die Myon- und Elektron-Pretriggersysteme existieren keinerlei Simulationen, der Hadron-Pretrigger kann mit einem Pascal Programm simuliert werden.

Diese Situation führte zu dem Entschluß ein völlig neues Programm zu entwickeln. Dabei wurden folgende grundsätzliche Entscheidungen getroffen:

1. Das Gesamtsystem aus drei Pretriggern und dem FLT muß simuliert werden.

2. Es soll nur „eine“ Simulation für die drei Anwendungsbereiche Hardwaretests, Gesamtsystem und Inbetriebnahme geben. Dies soll sicherstellen, daß die reale Hardware so gut wie möglich in der Simulation abgebildet wird. Das bedeutet, für die verschiedenen Anwendungsbereiche sollen keine speziellen, auf die jeweiligen Anforderungen zugeschnittenen Simulationsvarianten geschrieben werden, sondern die Gesamtsimulation soll auf der Software der Einzelboardsimulationen aufbauen. Die Hardwaretests sind gleichzeitig ein Test der Boardsimulation und stellen damit sicher, daß auch in der Gesamtsimulation tatsächlich das reale Verhalten der Hardware simuliert wird.

3. Jede Gruppe ist für die Simulation ihres Subsystems verantwortlich.

4. Das zeitliche Verhalten des Triggersystems muß simuliert werden.

5. Durch den Einsatz von Software Engineering Methoden und Entwicklungswerkzeugen soll eine gute Programmarchitektur und Dokumentation sichergestellt werden.

### 7.1.3 Generelle Anforderungen

1. Die verschiedenen Entwickler des Simulationsprogramms, sind auf mehrere Orte verteilt. Um trotzdem eine erfolgreiche Zusammenarbeit zu ermöglichen, wird ein Managment und eine Software Entwicklungs Technologie zur Koordinierung benötigt.

2. In den verschiedenen Bereichen, in denen Simulationen eingesetzt werden, liegen zum Teil auch unterschiedliche Schwerpunkte bezüglich der Anforderungen vor:

   **Bei den Hardwaretests** ist man an einem detaillierten Vergleich zwischen dem Zustand des Boards und seinem simulierten Pendant interessiert. Hier ist also ein Vergleich auf Bit-Niveau zwischen den simulierten und den realen Prozeßergebnissen, auf die man für Testzwecke Zugriff hat, angestrebt. Dies erhöht die Komplexität der Simulation gegenüber einem einfachen Nachprogrammieren des Algorithmus erheblich. Zum Beispiel kann ein mehrstufiger Addierer nicht einfach mit einer „+“ Operation simuliert werden, sondern muß logisch so implementiert werden, daß er auch die Ergebnisse der Zwischenschritte produziert.

   **In Simulationen zur Inbetriebnahme** finden Vergleiche zwischen realem und simuliertem System eher auf dem Niveau des Gesamtsystems oder dem Messageaustausch zwischen Boards statt.

   **Simuliert man das Gesamtsystem** so gilt das Interesse hauptsächlich den Ergebnissen. Mit den Hardwaredetails, die man unter Umständen gar nicht kennt, will man dabei so wenig wie möglich konfrontiert werden.

   Es muß also ein Weg gefunden werden, die detaillierte Hardwaresimulation so in die Simulation einzubauen, daß man sie verwenden kann ohne sich um Details kümmern zu müssen. Außerdem muß trotz der simulierten Details auch eine Gesamtsimulation noch schnell genug sein, um in vertretbarer Zeit Ergebnisse zu erhalten.

3. Ein Teil des Codes wird sowohl für die Simulation als auch später für die Betriebssoftware benötigt. Dies betrifft beispielsweise die LUTs und die Berechnung und Verwaltung der Geometriedaten, die als Parameter für die LUTs benötigt werden. Von diesen Code Teilen soll es nur eine Version geben, die in verschiedenen Bereichen eingestzt werden kann.

4. Die Simulation benötigt einen Zugriff auf ARTE-Daten. ARTE, eine Art Datenbank am DESY, enthält sowohl Informationen über den Detektor als auch simulierte Detektorereignisse, die als Eingangsdaten der Simulation dienen.

## 7.2 Das zeitdiskrete Simulationsmodell

In diesem Abschnitt wird ein für die Simulation des Triggersystems geeignetes Modell entworfen und ein passender Simulationstyp gewählt. Dabei wird in mehreren Schritten vorgegangen. Bei der Analyse wird zunächst das System klassifiziert und die grundlegenden Abstraktionen identifiziert, mit denen das Triggersystem beschrieben werden kann. Anschließend wird aus den bekannten Modelltypen das geeignete ausgewählt und das reale System in ein konzeptionelles Modell abgebildet. Im letzten Schritt wird aus den verschiedenen, für diesen Modelltyp in der Literatur beschriebenen, Simulationsarten eine Auswahl getroffen.

### 7.2.1 Klassifizierung des Systems

Zur Systemanalyse werden zunächst einige Grundbegriffe aus der Systemtheorie eingeführt. Mit diesen Basisabstraktionen können die Struktur und das Verhalten von Systemen allgemein beschrieben werden [38].

Ein System wird aus einer Menge von Elementen (siehe Abbildung 7.1) gebildet, die als nicht weiter teilbar angesehen werden. Systeme können hierarchisch aufgebaut sein, indem Subsysteme wiederum als Elemente betrachtet werden.

Zwischen den Elementen bestehen Beziehungen und sie besitzen Attribute, die ihre Eigenschaften beschreiben. Die Summe aller Attribute beschreibt den Zustand eines Systems.

Das Verhalten eines System wird durch eine Abfolge von Zuständen in der Zeit beschrieben.

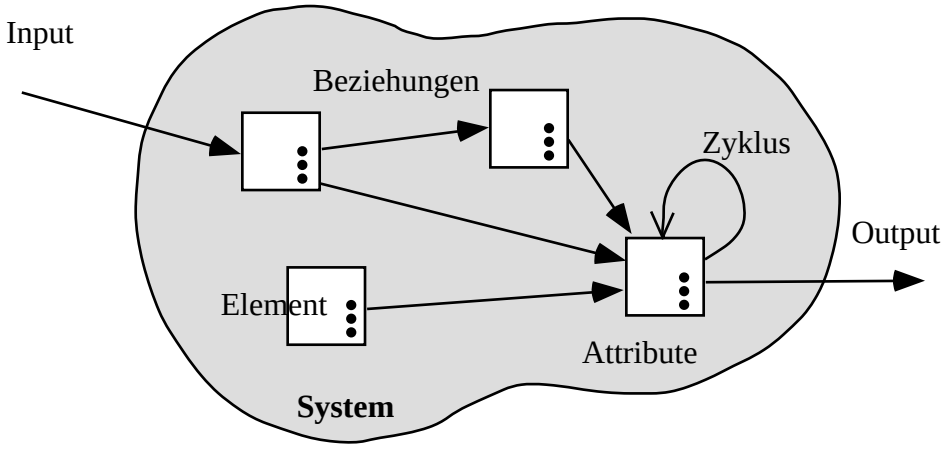


Abbildung 7.1: *Grundlegende Systembegriffe. [38]*

Weitere Charakteristika ergeben sich aus der Art der Beziehungen eines Systems zu seiner Umgebung und seinem Verhalten in der Zeit. Das Triggersystem erhält seine Eingangsdaten von dem Detektor und gibt seine Ergebnisse an den Second Level

Trigger weiter. Da weder der Detektor noch der Second Level Trigger Bestandteil des zu simulierenden Systems sind, handelt es sich um ein offenes System. Der Zustand des Triggersystems ändert sich mit der Zeit, es ist daher auch ein dynamisches System. Entsprechend sind die Werte der Attribute mit einem Zeitindex versehen. Als kybernetische Systeme werden dynamische Systeme mit Rückkopplungen (Zyklen) bezeichnet. Dies ist zum Beispiel bei einem Stall-Takt der TFU der Fall, wo in Abhängigkeit von den Rechenergebnissen einer Pipeline-Stufe, die davorliegenden Stufen angehalten werden.

Zusammenfassend handelt es sich bei dem System aus Pretriggern und FLT also um ein offenes, dynamisches und kybernetisches System.

Nun gilt es, die grundlegenden Elemente und ihre Beziehungen untereinander in der vorliegenden konkreten Aufgabenstellung zu identifizieren.

Die Funktionalität der Triggersubsysteme wird ausschließlich von digitalen elektronischen Bauelementen wahrgenommen. Diese Bauelemente werden, von wenigen Ausnahmen abgesehen, über einen Taktgeber (Clock) synchronisiert. Die Bauelemente sind jeweils zu verschiedenen Boards gruppiert und werden dort von einem oder mehreren lokalen Taktgebern gesteuert. Die Boards untereinander sind für den Datenaustausch über verschiedene Schnittstellen verbunden.

Es wird also eine Simulation benötigt, mit der man allgemein digitale elektronische Systeme beschreiben kann. Dabei ist nach den Anforderungen nur eine logische Simulation notwendig, physikalische Eigenschaften der Bauelemente werden nicht simuliert. Dementsprechend werden für die Simulation zwei Basiselemente abstrahiert, aus denen ein zu simulierendes System aufgebaut wird: „Schaltkreise“ und „Boards“. Ein Schaltkreis hat zwei grundlegende Eigenschaften:

1. Ein Eingangsbitmuster wird über eine Transformation auf ein Ausgangsbitmuster abgebildet.

2. Dies passiert mit jedem Taktzyklus.

Jedes Schaltkreiselement muß daher von einem Taktgeber gesteuert werden. Ein Taktzyklus hat eine feste Länge. Nach einer bestimmten Zeit werden die Prozeßresultate eines Schaltkreises an seinem Ausgang gültig und damit an den Eingängen der mit ihm verbundenen Schaltkreise. Für die Simulation sollen Schaltkreise zusammengesetzt und diese Gruppe wiederum als ein einzelner Schaltkreis betrachtet werden können.

Ein Board enthält eine Ansammlung von Schaltkreisen, die fest miteinander verbunden sind. Sie interagieren, um das Verhalten eines realen Elektronikboards zu simulieren. Das Board selbst ist aber ausdrücklich kein Schaltkreis. Es ist nicht getaktet und kann auch weitere Objekte, zum Beispiel boardbezogene Datenstrukturen, die bei dem realen Board der Software zuzuordnen sind, enthalten.

Nach außen besitzen die Elemente die gleichen Schnittstellen wie ihr reales Pendant. Die nach außen sichtbaren Attribute sind daher der Zustand ihrer Ausgangsbits. Hinzu können natürlich beliebige interne Attribute kommen.

Die Beziehungen zwischen den Elementen, Schaltkreisen und Boards, sind immer elektrische Verbindungen. Die Schaltkreise sind über ihre Ein- und Ausgangsbusse miteinander verbunden. Da es sich dabei um eine feste Verdrahtung handelt, sind auch die Beziehungen der Schaltkreiselemente untereinander statischer Natur. Die Boards können je nach Konfiguration des zu simulierenden Triggersystems unterschiedlich miteinander verbunden werden. Während eines Simulationslaufs sind die Verbindungen der Boards ebenfalls fest.

### 7.2.2 Das Modell des Triggersystems

Erkenntnisse über Systeme können mit analytischen Berechnungen oder durch Simulation gewonnen werden. Analytische Modelle ermitteln durch Lösen eines Gleichungssystems den Zustand eines System aus gegebenen Randbedingungen. Sie sind aufgrund mathematischer Restriktionen in der Regel auf Systeme geringer Komplexität begrenzt.

In Simulationsmodellen dagegen ist die Komplexität nur durch die Resourcen des Simulationsrechners begrenzt. In so einem Modell wird der Systemzustand Schritt für Schritt verändert und die Zwischenschritte entsprechen Zwischenzuständen des Originals. Aus diesem Grund eignen sich Simulationsmodelle besonders gut zur Veranschaulichung des Systemverhaltens. Da der Trigger ein sehr komplexes System ist, an dessen Verhalten wir interessiert sind ist es klar, daß für die Triggersimulation kein analytisches, sondern nur ein Simulationsmodell in Frage kommt. Es muß zudem ein dynamisches Modell sein, um das Systemverhalten abbilden zu können.

Das Simulationsmodell kann weiter nach der Art der Zustandsübergänge klassifiziert werden (Abbildung 7.2). Ein System wird als deterministisch bezeichnet, wenn bei einem gegebenen Zustand der Übergang in den nächsten Zustand eindeutig festgelegt ist. Die Zustandsfolge ist damit bei gegebenen Eingangsdaten eindeutig vorhersagbar. Die einzelnen synchronisiert arbeitenden Komponenten des Triggersystems verhalten sich in diesem Sinne deterministisch. Das Gesamtsystem besitzt jedoch kein deterministisches Verhalten, da es aus asynchron arbeitenden Hardwarekomponenten aufgebaut ist. (Jedes Board hat seine eigenen Taktgeber.) Die Phase zwischen den verschiedenen Taktgebern ist unbestimmt, sie ist nach jedem Einschalten des Triggers anders. Daher ist überall dort, wo zwei asynchrone Komponenten aufeinandertreffen, um Daten auszutauschen, nicht genau vorhersagbar mit welchem Taktzyklus des Empfängers die gesendeten Daten übernommen werden. Damit ist zum Beispiel die Reihenfolge in der Messages, die von verschiedenen Quellen stammen, von einer TFU bearbeitet werden, nicht vorhersagbar. Dies kann sich bei Überschreiten der Latenzzeit direkt auf das Ergebnis der Triggerentscheidung auswirken. Je

nach Reihenfolge kann es sein, daß auf eine Spur noch getriggert wird, oder daß das Ereignis verworfen wird, weil die Spur zu spät eintrifft.

Da bereits das digitale Original diskret arbeitet, und zwar sowohl bei den Werten seiner Attribute, als auch zeitlich, in Vielfachen von Taktzyklen, ist außerdem die Verwendung eines diskreten Modells sinnvoll. Im Gegensatz zu beispielsweise biologischen oder meteorologischen Systemen, wo sich Systemveränderungen kontinuierlich vollziehen und kontinuierliche Attributwerte vorliegen, kann hier mit einer expliziten Diskretisierung der Zeit gearbeitet werden.

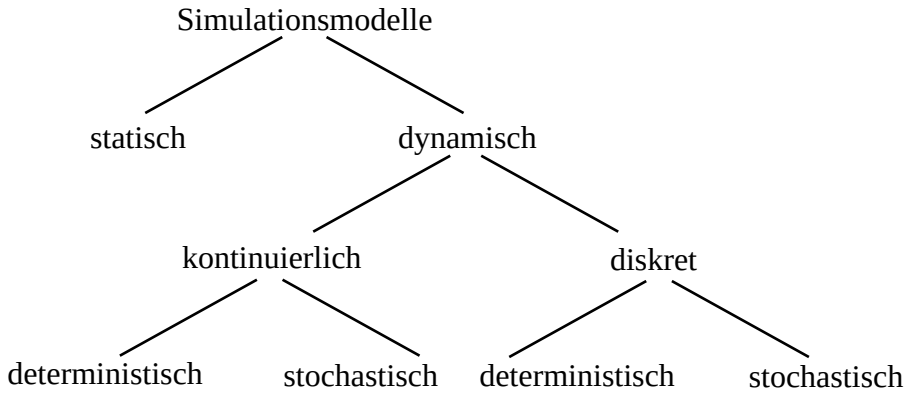


Abbildung 7.2: *Die Klassifizierung von Modellen. [38]*

### Modellbildung

In dem Prozeß der Modellbildung wird das reale System in ein Modell abgebildet, an dem dann Untersuchungen durchgeführt werden können. Dieser Vorgang geht immer mit einer Abstrahierung und einer Idealisierung des Systems und seiner Elemente einher. Welche Aspekte des realen Systems dabei vernachlässigt werden können hängt von den Fragestellungen an das Modell ab. Die Abbildung muß hinreichend genau sein, so daß man die interessierenden Abläufe des realen Systems nachvollziehen kann.

In einem ersten Schritt muß also die Frage geklärt werden, welche Aspekte des realen Systems vernachlässigt werden können, ohne dabei die gestellten Anforderungen an die Simulation außer acht zu lassen.

Für die Triggersimulation werden elektronische Eigenschaften der Hardware, wie zum Beispiel Signalanstiegszeiten und das Übersprechen zwischen Leitungen nicht berücksichtigt. Auch die spezifischen Eigenschaften der Logikbausteine, wie beispielsweise Clock-to-Output Zeiten, werden nicht simuliert. Bei den Schaltkreis- und Board-Elementen handelt sich also um eine reine Logiksimulation auf Bit-Niveau mit diskreten Zeitschritten.

Die wichtigsten Idealisierungen gegenüber dem realen System betreffen die Taktgeber. Für die gesamte Simulation gibt es nur einen zentralen Taktgeber. Er ist zugleich identisch mit der Simulationsuhr und taktet die Schaltkreise mit ihren jeweiligen Frequenzen. Von dem zentralen Taktgeber wird die Phase zwischen zwei unabhängigen Clock-Signalen nicht berücksichtigt. Dies ist zulässig, da die Phasendifferenzen zwischen den Takgebern alle gleichberechtigt sind. Es wird willkürlich ein Satz von Phasendifferenzen herausgegriffen (der Einfacheit halber die Differenz Null) und für alle Simulationen verwendet. Der Triggersimulation wird damit ein deterministisches Modell zugrunde gelegt. Die Simulationszeit wird immer bis zu dem nächsten Taktsignal erhöht, in der Zeit zwischen zwei Taktsignalen passiert konzeptionell nichts.

Die Zugriffsmöglichkeiten von Benutzern während des Triggerbetriebs sind nicht Bestandteil der Simulation. Es werden nur Eingangsdaten von dem Detektor verarbeitet und die Triggerentscheidung für den Second Level Trigger generiert. Damit ist auch bereits die Abgrenzung des zu modellierenden Systems von seiner Umgebung vorgenommen.

Zusammenfassend ergibt sich daraus, daß sich das offene, dynamische, kybernetische Triggersystem mit mit Hilfe eines dynamischen, diskreten und deterministischen Simulationsmodells beschreiben läßt.

### 7.2.3 Zeitdiskrete Simulationsmodelle

In der diskreten Simulation haben sich verschiedene sogenannte „Modellierungsstile“ oder „Weltbilder“ herausgebildet (Abbildung 7.3). Sie unterscheiden sich hauptsächlich in ihrer Sichtweise und ihrer Darstellung des dynamischen Verhaltens des Systems, also der Beziehung zwischen Systemzustand und Simulationszeit.

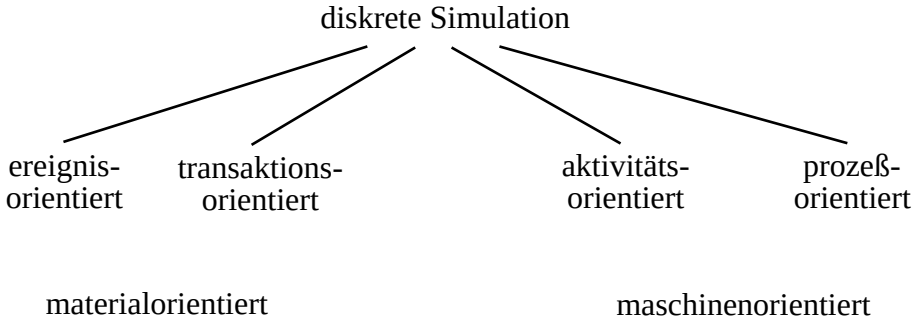


Abbildung 7.3: *Diskrete Simulation. [38]*

Dabei wird zusätzlich zwischen materialorientierten und maschinenorientierten Konzepten unterschieden. Bei den materialorientierten Ansätzen stehen die Materialflüsse durch die Modellelemente (bei dem Trigger Datenflüsse) und die dabei zurück-

gelegten Wege im Vordergrund. In der maschinenorientierten Sichtweise liegt der Schwerpunkt auf dem Bearbeitungsvorgang selbst (bei dem Trigger die elektronischen Baugruppen). Die beiden wichtigsten Modellierungsstile sind die prozeß- und die ereignisorientierte Simulation.

### Ereignisorientierte Simulation

Unter einem Ereignis ist die Änderung des Zustands von mindestens einem Modellelement zu verstehen. Die Beschreibung erfolgt in Form einer Ereignisroutine.

In der ereignisorientierten Simulation wird eine Liste mit zeitlich geordneten Ereignissen abgearbeitet. Dabei wird jeweils das Ereignis mit dem kleinsten Zeiteintrag ausgewählt. Die Simulationszeit springt entsprechend von Ereigniszeitpunkt zu Ereigniszeitpunkt und zu jedem Ereignis wird eine zugehörige, das Ereignis beschreibende Ereignisroutine aufgerufen. Zeitlich ausgedehnte Aktivitäten werden auf eine Folge von Ereignissen reduziert, wie zum Beispiel Beginn und Ende einer Aktivität. Auf elementarer Ebene entspricht eine Simulation von Systemabläufen immer einer Ereignisfolge.

### Prozeßorientierte Simulation

In prozeßorientierten Simulationsmodellen wird jedem Modellelement ein eigener „Prozeß“ zugeordnet, der die auf ein Element bezogenen Aktivitäten und Attribute in ihrer Gesamtheit zusammenfaßt und die Zustände des Elements in ihrer zeitlichen Abfolge beschreibt. Prozesse können in aktivem oder inaktivem Zustand sein.

Ein aktiver Prozeß wird bei Aufruf seiner Prozeßroutine so weit wie möglich vorangetrieben, bis er zum Beispiel auf das Ergebnis einer anderen Aktivität warten muß. (Das dabei ausgeführte Teilstück des Prozesses ist mit einer Ereignisroutine vergleichbar.) Prozesse können in diesem Fall dann in einen inaktiven Zustand übergehen. Die Ausführung kann zu einem späteren Zeitpunkt fortgesetzt werden. Der Prozeßzustand wird als lokale Attribute des jeweiligen Modellelements gespeichert, das die Fortführung seines Prozesses bei erneuter Aktivierung dann an der richtigen Stelle vornimmt.

Die Triggersimulation simuliert ein komplexes System mit komplexen Wechselwirkungen. Große Teile des Systems arbeiten synchron, weshalb eine große Zahl von Ereignissen gleichzeitig stattfindet. In solchen Fällen ist eine prozeßorientierte Simulation vorzuziehen, da bei ihr die Aufteilung logisch zusammengehöriger Abläufe auf verschiedene Ereignisroutinen entfällt. Der Trigger wird mit Hilfe einer prozeßorientierten Simulation simuliert.

Zudem korrespondiert die Zusammenfassung von Aktivitäten und Attributen und die lokale Speicherung des Prozeßzustands der prozeßorientierte Simulation besser

mit der objektorientierten Softwaretechnologie, als die Ereignisroutinensammlung der ereignisorientierten Simulationsmodelle.

## 7.3 Die eingesetzte Software Technologie

Das Simulationsprogramm FLTSIM, in dem das zeitdiskrete, prozeßorientierte Simulationsmodell in ein Programm umgesetzt ist, stellt für sich bereits eine anspruchsvolle Anwendung dar. Es simuliert ein komplexes System und besitzt einen Code-Umfang, der eine Größenordnung von 100000 Programmierzeilen erreichen wird. Hinzu kommen die, in den Anforderungen beschriebenen, schwierigen Randbedingungen für die Entwicklung. Dies macht es besonders wichtig, die richtige Software Technologie nach dem Stand der Technik einzusetzen. Im folgenden wird die Software Technologie beschrieben, mit der den Anforderungen begegnet wird, um das Programm erfolgreich zu implementieren und auch in Zukunft weiterentwickeln zu können.

Der Simulation liegt ein objektorientierter Programmentwurf zugrunde. Das Design des Programms erfolgt mit der objektorientierten Methode von Booch. Für eine kohärente Entwicklung der Simulation der verschiedenen Trigger-Subsysteme wird ein Framework entwickelt und dem Programm zugrunde gelegt. Es dient als Grundlage für die Entwicklung der einzelnen Subsystem- Simulationen und seine Architektur beinhaltet objektorientierte Entwurfsmuster. Dieser Abschnitt schließt mit einer Einführung in Frameworks und einer Begründung des Einsatzes für FLTSIM. Das Simulations-Framework wird in dem folgenden Abschnitt dann genauer beschrieben.

### 7.3.1 Objektorientierung

In dem der objektorientierten Technologie zugrundeliegenden Konzept werden Software Systeme als eine Menge zusammenarbeitender Objekte betrachtet. Nach Booch [4] definiert sich die objektorientierte Programmierung folgendermaßen:

> *Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlungen von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet ist.*

Dem objektorientierten Programmier-Paradigma dient das Objektmodell als konzeptionelle Grundlage. Die Hauptelemente dieses Modells, Abstraktion, Kapselung, Modularität und Hierarchie, bieten auch für die Triggersimulation sehr vorteilhafte Eigenschaften.

Kapselung bedeutet die Trennung von Implementierung und Schnittstelle einer Abstraktion. Dies ermöglicht es zum Beispiel, die komplizierte Implementierung der Simulation eines Prozessorboards in einem Board-Objekt zu kapseln. Damit ist es möglich Board-Objekte, die mit viel Detailwissen an einem anderen Institut entwickelt wurden, für eigene Simulationen einzusetzen ohne das Spezialwissen ihrer Implementierung zu besitzen. Lediglich seine Schnittstelle nach außen wird für die Verwendung benötigt. Diese entspricht weitgehend den Schnittstellen des realen Boards.

Durch die Verwendung von Hierarchien, insbesondere Vererbungshierarchien, kann die auf verschiedene Orte verteilte Entwicklung vereinheitlicht werden. Beispielsweise gibt es eine einheitliche Verbindung zwischen allen Arten von Schaltkreis-Objekten und der Simulationsuhr, die durch Vererbung von einem Basis-Schaltkreis zur Verfügung gestellt wird.

Zusätzlich zu den generellen Vorteilen einer objektorientierten Programmstruktur gibt es einige Eigenschaften, die den Bereich der Simulation betreffen.

Die konzeptionelle Nähe zwischen dem objektorientierten Ansatz und dem Simulationsproblem ermöglicht eine direktere Umsetzung des informellen Modells in ein lauffähiges Programm. Dies spart Zeit, erhöht die Transparenz von Modell und Programm und veringert die Gefahr grundlegender Fehler im Programmdesign.

Die große Ähnlichkeit der Ansätze von Systemtheorie und objektorientierter Programmierung ist im Grunde nicht überraschend. Die Systemtheorie versucht die allgemeinen Abstraktionen von Systemen herauszuarbeiten, aus denen sich dann beliebige Systeme aufbauen lassen. Dies wird für einen konkreten Fall dann in ein möglichst anschauliches Modell umgesetzt, was wiederum heißt der menschlichen Denkweise angepaßt. Da die menschliche Wahrnehmung komplexer Systeme eine objektorientierte ist, werden dabei die gleichen Prinzipien wie bei dem objektorientierten Design einer Software angewendet, die die objektorientierte Zerlegung einer Software zum Ziel hat. Mit dem Unterschied, daß das objektorientierte Programmieren sich nur mit komplexen Software-Systemen beschäftigt und die Systemtheorie mit komplexen Systemen im allgemeinen.

Simulation ist daher eine Anwendung, die aus objektorientierten Techniken sehr direkten Nutzen zieht. Es scheint kaum möglich zu sein, für Simulationsprogramme sich eine bessere Struktur auszudenken als diejenige, die direkt dem Muster der Objekte folgt, deren Verhalten simuliert werden soll [33]. Die Objekt-Technologie [1] ist daher als Technologie der Wahl für Simulationen anzusehen und wird deswegen auch für FLTSIM eingesetzt.

[1] In der Software wurde der Begriff Objekt zuerst in der Programmiersprache Simula (1967) verwendet. [4]

### 7.3.2 Objektorientierte Design Methode

Eine Design-Methode dient dazu, die Komplexität bei der Entwicklung zu bewältigen. Sie bringt Disziplin in den Entwicklungsprozeß und ist ein wichtiges Hilfsmittel für die räumlich oder zeitlich getrennt arbeitenden Entwickler. Angesicht der Komplexität und langen Lebensdauer von FLTSIM und der Erfahrungen mit dem Vorgängerprogramm, ist ein sorgfältiges Design des Programms dringend geboten.

Für den objektorientierten Entwurf von Software haben sich die drei Methoden von Grady Booch, Ivar Jacobson(OOSE), und James Rumbaugh(OMT) als wichtigste Vertreter durchgesetzt. Mitte der 90er Jahre haben sich diese drei führenden Methodiker zusammengetan und die Unified Modeling Language (UML) definiert. Sie soll die Stärken der drei Ausgangsmethoden in sich vereinen, die Methodenvielfalt beenden und zu einem Industriestandard werden [2]. Da die Spezifikation der UML (UML 1.0) erst seit Anfang 1997 zur Verfügung steht, konnte sie für die Triggersimulation nicht verwendet werden. Statt dessen wird die Methode von Booch eingesetzt. Ihr Schwerpunkt liegt auf dem Design und der Implementierung, während OMT ihren Schwerpunkt eher bei der Analyse und dem Software Entwicklungsprozeß hat. Da bei der Triggersimulation aber nicht die gesamte Bandbreite der Methode eingesetzt wird, sind in diesem Fall beide Methoden als gleichwertig anzusehen.

Designmethoden besitzen drei wichtige Elemente:

- Die Notation liefert die Sprache um ein Design-Modell zu beschreiben. Die Booch Methode enthält eine grafische Notation, mit der verschiedene Sichten auf das Modell des Systems in Diagrammen dargestellt werden können.

- Der Prozeß beinhaltet die Aktivitäten, die zu der Konstruktion eines Designmodells führen. Die Booch Methode enthält eine Vorgehensweise, mit der einer ausgereiften Entwicklungsorganisation ein vorhersagbarer und wiederholbarer Prozeß zur Verfügung gestellt wird.

- Werkzeuge. Bei der Verwendung einer etablierten Designmethode stehen Software Entwicklungswerkzeuge zur Verfügung, die die Anwendung dieser Methode unterstützen. Diese Werkzeuge sollen durch automatische Unterstützung der Notation die Erzeugung eines Designs erleichtern und führen Konsistenzüberprüfungen des Design-Modells aus.

Für das Design der Triggersimulation wird nur ein Teil der in der Notation vorhandenen Diagrammtypen verwendet, Klassendiagramme und Sequenzdiagramme (vergleiche jeweils Abbildungen 7.4 und 7.8).

[2] Die UML wurde im November 1997 von der Object Management Group als Standard verabschiedet [35].

Von dem in der Methode enthaltenen Prozeß wird kein Gebrauch gemacht, weil dazu leider die organisatorischen Mittel fehlen. Trotzdem hat aber bereits die Definition einer Analysephase, die anschließende Erstellung der ersten Version des Design-Modells und die dann erst beginnende Inplementierung zu einer kontrollierteren Vorgehensweise geführt.

Als Entwicklungswerkzeug wird im Rahmen der Triggersimulation das Programm `Rose` der Firma Rational eingesetzt. Das Programm enthält seit der Version 4.0 vom April 1997 auch einen brauchbaren C++ Code Generator, der ebenfalls für die Triggersimulation verwendet wird. Damit kann direkt aus dem Design-Modell heraus der jeweilige Code erzeugt werden.

### 7.3.3 Entwurfsmuster

Seit der 1995 erfolgten Veröffentlichung des inzwischen zum Standardwerk avancierten Buchs von Gamma et. al. [11] hat die Verwendung von objektorientierten Mustern, insbesondere Entwurfsmustern, bei der Anwendungsentwicklung einen Siegeszug angetreten. Auch in der Triggersimulation werden mehrere Entwurfsmuster auf sehr gewinnbringende Weise eingesetzt.

Ein Entwurfsmuster beschreibt ein Entwurfsproblem und seine praxisbewährte Lösung. Die Beschreibung erfolgt auf architektonischer Ebene, das heißt anwendungs- und sprachenunabhängig. Muster für den allgemeinen Programmentwurf bestehen in der Regel aus drei bis vier Klassen, die auf eine spezifische Art untereinander in Beziehung stehen. Die Wiederverwendung von objektorientierter Software wird damit auf kommunizierende Gruppen von Klassen erweitert.

Der Einsatz von Entwurfsmustern bringt eine Reihe von wichtigen Vorteilen mit sich:

- Mit der Hilfe von Mustern kann das Erfahrungswissen von Experten für neue Applikationen nutzbar gemacht werden. Dieser Punkt ist besonders wichtig, da die an der Triggersimulation beteiligten Entwickler nur wenig Entwurfserfahrung mitbringen.
- Die Muster stellen ein Vokabular zur Verfügung, das eine effiziente Kommunikation unter den Entwicklern erlaubt.
- Wenn man ein Muster, das ein prinzipielles Problem beschreibt, zur Lösung eines konkreten Problems einsetzt, wird man zur Abstraktion von dem konkreten Problem gezwungen. Dies führt zu besseren Lösungen.
- Bei komplexen Problemstellungen ist man gezwungen sie in sauber getrennte Teilprobleme zu zerlegen, da ein Muster immer nur ein Problem beschreibt.
- Für jedes Teilproblem muß man sich genau überlegen unter welchen Randbedingungen und Einflußgrößen die Lösung zum Einsatz kommt.

Als Folge der Verwendung von Entwurfsmustern erhält man Programme, die besser strukturiert, fehlerfreier und flexibler gegenüber Änderungen sind. Besonders wichtig ist der Einsatz von Entwurfsmustern bei der Entwicklung eines Frameworks, da hierfür Entwurfserfahrung und ein flexibles Design benötigt werden.

### 7.3.4 Frameworks

Ein Framework ist eine Musterapplikation, auf deren Grundlage man schnell eine spezifische Applikation innerhalb eines bestimmten Anwendungsbereichs entwickeln kann. Das Framework wird aus einer Menge kooperierender Klassen gebildet, die einen wiederverwendbaren Entwurf für eine bestimmte Art von Software darstellen. Es enthält die Entwurfsentscheidungen, die in seinem Verwendungsbereich, bei dieser Art von Software allgemein anzutreffen sind.

Mit Hilfe von Frameworks erreicht man in objektorientierten Systemen den höchsten Grad an Wiederverwertung, entsprechend schneller lassen sich dann auch die spezifischen Anwendungen entwickeln. Es wird von um einem Faktor zehn geringeren Kosten und Entwicklungszeiten im Vergleich zu denen eines Individualprojektes berichtet [48].

Ein Framework besteht zum einen aus einem abstrakten Anwendungsentwurf, der Aspekte enthält, bezüglich derer sich verschiedene Anwendungen aus einem Bereich gleichen. Dieser Teil - der Hauptteil des Programms - wird beim Erstellen einer neuen Applikation wiederverwendet.

Der zweite Teil enthält die Aspekte, in denen sich diese Anwendungen unterscheiden. Er besteht aus Klassenhierarchien, deren Basisklassen die Gemeinsamkeiten der variablen Aspekte abstrahieren und legt gegebenenfalls weitere Beziehungen zwischen Klassen fest. Ein Entwickler paßt das Framework an seine spezielle Anwendung an, indem er Unterklassen von diesen Basis-Frameworkklassen ableitet und ihre Objekte dann zusammensetzt. Es definiert die Architektur der Anwendung und betont daher mehr die *Entwurfswiederverwendung* als die Codewiederverwertung.

Der zentrale Unterschied zwischen einem Framework und einer normalen Klassenbibliothek ist die unterschiedliche Richtung des Kontrollflusses. Eine Klassenbibliothek stellt Code-Bausteine zur Verfügung, die man vom Hauptteil seines Programms aufrufen kann. Ein Framework jedoch stellt Plätze zur Verfügung, in die man seinen eigenen spezialisierten Code einfügt. Wenn dann das Programm läuft, wird der spezialisierte Code *vom Framework* aufgerufen.

Das Framework selbst zu entwerfen ist kompliziert, da die gewählte Architektur für alle Anwendungen in dem vorgesehenen Bereich verwendbar sein muß. Die wichtigsten Eigenschaften eines Frameworks sind daher Flexibilität und Erweiterbarkeit. Durch eine lose Kopplung zwischen dem Framework und der Anwendung reduziert man die Auswirkungen von Änderungen am Framework auf die Applikationen. Um die geforderte Flexibilität zu erreichen, ist die Verwendung von Entwurfsmustern

unbedingt zu empfehlen [47]. Vor allem auch, weil mit ihrer Hilfe die Entwurfserfahrung anderer Entwickler genutzt werden kann.

Bei Frameworks kommt der Dokumentation ein besonderer Stellenwert zu. In der Regel erfordern sie einen erheblichen Einarbeitungsaufwand, bevor man sie einsetzen kann. Auch hier sind Entwurfsmuster sehr hilfreich, da sie den Überblick über die Strukturen des Frameworks erleichtern.

Der wichtigste Punkt in Zusammenhang mit der Triggersimulation ist aber, daß ein Framework die Architektur einer Anwendung festlegt. Mit der Entwicklung eines Frameworks als Basis für die Triggersimulation, werden alle beteiligten Entwickler angehalten die Simulation ihres Subsystems in die vorgegebene Architektur einzupassen.

## 7.4 Das Simulationsframework

Der Simulation liegt ein zeitdiskretes prozeßorientiertes Simulationsmodell zugrunde. Die primäre Aufgabe des Frameworks besteht darin, die Basisabstraktionen dieses Modells umzusetzen und sie in seiner Framework-Schnittstelle für die Entwicklung von Simulationen zur Verfügung zu stellen.

Die Basiselemente des Triggersystems sind Schaltkreise, ihre nach außen sichtbaren Attribute sind die Zustände ihrer Schnittstellen (Datenbusse). Die Beziehungen zwischen den Elementen sind die statischen elektrischen Verbindungen zwischen den Schaltkreisen. Die Schaltkreise können zu Boards gruppiert werden.

Dementsprechend stellt das Framework für den Entwurf anwendungsspezifischer Simulationen drei Vererbungshierarchien zur Verfügung, Schaltkreise, Boards und Ausgangsdaten-Container für Schaltkreise (Abbildungen 7.6, 7.12 und 7.5). Hinzu kommen die zentralen Einheiten des Frameworks, wie zum Beispiel der Zeitgeber für die Simulationszeit, der den anwendungsspezifischen Code aufrufen, ohne an eine spezifische Anwendung angepaßt werden zu müssen.

Das Laufzeitverhalten einer Simualtion ist relativ einfach. Da die zu simulierenden Systeme fest verdrahtet sind, gibt es während des Ablaufs einer Simulation keine dynamische Erzeugung oder Vernichtung von Schaltkreis- oder Board-Objekten. Auch die Verbindungen zwischen Schaltkreisen und Boards sind fest. Beim Start einer Simulation werden daher alle benötigten Schaltkreise und Boards erzeugt und verbunden. Beim anschließenden Ablauf der Simulation werden nur noch Daten zwischen Schaltkreisen ausgetauscht. Zudem gibt es bei dem First Level Trigger auf Board-Niveau keine Rückkoppelungen. Das heißt, der Datenfluß durch das System ist auf Board-Niveau unidirektional (vergleiche hierzu Abbildung 3.3) .

Im folgenden werden zuerst die Abstraktionen gezeigt, die das Framework zum Aufbau von Schaltungen zur Verfügung stellt. Anschließend wird auf den Hauptteil des Framework-Programms, insbesondere die Zeitführung eingegangen. Zum Schluß

werden die Eigenschaften der Board-Klassen beschrieben. Alle Bereiche verwenden objektorientierte Entwurfsmuster.

### 7.4.1 Die Basiselemente zum Aufbau einer Schaltung

Der Schaltkreis ist das Basiselement, aus dem ein Simulationssystem aufgebaut wird. Entsprechend ist in dem Framework eine Basisklasse `Circuit` implementiert, von der alle anderen, spezialisierten Schaltkreis-Klassen abgeleitet werden müssen (siehe auch 7.6).

Allen Schaltkreisen gemeinsame Attribute und gemeinsames Verhalten sind bereits in dieser Basisklasse realisiert, um einen möglichst hohen Grad an Wiederverwendung zu erreichen. Für die Implementierung spezifischer Eigenschaften wird die Struktur zu ihrer Realisierung vorgegeben, um eine einheitliche Entwicklung zu gewährleisten. Nach dem Modell in Abschnitt 7.2.1 hat ein Schaltkreis zwei grundlegende Eigenschaften:

1. Ein Eingangs Bitmuster wird über eine Transformation auf ein Ausgangsbitmuster abgebildet.

2. Dies passiert mit jedem Taktzyklus.

Für die Durchführung der Transformation besitzen alle Schaltkreise eine Funktion `operate()`. Die Funktion `operate()` ist die Prozeßroutine eines Schaltkreises und sie ist natürlich für jede Schaltkreis-Klasse spezifisch. In der Basisklasse ist die Funktion `Circuit::operate()` daher rein virtuell. `Circuit` ist damit eine abstrakte Klasse und bei der Ableitung einer Unterklasse *muß* die Operation in der Unterklasse implementiert werden.

Dem Takten eines Schaltkreises, also einer ansteigenden Flanke des Taktsignals in der Hardware, entspricht in der Simulation der einmalige Aufruf der Funktion `operate()`. Jedes `Circuit`-Objekt [3] muß daher von einem Taktgeber gesteuert werden, der in Vielfachen der jeweiligen Taktzyklen die operate-Funktionen aufruft. Dieser Taktgeber ist das `Clock`-Objekt und wird in Abschnitt 7.4.2 beschrieben.

Damit sind die beiden Grundeigenschaften von Schaltkreisen in `Circuit` umgesetzt. Hinzu kommen weitere Eigenschaften, die im folgenden beschrieben werden. Dies beinhaltet die grundlegende Klassenstruktur des Frameworks für Schaltkreise und die Containerklassen zur Ablage ihrer Prozeßdaten. Anschließend werden die Objekthierarchie der Schaltkreise zur Laufzeit und ihre Kommunikation untereinander beschrieben.

[3] Im folgenden ist damit immer ein beliebiges Objekt einer von `Circuit` abgeleiteten Klasse gemeint, also generell ein instantiierter Schaltkreis.

### Das Fabrikmethodemuster

Das Fabrikmethodemuster ist ein klassenbasiertes Erzeugungsmuster, das häufig für den Entwurf von Frameworks verwendet wird. Die Verwendung von Fabrikmethoden verhindert das Einbinden anwendungspezifischen Codes in Frameworkcode.

Das Muster kann eingesetzt werden, wenn eine Klasse den Typ der von ihr zu erzeugenden Objekte nicht im voraus kennen kann und deren Erzeugung deswegen an ihre Unterklassen delegiert. Genau dies ist bei den Schaltkreisen des Simulationsframeworks der Fall.

Jeder Schaltkreis erzeugt pro Taktzyklus einen Satz für ihn spezifischer Ausgangsdaten. Die Basisklasse `Circuit` des Frameworks weiß lediglich *wann* ein neuer Datensatz anfällt, nämlich bei Aufruf der Funktion `operate()`, sie kann aber nicht wissen *welche Art* von Datensatz dies ist. Sie kennt weder die Implementierung von `operate()`, noch die Struktur der Datenbusse eines spezifischen Schaltkreises. Trotzdem muß das Framework in der Lage sein mit diesen Objekten umzugehen, um sie zum Beispiel zu takten.

Diesen Anforderungen wird in dem Simulationsframework mit Hilfe einer Klassenstruktur begegnet, deren Grundstruktur dem Fabrikmethodemuster aus [11] entspricht. Daher wird nun zunächst das Fabrikmethodemuster eingeführt und anschließend auf die spezielle Ausprägung in Zusammenhang mit dem Simulationsframework eingegangen.

Die Struktur des Fabrikmethodemusters (auch als Virtueller Konstruktor bezeichnet) zeigt Abbildung 7.4. Es enthält zwei Vererbungshierarchien, eine für Produkte und eine für Erzeuger.

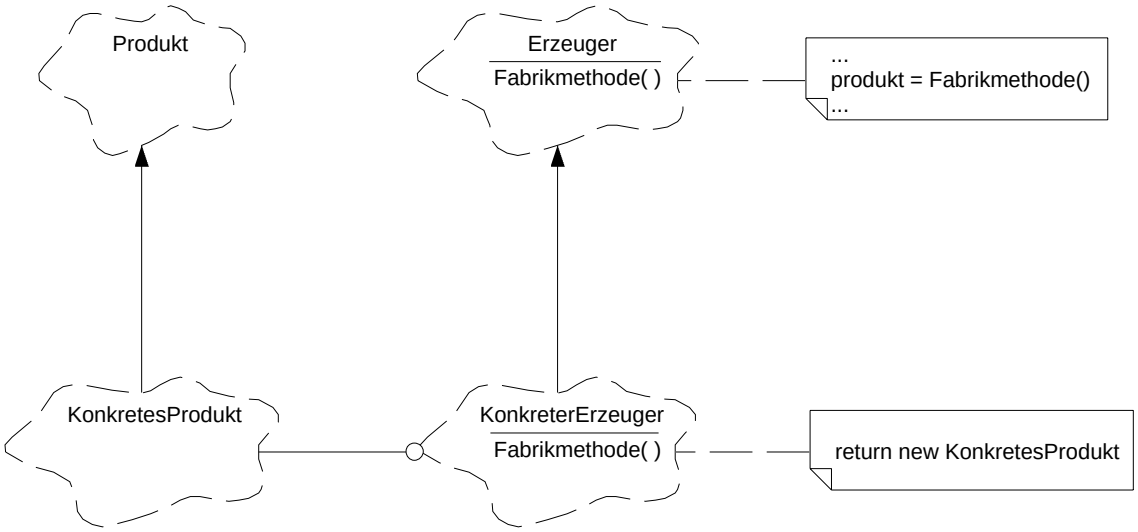


Abbildung 7.4: *Das Fabrikmethodemuster. [11]*

Die Klasse Produkt (auch Dokument genannt) ist die Basisklasse aller von den Fabrikmethoden erzeugten Objekte. Von ihr werden jeweils die konkreten Produktklassen abgeleitet.

Die Basisklasse Erzeuger deklariert die Fabrikmethode, die ein Objekt von dem Typ Produkt zurückgibt. Die Fabrikmethode des Erzeugers kann abstrakt sein oder auch eine Defaultimplementierung enthalten. Alle konkreten Erzeugerklassen müssen von dieser Basisklasse abgeleitet werden und überschreiben dabei die Fabrikmethode, so daß sie ein Objekt vom Typ KonkretesProdukt zurückgibt. Jeder konkreten Erzeugerklasse ist also eine konkrete Produktklasse zugeordnet, die von ihrer Fabrikmethode verwendet wird.

Im Fall des Simulationsframeworks entspricht die Klasse `Circuit` dem Erzeuger. Seine Fabrikmethode `operate()` erzeugt bei jedem Aufruf einen Satz von Ausgangsdaten, der den Zustand des Ausgangsbuses des Schaltkreises zu einer gegebenen Zeit repräsentiert. Hierzu besitzt das Framework eine Vererbungshierarchie mit der Basisklasse `CircuitData`, deren abgeleitete Klassen die Produkte, also die Ausgangsdaten, ihres zugehörigen Schaltkreises speichern können. Die Produkthierarchie des Frameworks zeigt Abbildung 7.5 und die Erzeugerhierarchie Abbildung 7.6.

In der Klassenstuktur des Simulationsframeworks besitzt das Fabrikmethodemuster noch eine Reihe von besonderen Entwurfsentscheidungen:

- Die Erzeugerklasse `Circuit` ist abstrakt und bietet keine Implementierung der Fabrikmethode.
- Die Fabrikmethode `operate()` gibt ihr Resultat nicht zurück, sondern es wird intern in dem jeweiligen Schaltkreisobjekt gespeichert.
- Die Konkreten Erzeugerklassen werden mit ihrer Ausgangsdatenklasse parametrisiert.
- Die konkreten Erzeugerklassen werden nicht direkt von `Circuit` abgeleitet, sondern über eine parametrisierte Zwischenstufe, in der das Ausgangsverhalten des Schaltkreises hinzukommt.

Diese Punkte werden in den folgenden Unterabschnitten angesprochen.

### Die Produkthierarchie

Für das Speichern der Schaltkreis-Ausgangsdaten besitzt das Framework eine Vererbungshierarchie von Containerklassen, in denen die Daten abgelegt werden. Die Hierarchie ist im Prinzip eine Hierarchie von Datenstrukturen, die jeweils einen Schaltkreis Ausgangsdatenbus repräsentieren und entspricht der Produkthierarchie des Fabrikmethodemusters. Wie in Abbildung 7.5 zu sehen, enthält das Framework die Klasse `CircuitData` als Basisklasse. Sie enthält zum Beispiel als Attribut die Zeit, zu der die Daten am Ausgang gültig sind. Alle spezifischen Klassen für

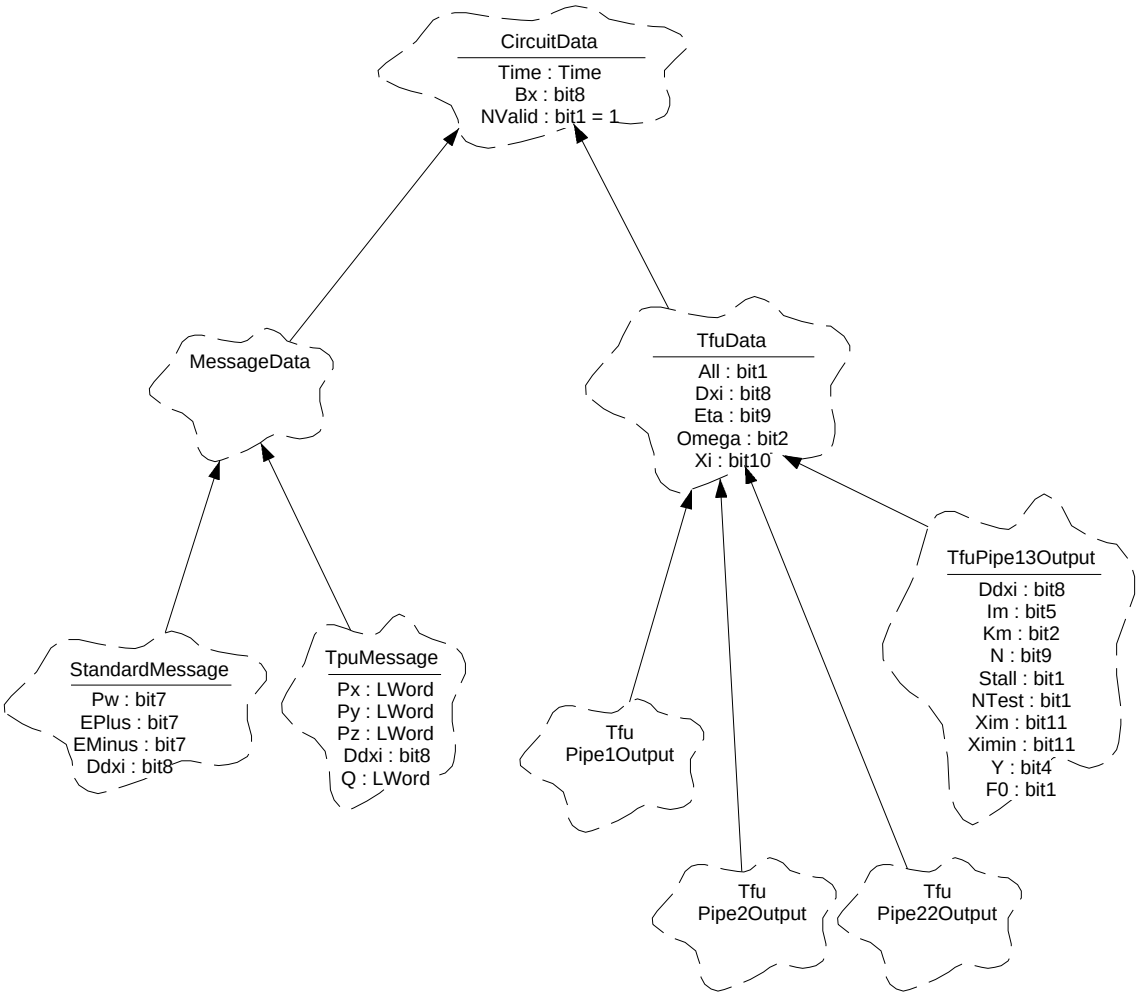


Abbildung 7.5: *Die Vererbungshierarchie der Containerklassen für die Schaltkreis Ausgangsdaten.*

Schaltkreis-Ausgangsdaten müssen von `CircuitData` abgeleitet werden, was, wie bei den TFU-Ausgangsdatenklassen in der Abbildung, auch über mehrere Stufen geschehen kann. Die Attribute von `TfuData` sind in allen Schaltkreis-Ausgängen der TFU vorhanden. Von der als Beispiel aufgeführten Klasse (`TfuPipe13Output`) sind alle Attribute, auch die aus den Basisklassen, in der Abbildung zu sehen. Ein solches Ausgangsdaten-Objekt speichert den, von einem Objekt der Klasse `TfuPipe13` erzeugten, Zustand des Ausgangsbusses der dreizehnten Pipelinestufe der TFU während eines bestimmten Zeitintervalls.

### Die Vererbungshierarchie der Schaltkreis-Klassen

Die Vererbungshierarchie der Schaltkreis-Klassen entspricht der Erzeugerhierarchie des Fabrikmethodemusters, das jedoch um zusätzliche Eigenschaften erweitert wur-

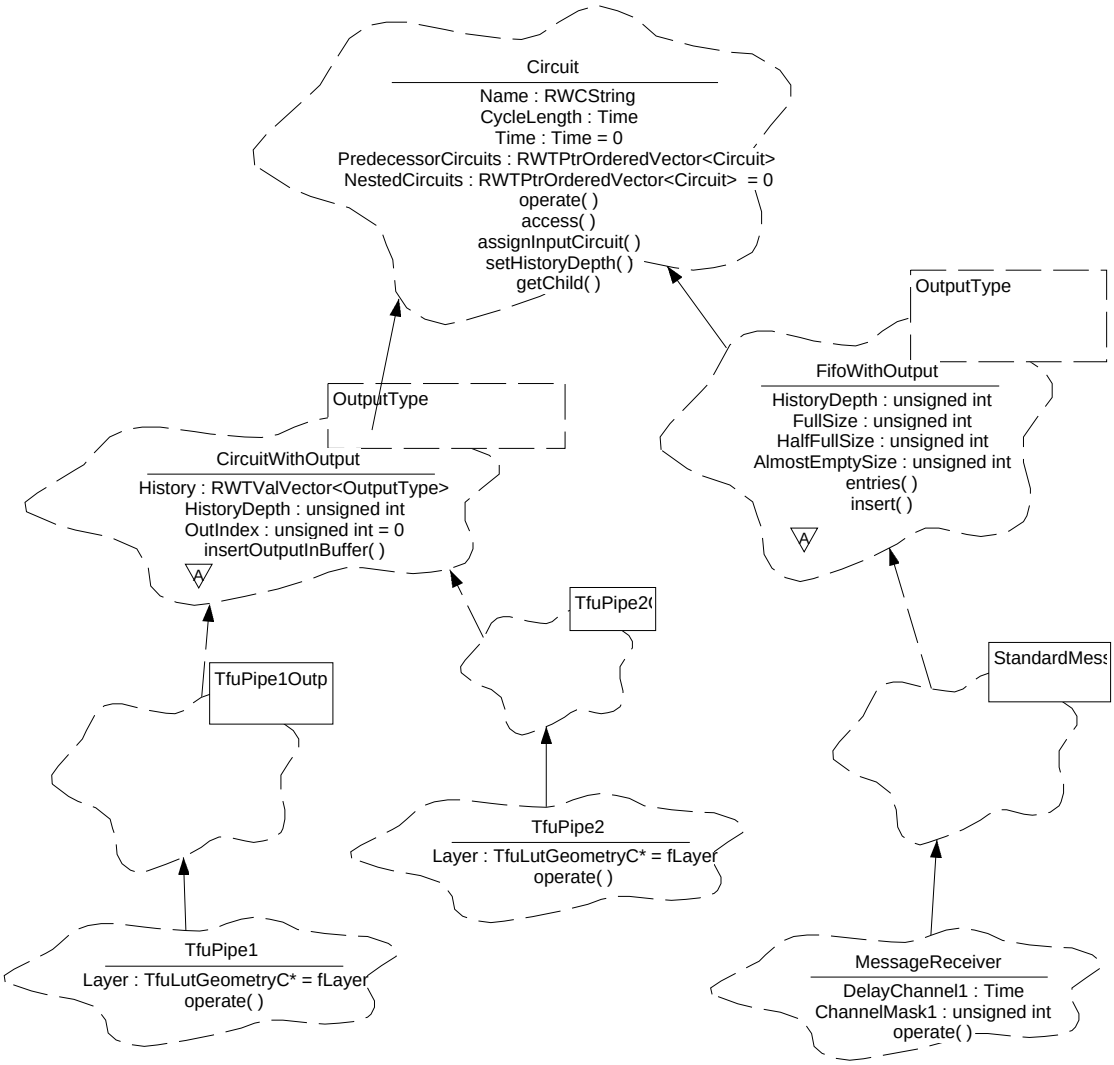


Abbildung 7.6: *Die Vererbungshierarchie der Schaltkreise.*

de, um den Frameworkcode wiederverwendbarer und erweiterbarer zu machen. Um, ausgehend von den Framework Basisklassen `Circuit` und `CircuitData`, einen Schaltkreis zu implementieren wird die Vererbung mit Generizität kombiniert:

Mit Hilfe der Vererbung erhalten die spezialisierten Schaltkreis-Klassen von der abstrakten Basisklasse `Circuit` Funktionalität, die allen gemeinsam ist. Zusätzlich wird durch die Abstammung von einer gemeinsamen Basisklasse Polymorphie zwischen den Schaltkreis-Klassen ermöglicht. Dies wird beispielsweise von dem Taktgeber bei dem Aufruf der `operate`-Funktionen genutzt.

Zum zweiten wird bei der Ableitung eines Schaltkreises die Basisklasse mit der zu ihm gehörenden Containerklasse parametrisiert. Dadurch kann der Typ der Ausgangsdatenklasse in die Funktionen und Attribute, die das Ausgangsverhalten bilden eingebaut werden, was die Wiederverwertung auf die Ausgangsfunktionalität

ausdehnt.

Wenn man die Klasse `Circuit` direkt mit den Ausgangsdatenklassen parametrisiert, gerät man aber in Konflikt mit der erwünschten Polymorphie. Bei dieser Vorgehensweise, in C++ mit Templates realisiert, stellt jede spezifische Schaltkreis-Klasse einen eigenen, unabhängigen Typ dar, weil die Parametrisierung im Prinzip durch Textersetzung von dem Compiler übernommen wird. Damit ist dann kein Polymorphismus zwischen den Schaltkreis-Klassen möglich, weil keine gemeinsame Basisklasse existiert.

Zur Lösung dieses Problems wird die Schaltkreis-Funktionalität auf zwei Klassen in der Vererbungshierarchie aufgeteilt: `Circuit` und davon abgeleitet die parametrisierte Klasse `CircuitWithOutput` (siehe Abbildung 7.7). Durch das Ableiten der parametrisierten Klasse von einer nicht parametrisierten Basisklasse, wird den, von der parametrisierten Klasse abzuleitenden, Schaltkreisklassen eine gemeinsame Implementierung zugrunde gelegt. Somit kann Polymorphismus in Kombination mit Parametrisierung verwendet werden.

Die Klasse `Circuit` enthält dabei die Basisfunktionalität und bei der parametrisierten Klasse `CircuitWithOutput` kommt alles hinzu, was die Ausgangsdaten betrifft. Die Klasse `CircuitWithOutput` ist damit eine abstrakte parametrisierte Klasse, der bei der Instanziierung die jeweilige Ausgangsdatenklasse als Parameter übergeben werden muß.

Im Laufe der Implementierung hat sich gezeigt, daß diese Trennung der Ausgangsfunktionalität von der Schaltkreis-Basisfunktionalität auch notwendig ist, um die erforderliche Flexibilität für das Ausgangsverhalten von Schaltkreisen zu erreichen. Damit können dann Schaltkreise, die eine eigene Gruppe mit einem fundamental verschiedenen Ausgangsverhalten bilden, von einer extra Klasse abgeleitet werden. Dies wurde für die Implementierung von FIFOs genutzt, die, zum Beispiel verglichen mit einer Pipelinestufe, ein grundlegend anderes Ausgangsverhalten besitzen. Sie werden daher nicht von `CircuitWithOutput`, sondern von `FifoWithOutput` (Abbildung 7.7) abgeleitet.

### Die Objekthierarchie der Schaltkreise

Nach dem zugrundeliegenden Modell sollen die Elemente der Simulation zusammengesetzt werden und wiederum als Elemente betrachtet werden können. Dies dient im wesentlichen dazu, komplexe Schaltkreise sinnvoll in Subschaltkreise zu zerlegen, um eine übersichtliche Struktur zu erhalten. Zur Realisierung dieser Möglichkeit im Rahmen des Frameworks, wird für die Implementierung der Klasse `Circuit` das Strukturmuster „Kompositum“ aus [11] verwendet. Das Kompositum fügt Objekte in Baumstrukturen zusammen. Das Muster ermöglicht es Klienten solche Objekte einheitlich zu behandeln, unabhängig davon, ob es sich um ein einzelnes oder ei-

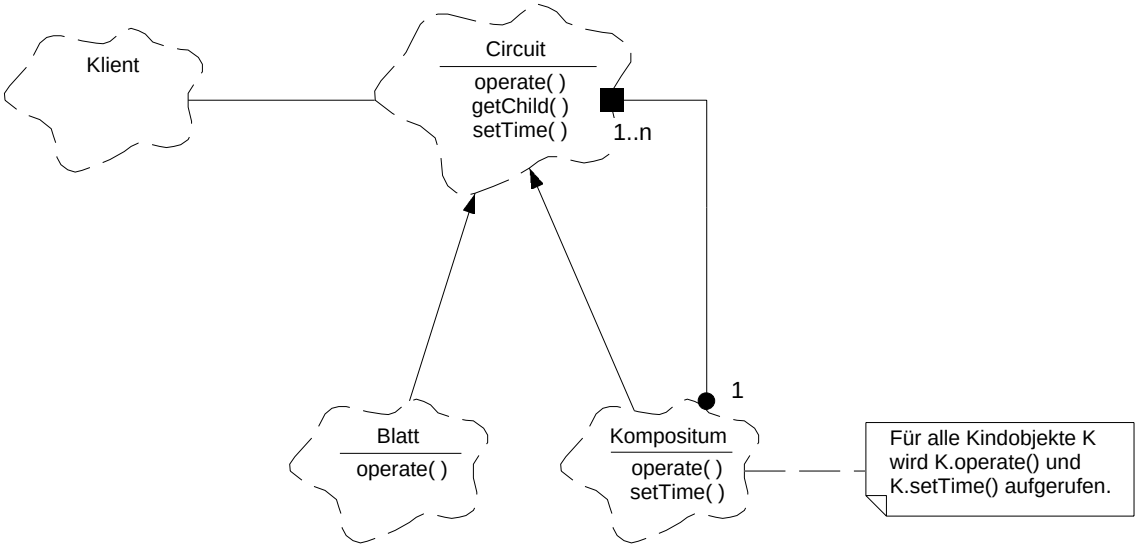


Abbildung 7.7: *Die Schaltkreis-Objekte können nach dem Kompositum Muster in Baumstrukturen organisiert werden.*

ne Komposition von Objekten handelt (also um ein Blatt oder einen Knoten des Baumes).

Das Klassendiagramm des Kompositum zeigt Abbildung 7.7. Dabei sind für die Bezeichnungen teilweise bereits die Namen der Simulation verwendet. Die Basisklasse `Circuit` stellt eine allgemeine Komponente eines Baumes dar. In der Spezialisierung kann eine Komponente sowohl ein Blatt, als auch ein Knoten sein. Wenn ein abgeleiteter konkreter Schaltkreis selbst keine Schaltkreise enthält, so stellt er ein Blatt dar. Enthält er weitere Schaltkreise (by value), so ist er ein Kompositum. Ein Klient greift immer über die Klassenschnittstelle von Komponente (also hier `Circuit`) auf Objekte zu und sieht daher keinen Unterschied zwischen einem Blatt und einem Kompositum. Daher ist das Kompositum für den Aufruf der Komponentenfunktionen seiner enthaltenen Objekte zuständig. Die Klasse `TfuPipeline` ist beispielsweise als Kompositum organisiert und enthält die einzelnen Pipelinestufen-Schaltkreise. In ihrer `operate`-Funktion ist sie daher für den Aufruf der `operate`-Funktion ihrer Schaltkreise zuständig. Dies bedeutet, daß von einem Schaltkreis-Baum nur der Wurzel-Schaltkreis bei dem Taktgeber registriert wird. Zudem werden alle Schaltkreise eines Baumes synchron mit derselben Frequenz getaktet.

Zusammengefaßt besitzt das Muster folgende Eigenschaften:

- Das Kompositionsmuster definiert eine Klassenhierarchie, in der aus primitiven Objekten rekursiv komplexere Objekte zusammengesetzt werden können.
- Es vereinfacht die Klienten, im Fall der Simulation zum Beispiel die `Clock`-Klasse (siehe 7.4.2). Unterschiede zwischen zusammengesetzten und einzelnen Objekten können von Klienten ignoriert werden.

- Klienten verwenden immer die Klassenschnittstelle von `Circuit`, um mit den Objekten der enthaltenen Baum-Hierarchie zu interagieren.
- Neue Kompositum- oder Blattklassen können einfach hinzugefügt werden, da sie automatisch zu den existierenden Strukturen passen.

Die Hierarchie aus Schaltkreis-Objekten, die von dem Framework zur Verfügung gestellt wird, enthält zudem einige spezielle Entwurfsentscheidungen, mit denen das Muster an die Anforderungen der Simulation angepaßt ist.

- Die Kindobjekte einer Baum-Hierarchie enthalten keine expliziten Referenzen auf das Elternobjekt. Schaltkreise haben eine Schnittstelle zu anderen Schaltkreisen und zu dem Taktsignal. Sie müssen nicht „wissen“ in welchem Kompositum von Schaltkreisen sie gegebenenfalls enthalten sind.
- Eine Unterscheidung der Schnittstellen zwischen Blatt- und Kompositum-`Circuits` wurde nicht getroffen.
- Da die Schaltkreise fest verdrahtet sind, gibt es keine Member-Funktionen zum hinzufügen oder entfernen von Komponentenobjekten.
- Aufgrund der festen Objektstruktur, die einmal beim Start des Programms aufgebaut wird, gibt es auch keine Typüberprüfung zur Laufzeit. Dies ist normalerweise empfehlenswert, da der Entwurf sonst zu allgemein werden kann.

### Die Kommunikation zwischen Schaltkreisen

Für die Kommunikation zwischen Schaltkreisen müssen diese zuerst untereinander verbunden werden. Anschließend, während des Ablaufs einer Simulation, müssen die Prozeßresultate abgespeichert und zwischen den verbundenen Schaltkreis-Objekten über eine definierte Schnittstelle ausgetauscht werden.

Zu Beginn des Programms werden die erzeugten Schaltkreis-Objekte, die jeweils untereinander Daten austauschen müssen, miteinander bekannt gemacht. Dies entspricht dem Herstellen einer elektrischen Verbindung zwischen den entsprechenden Baugruppen in der Hardware. Zu diesem Zweck besitzt jeder Schaltkreis eine interne Liste, wo die Referenzen zu den `Circuit`-Objekten, von denen er seine Eingangsdaten empfängt, eingetragen werden. Über diese Liste greift ein `Circuit`-Objekt auf seine Vorgänger Schaltkreise zu, die ihm dann seine Eingangs-Daten liefern.

Mit jedem Aufruf seiner operate-Funktion muß ein Schaltkreis an seinem Eingang Ausgangsdaten von anderen Schaltkreisen übernehmen (Eingangsdatenbus), führt damit Rechenoperationen durch und die jeweiligen Resultate werden mit einer spezifischen Zeitverzögerung an seinem Ausgang gültig (Ausgangsdatenbus). Die Ergebnisse eines Taktzyklus, also seine Ausgangsdaten, speichert jedes `Circuit`-Objekt mit einem Zeitindex versehen intern ab. Der Zeitindex gibt die Zeit an, zu der die

Daten an seinem Ausgang gültig sind. Die Ausgangsdaten werden intern in einem Puffer-Speicher in schaltkreisspezifischen Objekten abgelegt. Der Puffer-Speicher und seine Verwaltung ist in dem Schaltkreis gekapselt, die Tiefe dieses internen Speichers kann aber variiert werden, um die Resultate eines Schaltkreises über längere Zeit aufzeichnen zu können.

Das Sequenzdiagramm in Abbildung 7.8 zeigt wie die Schaltkreise den Zustand ihres Eingangsdatenbuses bei ihren registrierten Schaltkreisen erfragen. Dazu wird dem sendenden Schaltkreis (`TfuPipe1`) die Zeit des Zugriffs mitgeteilt und er gibt dann die zu diesem Zeitpunkt an seinem Ausgangsbus gültigen Daten aus seinem internen Speicher zurück. Der EMpfänger (`TfuPipe2`) bearbeitet die Daten und legt sie in einem seiner Ausgangsdatenobjekte ab.

Insgesamt ist die Verschaltung so ausgelegt, daß die Schnittstellen möglichst minimal sind und alle Informationen möglichst lokal verwaltet werden. Die Verwaltung der Eingangsschaltkreise und der Ausgangsdaten sind vollständig unter der lokalen Kontrolle des jeweiligen Schaltkreises. Die zugehörige Funktionalität ist bereits in der Basisklasse `Circuit` enthalten. Der Zugriff auf einen Schaltkreis erfolgt lediglich über eine Zeitangabe. Was damit gemacht wird ist vollständig dem angesprochenen Schaltkreis überlassen. Auch diese Funktion ist bereits in der Basisklasse enthalten, kann aber in speziellen Fällen auch überschrieben werden.

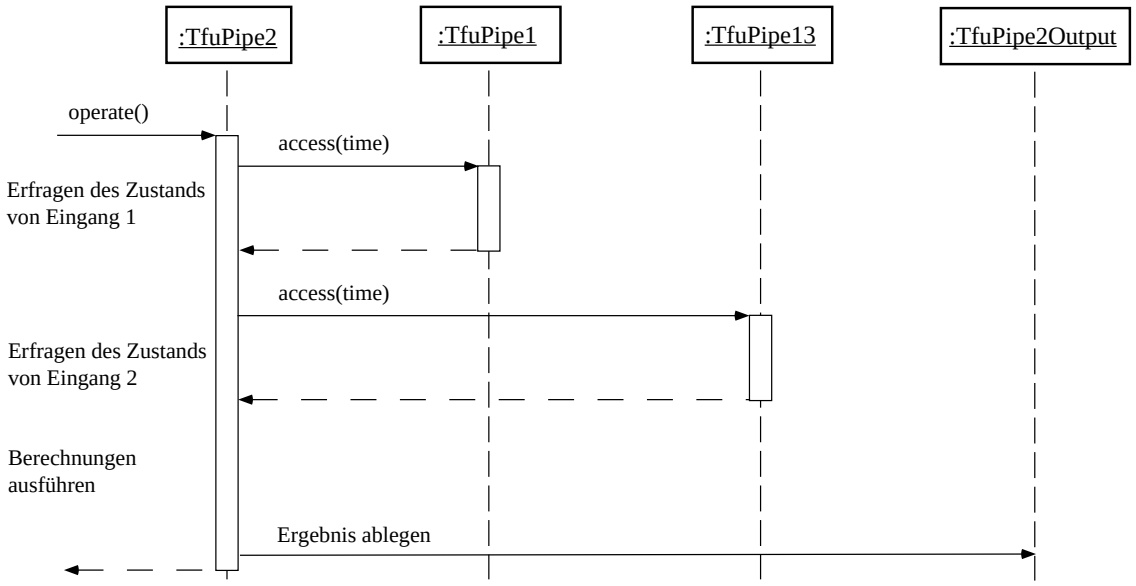


Abbildung 7.8: *Sequenzdiagramm der Ein-/Ausgabe Operationen, die ein Schaltkreis der TFU in seiner Funktion* `operate()` *ausführt. In dem Beispiel erhält der Schaltkreis* `TfuPipe2`*, der die zweite Pipelinestufe der TFU simuliert, seine Eingangsdaten von der ersten Stufe* `TfuPipe1` *und der dreizehnten Stufe* `TfuPipe13`*. Nachdem er seine Prozeßroutine abgearbeitet hat legt er an deren Ende die Ausgangsdaten, mit einem Zeitindex versehen, in einem Objekt vom Typ* `TfuPipe2Output ab`.

### 7.4.2 Zeitgeber und Steuerung der Simulation

Die Verwaltung der Simulationszeit und die Steuerung während eines Simulationslaufs wird von einem zentralen Objekt der Klasse `Clock` übernommen. Von dieser Klasse wird nur ein einziges Exemplar benötigt, das zwei Zuständigkeitsbereiche besitzt:

1. Die Steuerung aller Schaltkreise.
   Hierzu sind alle Schaltkreise bei dem `Clock`-Objekt in Listen registriert. Es nimmt seine Funktion als zentraler Taktgeber für alle `Circuit`-Objekte wahr, indem es die Prozeßroutinen der Schaltkreise (`operate()` ) aufruft.

   Weiterhin bestehen zentrale Zugriffsmöglichkeiten auf alle Schaltkreise, um beispielsweise deren lokale Zeit oder die Größe ihrer internen Ausgangsdatenpuffer zu setzen.

2. Die Simulationszeit.
   Das `Clock`-Objekt verwaltet die zentrale Simulationszeit. Es enthält zudem die Zeitführungsroutine, die bei dem Start einer Simulation aufgerufen wird.

Dies sind zentrale Funktionen, die für alle simulierten Subsysteme einheitlich gehandhabt werden müssen und daher zu den zentralen Bestandteilen des Frameworks gehören. Diese zentralen Funktionalitäten sollen dabei möglichst automatisch zur Verfügung stehen. Dies verringert den Gesamtaufwand, da sie bei der Entwicklung von Simulationen nicht erneut beachtet werden mussen und verhindert zudem weitgehend die Entwicklung von, womöglich inkompatiblen, Speziallösungen einzelner Entwickler. Trotzdem müssen die zentralen Einheiten flexibel gegenüber Änderungen sein, die bei der Entwicklung eines Frameworks auf jeden Fall zu erwarten sind. Die Änderungen sollten sich aus diesem Grunde möglichst nicht auf die Schnittstellen auswirken, da dies unter Umständen Änderungen in allen Objekten, die mit den zentralen Einheiten kommunizieren zur Folge hat, was einen enormen Aufwand bedeuten kann.

Für die Implementierung der Klasse `Clock` wird das Erzeugungsmuster „Singleton“ aus [11] verwendet. Das Singletonmuster stellt sicher, daß es genau ein automatisch erzeugtes Exemplar dieser Klasse gibt und es stellt gleichzeitig einen globalen Zugriffspunkt darauf zur Verfügung:

- Das Singleton definiert eine Exemplarmethode, die es Klienten ermöglicht global auf das einzige Exemplar zuzugreifen. In C++ ist dies eine statische Member-Funktion. Gegenüber globalen Variablen ist dies ein Fortschritt, da es eine Überladung des Namensraums mit denselben vermeidet.

- Das Singleton ist für die Erzeugung seines eigenen Exemplars zuständig. Dadurch besitzt es eine vollständige Kontrolle darüber, wie Klienten auf das Exemplar zugreifen können.

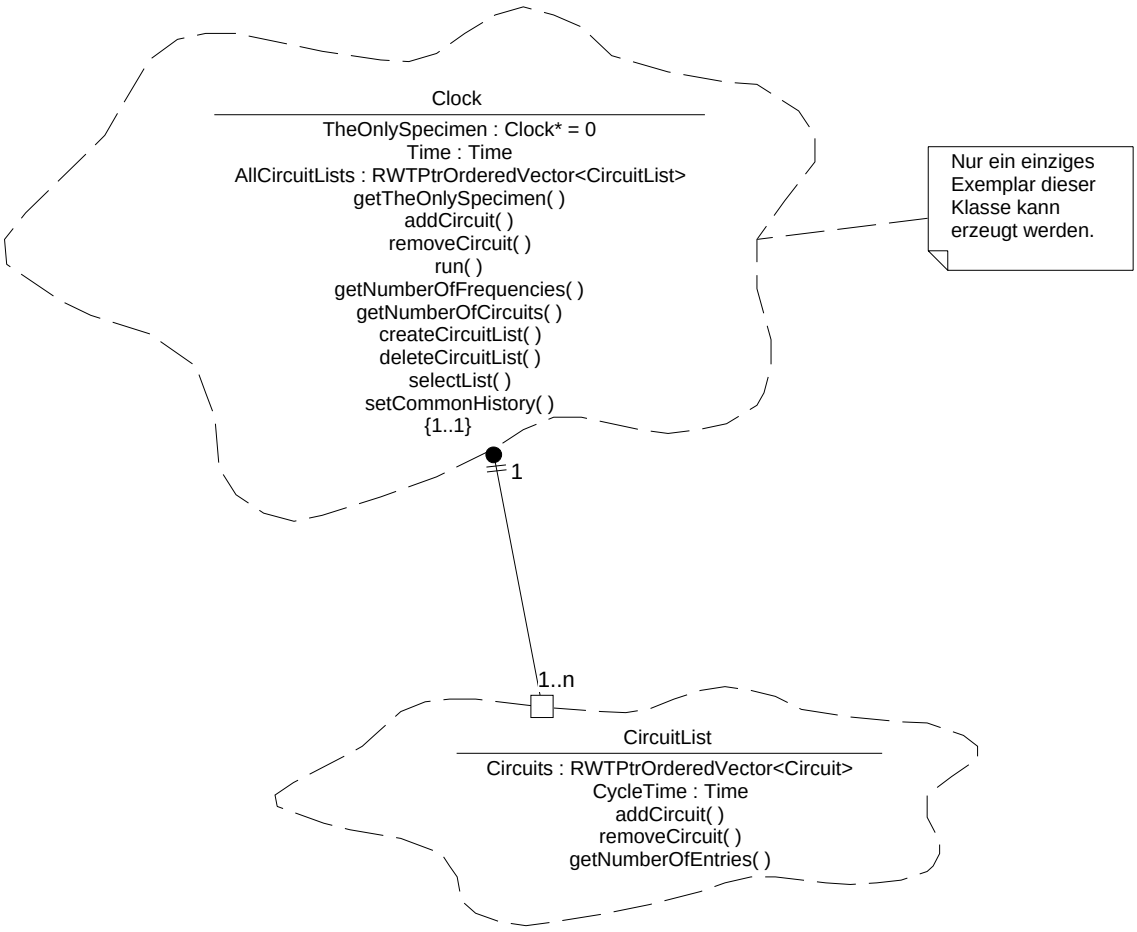


Abbildung 7.9: *Die Klasse* `Clock` *ist als Singleton implementiert. Sie verwaltet die zu steuernden Schaltkreise in Objekten der Klasse* `CircuitList`*. Zudem enthält sie die Zeitführungsroutine der Simulation.*

In Punkto Flexibilität gegenüber zukünftigen Änderungen besitzt das Singleton die folgenden Vorteile:

- Die Singleton Klasse kann abgeleitet und spezialisiert werden. Das erweiterte Exemplar ist für Klienten verwendbar ohne daß dazu ihr Code modifiziert werden muß. Eine Anwendung könnte sogar zur Laufzeit entscheiden, welches spezielle Exemplar der Clock verwendet werden soll.

- Es ist nachträglich möglich die Anzahl der Exemplare zu erhöhen, falls später in der Entwicklung festgestellt wird, daß mehr als ein Exemplar benötigt wird.

Mit diesem Entwurf ist sichergestellt, daß jeder Benutzer des Frameworks an beliebiger Stelle im Programm Zugriff auf das `Clock`-Objekt hat, und zum Beispiel einen Schaltkreis anmelden kann. Um die Erzeugung der Clock muß er sich nicht kümmern, sie steht automatisch zur Verfügung.

Gleichzeitig wird der Benutzer aber auf die Clock Architektur festgelegt. Es ist nicht möglich ein zweites `Clock`-Objekt zu erzeugen und zumindest erschwert andere ausgefallene Wege zu beschreiten, die zu Inkompatibilitäten oder Fehlern zwischen den verschiedenen Simulationen der Trigger-Subsysteme führen könnten. Trotzdem sind aber, im Rahmen des Singleton Musters, nachträgliche Änderungen möglich.

Abbbildung 7.9 zeigt das Klassendiagramm der `Clock`-Klasse zusammen mit der Listen-Klasse, mit deren Instanzen Schaltkreise registriert werden.

### Registrierung und Verwaltung der Schaltkreise bei der Simulationsuhr

Objekte der von der Basisklasse `Circuit` abgeleiteten Klassen, werden bei ihrer Erzeugung automatisch bei der Clock registriert. Bei der Implementierung spezieller Schaltkreise muß man sich darum nicht kümmern. Die Registrierung wird in dem Konstruktor der Basisklasse `Circuit` vorgenommen. Das Sequenzdiagramm der Registrierung zeigt Abbildung 7.10. Ein `Circuit`-Objekt schickt bei seiner Erzeugung eine Referenz auf sich selbst an das `Clock`-Objekt, das es über den globalen Zugriff des Singletons referenziert. Dieses wiederum erfrägt daraufhin die Taktzykluslänge bei dem Schaltkreis, um zu erfahren, wann er getaktet werden muß und ihn anschließend in eine entsprechenden Liste einzutragen. Für jede registrierte Frequenz erzeugt sich die `Clock` ein `CircuitList`-Objekt, in dem die Verweise auf die Schaltkreise dieser Frequenz abgelegt werden (Abbildung 7.9).

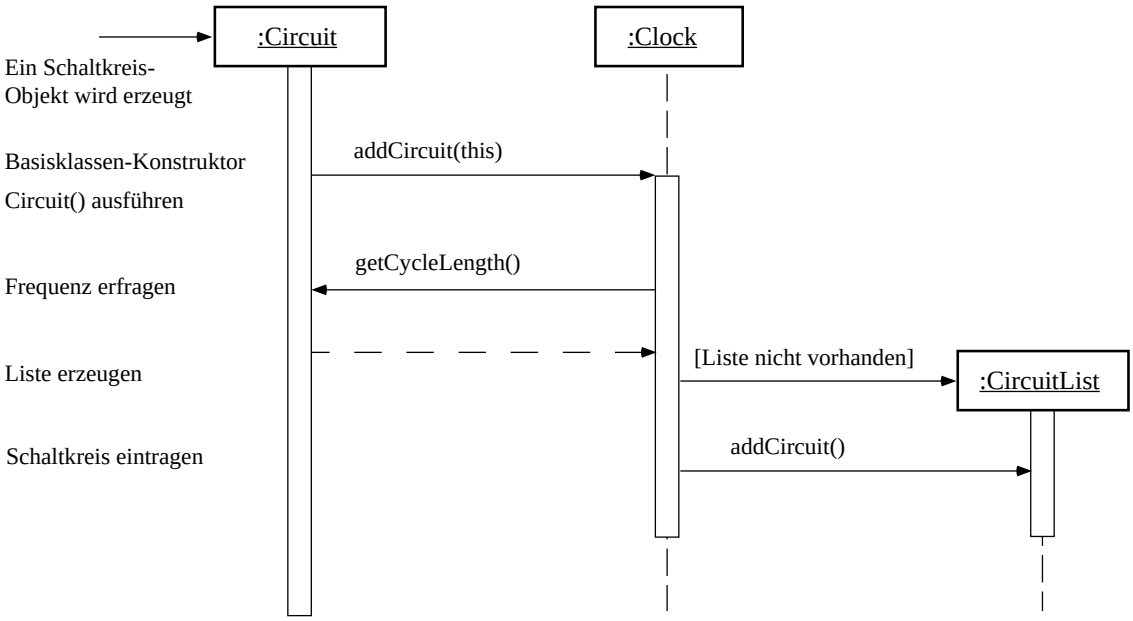


Abbildung 7.10: *Sequenzdiagramm der Registrierung eines Schaltkreises bei dem* `Clock`*-Objekt. Das* `Clock`*-Objekt trägt die Schaltkreise nach Frequenzen sortiert in Listen ein. Da für die Frequenz dieses Schaltkreises noch keine interne Liste vorhanden war, muß sie von der Clock erst erzeugt werden, bevor der Schaltkreis eingetragen werden kann.*

Die Belange der Zeitführung und Simulationssteuerung sind vollständig in dem `Clock`-Objekt gekapselt. Daher muß sich ein Entwickler, der das Framework benutzt, nicht darum kümmern wie seine Schaltkreise mit der Steuerung zusammenarbeiten - er muß nicht einmal verstehen wie die Steuerung überhaupt funktioniert. Zum anderen stellt dies eine gemeinsame Schnittstelle aller Subsysteme zu der Simulationssteuerung dar. Vorausgesetzt die Schnittstellen der Subsysteme untereinander funktionieren, so ist dann sichergestellt, daß die verteilt entwickelte Software in einem gemeinsamen Simulationsprogramm arbeiten kann.

Die Schnittstelle des Frameworkbenutzers zu der Clock ist damit minimal. Die Erzeugung der Clock erfolgt automatisch und man kann global auf sie zugreifen. Bei der Erzeugung eines Schaltkreises muß lediglich die Länge seines Taktyklus angegeben werden, seine Registrierung bei dem `Clock`-Objekt erfolgt dann ebenfalls automatisch unter der entsprechenden Frequenz. Der einzige direkte Zugriff erfolgt bei dem Start der Simulation mit der Simulationsdauer als Parameter: `Clock.run(time);`. Insgesamt reduziert sich der Entwicklungs- und Einarbeitungsaufwand erheblich und es verringert sich die Zahl möglicher Fehler.

#### Zeitführung

Die Clock startet bei einer bestimmten Simulationszeit (in der Regel bei Null). Von diesem Startwert aus werden alle registrierten Schaltkreise mit der ihnen eigenen Frequenz getaktet. Die `Clock` ist also eine Art Multifrequenz-Taktgeber, der sämtliche Clocksignale synchron verteilt.

Alle Frequenzen beginnen bei dem vorgegebenen Start-Zeitpunkt (vergleiche Abbildung 7.11). Von da an erhöht der Zeitgeber die Simulationszeit zu dem jeweils nächsten Zeitpunkt eines ansteigenden Taktsignals. Die `Circuit`-Liste, die der entsprechenden Frequenz zugeordnet ist, wird dann abgearbeitet und von jedem `Circuit`-Objekt einmal die `operate()` Funktion aufgerufen. Jeder Schaltkreis wird also bei Vielfachen der Dauer seines Taktzyklus getaktet, die Reihenfolge, in der die Schaltkreise einer Liste getaktet werden, spielt dabei keine Rolle.

### 7.4.3 Das Board als Aggregat von Objekten

Eine weitere Abstraktion, die das Framework bereitstellt, ist die Basisklasse `Board` (siehe Abbildung 7.12). Die Board-Klassen sind analog zu ihren realen Pendants zu sehen. Ein Board enthält eine Ansammlung von Schaltkreisen, die statisch untereinander verbunden sind und das Verhalten eines Elektronikboards simulieren. Ein Board kann Schaltkreise unterschiedlicher Frequenz enthalten, ist selbst aber kein Schaltkreis. Während der Simulation ist das Board passiv, das heißt es wird selbst nicht getaktet, sondern nur die `Circuit`-Objekte, die es enthält. Zusätzlich wird alles was sich spezifisch auf ein Board bezieht, zum Beispiel Datenstrukturen zu seiner Initialisierung, in der Board-Klasse untergebracht.

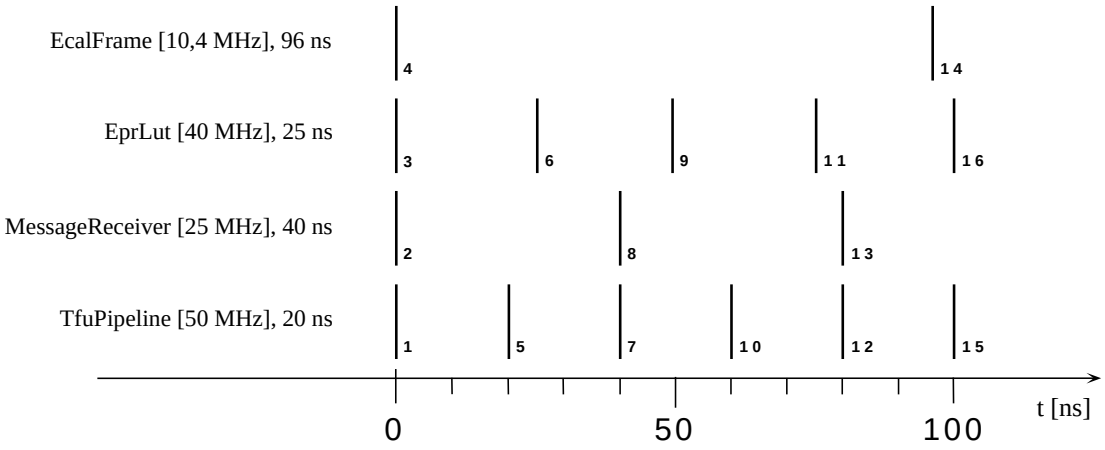


Abbildung 7.11: *Die senkrechten Striche markieren die Zeitpunkte, an denen* `Circuit`*-Objekte von dem* `Clock`*-Objekt getaktet werden. Die Numerierung gibt die Reihenfolge an und es wird erkennbar, wie die Simulationszeit hochgezählt wird, indem sie zu dem jeweils nächsten Zeitpunkt springt, an dem eine Taktfrequenz ein ansteigendes Clock Signal erhält.*

Die Hauptaufgabe der Board-Klassen besteht in der Kapselung (vergleiche Abschnitt 7.3.1) der Implementierungs-Details der jeweiligen spezifischen Boardsimulation. Durch die Bereitstellung einer einzelnen, einfachen Schnittstelle für das gesamte Board minimiert sich die Kommunikation und die Abhängigkeiten zwischen den Subsystemen. Nach außen besitzen die `Board`-Objekte die gleiche Schnittstelle wie die Hardware. Dies ist sinnvoll, da Schnittstellen elektronischer Boards in der Regel sorgfältig und minimal gewählt werden. Durch die Kapselung und die Verwendung der gleichen Schnittstelle wie in der Hardare, können die Board-Objekte in äquivalenter Weise zu den realen Boards gehandhabt werden.

Für den Entwurf der Board-Klassen wurde das Fassadenmuster aus [11] gewählt. Das Fassadenmuster ist ebenfalls ein Strukturmuster. Es bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.

Im Fall der `Board`-Klassen bedeutet dies, daß alle Anfragen eines Klienten, auch wenn sie interne Objekte eines Boards betreffen, über die Schnittstelle des `Board`-Objekts gestellt werden. Es bietet damit eine einfache Schnittstelle zu einem komplexen Subsystem. Dies entkoppelt den Klienten von der Struktur des Subsystems, über die er keine Information besitzt.

Ein typischer Fall für ein Board vom Typ `TfuBoard` ist ein Klient, der es mit einem weiteren `TfuBoard`-Objekt verbinden will, das die Ausgangsdaten des ersteren über das Messagesystem empfängt. In diesem Fall muß der Schaltkreis von Board A, der das Ausgangs-FIFO simuliert, mit dem Schaltkreis von Board B, der die Messages empfängt, verbunden werden. Dies geschieht, wie oben beschrieben, indem der Empfänger eine Referenz auf den Sender Schaltkreis zugewiesen bekommt.

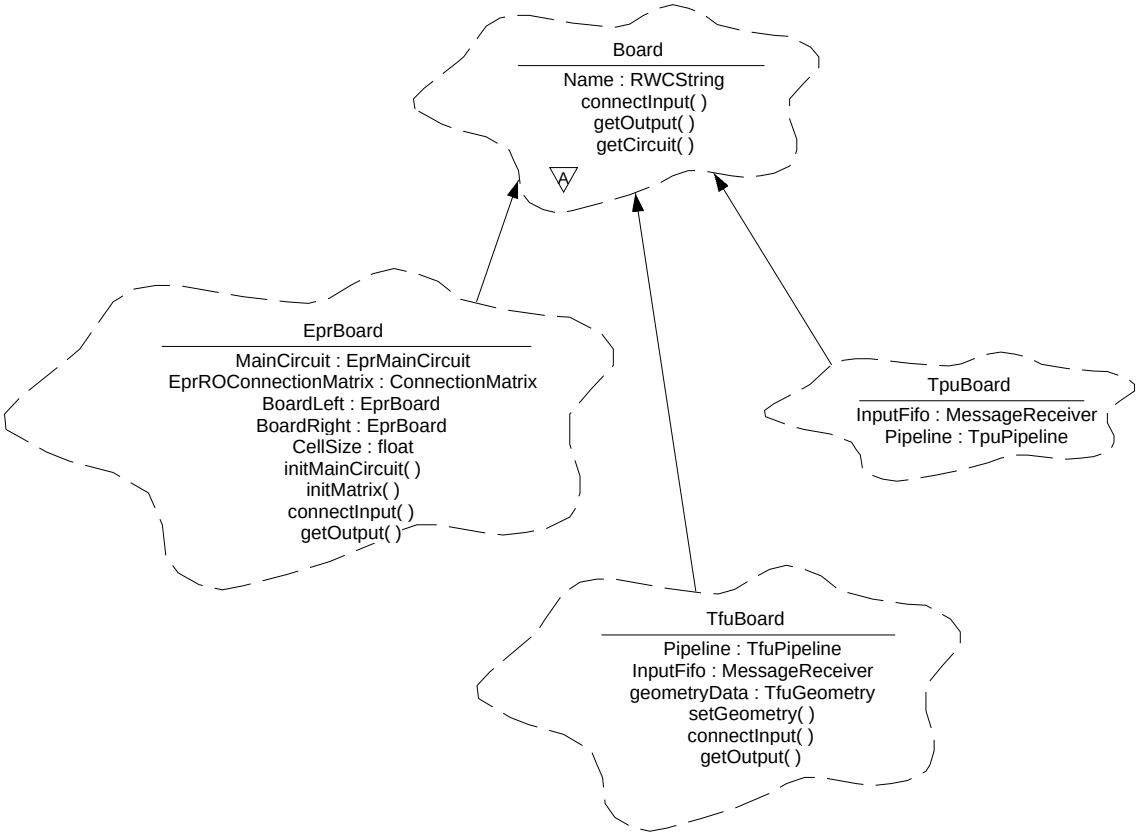


Abbildung 7.12: *Die Vererbungshierarchie der Board-Klassen. Von der abstrakten Basisklasse Board müssen alle anderen Board-Klassen abgeleitet werden, hier die TFU. die TPU und das Elektron-Pretrigger Board.*

Bei einer einfachen Implementierung würde der Klient die Zuweisungsfunktion des Empfänger-Schaltkreises aufrufen und ihm die Referenz des Senders übergeben. Um auf diese Weise zwei Boards zu verbinden, muß der Klient Informationen über die interne Struktur des `TfuBoard`-Objekts besitzen, da er direkt auf dessen interne Schaltkreise zugreift (Abbildung 7.13 linker Teil). Dadurch wird die Implementierung des Klienten abhängig von der Implementierung des Boards. In diesem Fall müßte er beispielsweise wissen, ob die vier Eingangs-FIFOs einer TFU als einzelne Schaltkreise oder als ein gemeinsamer Schaltkreis realisiert sind. Bei einer diesbezüglichen Änderung der `TfuBoard`-Klasse würde dies auch eine Änderung des Klienten nach sich ziehen.

Mit Hilfe des Fassadenmusters werden diese Probleme vermieden. Im einzelnen bietet es folgende Vorteile:

- Es vereinfacht die Benutzung des Subsystems, indem Klienten von den Subsystemkomponenten abgeschirmt werden.

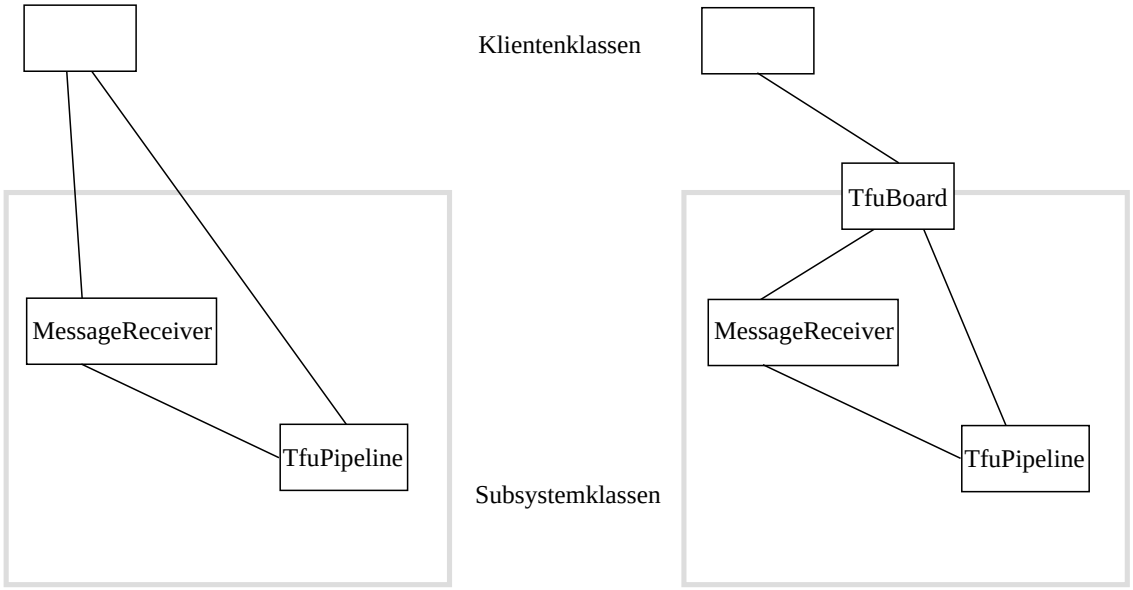


Abbildung 7.13: *Beispiel für Zugriff mit und ohne Fassadenmuster. Links greift ein Klient direkt auf Objekte des Board Subsystems zu. Recht erfolgen alle Zugriffe über die Schnittstelle von* `TfuBoard` *die als Fassade für die Interna des Subsystem dient.*

- Es erhöht die Flexibilität gegenüber nachträglichen Änderungen, weil die Implementierung von Klient und Subsystem entkoppelt ist.
- Es erlaubt trotzdem Klienten, daß sie hinter die Fassade blicken, wenn sie dies müssen.

und

- Klienten schicken ihre Anfragen an das Board, das sie dann an das zuständige Subsystemobjekt weiterleitet.
- Ein Board Objekt kann einfach gehandhabt werden, ohne das der Entwickler über die Interna Bescheid wissen muß.

Die Verbindung zweier `Board`-Objekte erfolgt dann über die Schnittstelle der beteiligten Boards, obwohl die Boards letztendlich auf Schaltkreis-Niveau verbunden werden. Anschließend schaut der Empfänger Circuit hinter die Fassade des sendenden Boards, das heißt, er greift direkt auf seinen Eingangs-Schaltkreis auf dem anderen Board zu. Die Clock schaut ebenfalls hinter die Fassade des Boards, da die enthaltenen Schaltkreise direkt getaktet werden.

Bei der Entwicklung spezifischer Boards müssen diese von der Basisklasse `Board` abgeleitet werden (Abbildung 7.12). Ein `Board`-Objekt erzeugt und initialisiert bei seiner Erzeugung alle `Circuit`-Objekte und sonstigen Objekte, die es enthält. Außerdem stellt es alle Verbindungen zwischen seinen `Circuit`-Objekten her. Die Details

werden durch die Fassade des Boards so gekapselt, daß ein anderer Entwickler ohne Kenntnisse des inneren Aufbaus `Board`-Objekte für eigene Programme einsetzen kann.

## 7.5 Zusammenfassung

Im Rahmen des First Level Trigger Projekts werden in mehreren Bereichen Simulationen benötigt, für Tests der Elektronikboards, Inbetriebnahme des Systems und Simulationen des Gesamtsystems zur Beantwortung physikalischer Fragestellungen. Dies beinhaltet auch die Simualtion der an anderen Instituten entwickelten Pretriggersysteme. In dem vorliegenden Konzept baut die Gesamtsimulation auf den zum Test der Hardware verwendeten (und dadurch verifizierten) Einzelboardsimulationen auf.

Das Triggersystem ist ein offenes dynamisches kybernetisches System. Als zeitdiskretes Simulationsmodell für das System wird ein prozeßorientiertes Simulationsmodell verwendet, das durch die, in der objektorientierten Sprache C++, entwickelte Software umgesetzt wird.

Um den Schwierigkeiten, die sich aus dem Umfang der Systeme, der langen Lebensdauer der Software und der auf mehrere Institute verteilten Entwicklung ergeben, zu begegnen, werden Teile der objektorientierten Design Methode von Booch und eine Reihe von Entwicklungswerkzeugen eingesetzt. Im Einzelnen sind dies das Programm Rose zur Erzeugung eines Design Modells und für C++ Code-Generierung, die Klassenbibliothek Tools.h++ und DOC++ für die automatische Generierung einer Dokumentation aus dem Quellcode.

Die Basisabstraktionen des Modells werden in einem Framework umgesetzt, in dessen Aufbau mehrere Entwurfsmuster enthalten sind, um die erforderliche Flexibilität zu erreichen. Unter Verwendung des Frameworks werden die Simulationsprogramme für spezifische Hardware erstellt. Es enthält hierzu drei Klassenhierarchien, für Schaltkreise, die Containerklassen ihrer Ausgangsdaten und Boards, mit denen durch ableiten ein zu simulierendes System aufgebaut werden kann. Alle Aspekte der Steuerung der Simulation sind bereits in dem Framework enthalten. Da das Framework die Architektur der einzelnen Simulationen festlegt, können sie anschließend zu einer Simulation des Gesamtsystems zusammengefügt werden.

# Kapitel 8

# Status und Ausblick

Die vorliegende Arbeit befaßt sich mit der Software für den First Level Trigger des HERA-B Experiments am DESY. Softwareentwicklungen werden für den Betrieb, die Simulation und den Test des massiv parallelen Multiprozessorsystems benötigt.

Die Software kommt auf verteilten, heterogenen Rechnersystemen zum Einsatz und wird zudem über Jahre hinweg von wechselnden Personen an zum Teil unterschiedlichen Orten weiterentwickelt und eingesetzt. Dies kann nur gelingen, wenn die im Laufe der Zeit geleistete Entwicklungsarbeit aufeinander aufbaut. Aufgrund dieser Rahmenbedingungen spielen bei allen Entwicklungen auch der Einsatz moderner Methoden und Werkzeuge, die die Entwicklungsarbeit unterstützen, eine wichtige Rolle.

In einem ersten Schritt wurde ein Gesamtkonzept [58] für die Software entworfen. Die Modularisierung in diesem Konzept hat sich insgesamt bewährt. Teile dieses Konzepts, die VME-Systemsoftware, die grafische Benutzeroberfläche und die Netzwerkkommunikation, wurden praktisch unverändert realisiert. Andere, wie die Simulation, sind gegenüber den ersten Entwürfen in sehr veränderter Form implementiert worden. Bei der Simulation haben sich die Anforderungen verändert, da sie zu einem Gemeinschaftsprojekt mit anderen Gruppen (Pretrigger) wurde.

Im Rahmen dieser Arbeit wurden die meisten Teile der Systemsoftware realisiert, die VME-Systemsoftware, ein Kontrollprogramm mit grafischer Oberfläche, die Netzwerkkommunikation und ein Simulationsframework.

Das Ziel der VME-Systemsoftware ist, die Programmierung der Mikroprozessorrechner auf den FLT-Boards in einer Hochsprache zu ermöglichen. Gleichzeitig müssen die wichtigsten Betriebssystemfunktionen, wie das Ausführen von Programmen, Datei-Handling und die Speicherverwaltung zur Verfügung gestellt werden. Sowohl Entwicklung als auch Ablauf der Benutzerprogramme sollen möglichst transparent gegenüber der speziellen Hardwareumgebung erfolgen.

Erreicht wird dies durch die Implementierung einer an die spezielle Hardware angepaßten Version der C-Standardbibliothek zusammen mit einem zugeordneten Service Prozeß (`trun`) auf dem Unix-Host. Dabei wird ein Teil der Betriebssystemschnittstelle des Unix-Hosts auf das Board exportiert. Die VME-Bus Kommunikation ist für den Entwickler und Benutzer von Board-Programmen vollständig in der VME-Systemsoftware gekapselt. Für die Entwicklung dieser Software stand das Testboard als Hardwarebasis zur Verfügung.

Der erste Einsatz der VME-Systemsoftware erfolgte bereits bei den Testläufen von HERA-B Ende 1996 und Ende 1997 am DESY in Hamburg. Dabei wurde das Testboard als einfaches Triggersystem eingesetzt, das Messages von dem Elektron-Pretrigger Prototyp empfing. Das auf dem Testboard arbeitende C-Programm fällte anhand der empfangenen Message Daten eine Triggerentscheidung und gab sie an das Fast Control System weiter.

Mittlerweile sind als weitere lauffähige C-Applikationen große Teile der C-Schnittstelle zur Hardware von TFU und TPU vorhanden. Zudem wurde als C-Applikationen auch Testsoftware für die beiden Boardtypen entwickelt.

Insgesamt hat sich die VME-Systemsoftware bei den Entwicklungen sehr gut bewährt. Sie ist bisher von 5 Personen ohne Kenntnisse der zugrundeliegenden VME-Bus Kommunikation zum Entwickeln von C-Programmen eingesetzt worden. Dabei waren keine nachträglichen Änderungen in der Struktur der Software notwendig. Die Benutzer konnten mit geringem Einarbeitungsaufwand ihre Programme entwicklen und ablaufen lassen, die vorhandenen ausführlichen Benutzer- und Entwickler Dokumentationen haben sich dabei gut bewährt.

Damit ist das Ziel, daß die Boards mit wenig Spezialwissen programmiert werden können voll erreicht worden. Es ist auch bereits erkennbar, daß sich der Aufwand, der in diese „transparente“ Lösung investiert wurde, wieder auszahlt. Die Entscheidung, für die Boards keine spezielle VME-Bus Kommunikationsbibliothek zu schreiben und auch kein vollständiges Betriebssystem auf den Boards zu installieren, sondern eine Host gestützte Laufzeitumgebung, hat sich somit als richtig erwiesen. Zudem arbeitet das System sehr effizient, da es vollständig asynchron arbeitet, indem Kommunikation immer über Interrupts ausgelöst wird. Die VME-Software ist damit vollständig und erfolgreich implementiert worden.

Die bisher größte Änderung der VME-Systemsoftware erforderte ihre Anpassung an die TFU/TPU-Hardware, die im Oktober 1997 mit Hilfe des ersten TFU-Prototyps erfolgte. Wichtigster noch fehlender Teil zur Programmierung der Boards ist im Moment eine Sammlung von Mathematikfunktionen, die insbesondere zur Berechnung der Lookup-Tabellen benötigt werden.

Im Betrieb des FLT müssen die Board-Prozesse mit zentralen Einheiten, die der Steuerung und Überwachung des gesamten Systems dienen, über Netzwerk kommunizieren. Das zentrale Kontroll-Programm hat zudem die Aufgabe, eine möglichst

einfache Benutzerschnittstelle zu dem Triggersystem zur Verfügung zu stellen. Es benötigt daher eine grafische Benutzeroberfläche, die die Komplexität des Systems anschaulich darstellt, und die Fähigkeit zur Netzwerkkommunikation.

Hierfür wurde ein Programm entwickelt, das als zentraler Kontrollprozeß für den FLT verwendet werden kann. Seine grafische Benutzeroberfläche wurde in Tcl/Tk implementiert und für die Netzwerkkommunikation finden die Tcl-Sockets Verwendung. Das Programm verwaltet die Triggerkonfiguration, liefert verschiedene Ansichten des Gesamtsystems, überwacht die einzelnen Boards und stellt Board-Daten, wie zum Beispiel den Inhalt des Wire-Rams, grafisch anschaulich dar.

Der Kontrollprozeß konnte bisher allerdings nur mit bis zu vier FLT-Boards eingesetzt werden, da nicht mehr Hardware zur Verfügung stand. Die Entwicklung der Hardware, dabei vor allem der TFU als wichtigstes und komplexestes Board, hat sich gegenüber der ursprünglichen Planung leider um über ein Jahr verzögert. Der Grund hierfür waren wiederholte Änderungen in den Spezifikationen der TFU, die jeweils ein Redesign der Hardware erforderten. Für den Betrieb des Kontrollprozesses standen daher neben dem Testboard nur zwei TFU-Prototypen und eine TPU zur Verfügung. Im Betrieb mit mehreren Crates und einer größeren Anzahl von Boards liegen daher keine praktischen Erfahrungen vor. Diese Implementierung des Kontrollprozesses ist daher nur als Prototyp anzusehen. Er zeigt, daß die verwendete Technologie für die grafische Oberfläche und die Netzwerkkommunikation, mit den `trun`-Prozessen als Vermittler zu den Board-Prozessen, funktioniert. Die Implementierung in Tcl/Tk war wie erwartet sehr effizient und die Anforderungen an die Oberfläche konnten damit erfüllt werden.

Als nächsten Schritt müssen nun Erfahrungen mit der Ende 1998 verfügbaren Kleinserie von 10 TFUs gesammelt werden, um darauf aufbauend die zentrale Kontrolle weiterzuentwickeln. Für die zukünftige Entwicklung ist auch zu beachten, daß die Steuerung und Überwachung des FLT eng in das Online-Konzept für das Experiment als Ganzes eingebunden werden muß. Leider ist dieses Online-Konzept bis heute nur in Umrissen erkennbar. Klar ist, daß die Boards ihre Betriebsdaten voraussichtlich in eine Datenbank schreiben müssen. Auf die Monitoring-Funktionen des Prototyps hat dies keine Auswirkungen, er bekommt lediglich seine Betriebsdaten nicht direkt, sondern über den Umweg der Datenbank. Aber die Rollenverteilung ändert sich, der zentrale Kontrollprozeß wäre in diesem Fall ein Client, der auf einen Datenbankserver zugreift.

Nicht geklärt ist, ob die Steuerung ebenfalls über die Datenbank abgewickelt werden muß oder direkt erfolgen kann und ob es mehrere Prozesse gibt, die steuernd eingreifen dürfen, eventuell in einer Hierarchie, ausgehend von der Steuerung des Gesamtexperiments, organisiert. Diese Fragen müssen am DESY geklärt werden und sind dann Grundlage für die Applikationsentwicklung der Online-Software des FLT.

Generell sollte man dabei versuchen die Schnittstellen zu den anderen Systemen so schmal und allgemein wie möglich zu halten, um die Abhängigkeiten noch überschauen zu können und den FLT in Testläufen ohne allzu großen Aufwand auch

autonom betreiben zu können.

Im Rahmen der Arbeit ist ein objektorientiertes Framework für die logische Simulation digitaler Schaltungen entworfen und implementiert worden. Für HERA-B wird eine Simulation des Gesamtsystems bestehend aus First Level Trigger und den drei Pretriggern benötigt. Weiterhin ist die Simulation für eine kontrollierte Inbetriebnahme des Triggersystems erforderlich und es werden Simulationen einzelner Boards im Rahmen von Hardwaretests gebraucht. Die Simulation des First Level Triggers muß auf den, im Rahmen der Harwaretests verwendeten, Einzelboardsimulationen aufbauen, um die Konsistenz mit der realen Hardware sicherzustellen. Die Simulationen der vier Subsysteme wiederum sollen dann in einer Gesamtsimulation von FLT und Pretriggern vereint werden. Das Framework dient dabei als Grundlage für die Entwicklung der jeweiligen Simulationsprogramme. Es hat die primäre Aufgabe die Architektur der Simulationen der einzelnen Subsysteme soweit festzulegen, daß sie problemlos in einer Gesamtsimulation vereint werden können.

Um dieses umfangreiche verteilte Entwicklungsprojekt technisch und organisatorisch zu bewältigen, wurde aktuellste Softwaretechnologie eingesetzt. Dies beinhaltet einen objektorientierten Entwurf mit Hilfe der Designmethode von Booch, die Verwendung von Entwurfsmustern, den Einsatz von Software Entwicklungswerkzeugen und die Vereinbarung von Programmierrichtlinien und Dokumentation.

Bisher wurde mit Hilfe des Frameworks eine komplette Simulation des Elektron-Pretriggers in Bologna programmiert. Die Simulation des Myon-Pretriggers in Berlin ist fast abgeschlossen und die des Hadron-Pretriggers in Moskau ist ebenfalls in Arbeit. Für den First Level Trigger wurde mit dem Framework eine Simulation der TFU und der TPU entwickelt, womit dann ein Multiprozessorsystem aus vielen dieser Boards simuliert werden kann. Die TFU-Simulation wurde zudem bereits in vergleichenden Hardwaretests mit dem TFU-Board eingesetzt. Eine Simulation der TDU steht noch aus. Alle FLT-Simulationen erfolgten bisher aber nur mit willkürlichen Testdaten, da von Seiten des DESY noch keine Datensätze mit der Detektorgeometrie vorhanden sind. Sie werden aber zur Berechnung der Lookup-Tabellen und damit für eine realistische Konfiguration des FLT zur Simulation von Detektorereignissen benötigt. Die Bereitstellung dieser Datensätze ist der nächste wichtige Schritt, sowohl für die Simulation, als auch für die Erprobung der Boards.

Weiterhin müssen die Simulationen der Subsysteme in einem Programm vereint werden, was bisher lediglich für Elektron-Pretrigger und FLT geschehen ist. Die Erfahrungen bei der Zusammenlegung von Elektron-Pretrigger- und FLT Simulation sind sehr positiv. Trotz des über Monate hinweg nur sehr sporadischen Kontakt zwischen Bologna und Mannheim konnten die Programme nach nur etwa zwei Tagen Arbeit zusammen kompiliert werden. Nachdem noch fehlende Teile des Frameworks ergänzt wurden, die für die Verbindung von Subsystemen benötigt werden, und die TFU-Simulation vervollständigt war, waren dann weitere zwei Tage notwendig,

um die beiden Subsysteme FLT und Elektron-Pretrigger zusammen simulieren zu können.

Eine wichtige noch fehlende Erweiterung ist eine Benutzerschnittstelle, um das ganze System zu handhaben. Die vorhandene Schnittstelle zur Handhabung einzelner Boards ist für eine Gesamtsimulation zu detailliert. Hier ist es unter Umständen sinnvoll, die Konfiguration der Simulation und die Darstellung der Ergebnisse teilweise in das Framework zu übernehmen.

Die Probleme des FLTSIM-Projekts liegen im personellen und organisatorischen Bereich. Es hat sich als schwierig erwiesen im Rahmen des Experiments unter den Physikern genügend Mitarbeiter zu finden, die das notwendige Wissen für eine Mitarbeit mitbringen oder willens sind und die Zeit haben, sich dies anzueignen. Zum Teil sind die personellen Probleme auch durch die problematische und zeitweise gar nicht vorhandene Projektleitung bedingt. Rückblickend zeigt sich, daß unbedingt ein Projektleiter erforderlich gewesen wäre, der von Anfang an den steten Fortgang des Projekts organisiert und die zentralen Entscheidungen trifft und umsetzt. Dies hätte den Fortgang des Projekts erheblich beschleunigen können, auch, indem zum Beispiel in der Anfangsphase die Diskussionen um Richtlinien und Methoden verkürzt worden wären. Hier konnte nur unter erheblichen Schwierigkeiten die Verwendung einer Designmethode (Booch), des entsprechenden Werkzeugs (Rose) und die Programmierrichtlinien durchgesetzt werden. Üblicherweise sind Simulationen für Physikexperimente nach wie vor in Fortran geschrieben. Dies trifft zum Beispiel für die anderen Simulationen im Rahmen von HERA-B zu (Second Level Trigger), das ARTE Paket mit den HERA-B Detektorinformationen und auch auf die weit verbreitete Hochenergiephysik-Simulation GEANT und zum Beispiel die komplette Software des ALEPH-Experiments am LEP (CERN). Aus diesem Grund war einige Überzeugungsarbeit notwendig, um die Verwendung einer objektorientierten Sprache mit weiterführenden Methoden und Werkzeugen zu ermöglichen.

Das Framework wurde nach dem aktuellen Stand der Softwaretechnik entworfen und implementiert. Insgesamt hat es sich als flexibel genug erwiesen, um für die verschiedenen Simulationen als Basis zu dienen. Für alle Teile existiert eine automatisch generierte Entwicklerdokumentation, die sich als sehr hilfreich herausgestellt hat.

# Literaturverzeichnis


[1] Iris Abt
*Physics reach of HERA-B*
Nucl. Instr. and Meth. A384 (1996) 113.

[2] H. Albrecht et al.
*HELENA, A Beauty Factory in Hamburg*
DESY 92/041, März 1992.

[3] American National Standard for Information Systems
*ANSI X3.159-1989: Programming Language C*
1989 – American National Standard Institute

[4] Grady Booch
*Object-oriented Analysis and Design with Applications*
Benjamin / Cummings (1994)

[5] Grady Booch
*Object solutions: managing the object-oriented project*
Addison-Wesley (1996)

[6] Lothar Borrmann
*Kleine und kleinste Kerne: Betriebssysteme mit Mikro- und Nanokernen*
Informationstechnik und Technische Informatik 38 (1996) 2

[7] Jeffrey Child
*What's real in real-time operating systems?*
June 1992 – Computer Design (Vol. 31, No 6, pp 107-117)

[8] George F. Coulouris und Jean Dollimore
*Distributed Systems Concept and Design*
Second Edition
1994 – Addison-Wesley; London

[9] Margaret A. Ellis und Bjarne Stroustrup
*The Annotated C++ Reference Manual*

1990 – AT&T Bell Laboratories; Murray Hill (New Jersey (USA));
Addison-Wesley; Reading (Massachusetts (USA))

[10] D. R. Engler, M. F. Kaashoek und J.W. O'Toole Jr.
*The Operating System Kernel as a Secure Programmable Machine*
Januar 1995 – Operating Systems review – ACM-Press (Vol 29. Nr. 1, pp 78-81)

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
*Design Patterns*
Addison Wesley 1995.

[12] Alexander Genther
*NERV – Ein Entwicklungssystem zur Simulation neuronaler Netzwerke*
Diplomarbeit im Studiengang Physik der Ruprecht-Karls-Universität Heidelberg
1989 – Lehrstuhl für Informatik V, Universität Mannheim

[13] Ekkehard Gerndt
*FLT Latency*
Hera-B Note 98-021, Februar 1998

[14] Carsten Hageböke
*Systemsoftware des First Level Triggers für das HERA-B Experiment am DESY*
Diplomarbeit im Studiengang Wirtschaftsinformatik der Universität Mannheim
11. Februar 1997 – Lehrstuhl für Informatik V, Universität Mannheim

[15] Christian Hähnel
*Verschiedene Konzepte zur digitalen Hochgeschwindigkeitsübertragung über Leitungen*
Diplomarbeit im Studiengang Physik der Ruprecht-Karls-Universität Heidelberg
29. September 1995 – Lehrstuhl für Informatik V, Universität Mannheim

[16] HERA-B Collaboration
*HERA-B: An Experiment to Study CP Violation in the B System Using an Internal Target at HERA Proton Ring*
Design Report, DESY 95/01
1995 – Hamburg (DESY)

[17] Werner Hilf und Anton Nausch
*M68000 Familie*
*Teil 1, Grundlagen und Architektur*
1984 – tewi-Verlag; München

[18] Werner Hilf und Anton Nausch
*M68000 Familie*
*Teil 2, Anwendungen und 68000-Bausteine*
1984 – tewi-Verlag; München

[19] Fridolin Hofmann
*Betriebssysteme: Grundkonzepte und Modellvorstellungen*
2. überarbeitete Auflage
1991 – Teubner-Verlag; Stuttgart

[20] Reiner Hauser
*Ein Multiprozessorsystem zur Simulation neuronaler Netzwerke*
Diplomarbeit im Studiengang Physik der Ruprecht-Karls-Universität Heidelberg
1989 – Lehrstuhl für Informatik V, Universität Mannheim

[21] Reiner Hauser
*Parallele Genetische Algorithmen und die Optimierung von Field Programmable Gate Arrays*
Dissertation an der Ruprecht-Karls-Universität Heidelberg
November 1994 – Lehrstuhl für Informatik V, Universität Mannheim

[22] Heiko Itterbeck
*Techniques and Physics of the H1-Central-Muon-Central-Trigger System*
Dissertation an der RWTH Aachen, September 1997

[23] Jia, Xiaohua und Maekawa, Mamoru
*Operating System Kernel Automatic Construction*
1995 – Operating Systems review – ACM-Press (Vol 29. Nr. 2, pp 78-81)

[24] R.E. Kalman
*A New Approach to Linear Filtering and Prediction Problems*
Trans. ASME, J. Basic Engineering series D, Vol82, 35 (1960)

[25] R.E. Kalman
*New Results in Linear Filtering and Prediction Problems*
Trans. ASME, J. Basic Engineering series D, Vol83, 95 (1961)

[26] Brian W. Kernighan und Dennis M. Ritchie
*The C Programming Language, Second Edition*
1988 – AT&T Bell Laboratories; Murray Hill (New Jersey (USA));
Addison-Wesley; Reading (Massachusetts (USA))

[27] Thomas Kihm
*Dokumentation zur VME-Bus Library V1.4.6 von MIZZI Computer Systems*
September 1995 – Mannheim

[28] Gernot Kuhr
*Inbetriebnahme und Test von Komponenten des HERA-B First Level Triggers*
Diplomarbeit im Studiengang Physik der Ruprecht-Karls-Universität Heidelberg
August 1996 – Lehrstuhl für Informatik V, Universität Mannheim

[29] Jean J. Labrosse
*µC/OS – The Real-Time Kernel*
1992 – R & D Publications; Lawrence (Kansas (USA))

[30] Erich Lohrmann
*Das Standardmodell auf dem Prüfstand*
Phys. Bl. 53 (1997) Nr.10 967.

[31] Reinhard Männer
*Entwicklung und Bau eines First-Level-Triggers für das Vorhaben: „Experiment zum Studium der CP-Verletzung in Proton-Kern-Reaktionen am Speicherring HERA“*
Antrag auf Erlangung von Förderungsmitteln für die Entwicklung des First Level Triggers 27. April 1995 – Lehrstuhl für Informatik V, Universität Mannheim

[32] Mike Medinnis
*HERA-B Trigger and DAQ System Architecture*
Beauty 97, UCLA Los Angeles. (1997)

[33] Bertrand Meyer
*Objektorientierte Softwareentwicklung*
Hanser (1990)

[34] Motorola
*MC68000 Family Reference Manual*
1990 – Motorola Inc. (USA)

[35] Object Management Group (OMG)
Internet: http://www.omg.org/news/pr97/umlpr.htm, November 1997.

[36] John Ousterhout
*Tcl und Tk*
Addison Wesley 1994.

[37] John Ousterhout
*Scripting: Higher Level Programming for the 21st Century*
IEEE 1997.

[38] Bernd Page
*Diskrete Simulation*
Springer 1991

[39] Rational Software Corporation
*The Unified Modeling Language, Version 1.1*
Internet: http://www.rational.com, September 1997.

[40] Dominik Ressing
*First level trigger for HERA-B*
Nucl. Instr. and Meth. A384 (1996) 131.

[41] Dominik Ressing
*DAQ Architecture for HERA-B*
Internal Paper for the collaboration. (1996)

[42] Lutz Richter
*Betriebssysteme (2. Auflage)*
1985 – Teubner-Verlag; Stuttgart

[43] A.D. Sakharov
*Violation of CP Invariance, C Asymmetry and Baryon Asymmetrie of the Universe*
ZhETF Pisma 5 (1967) 32; Sov. Phys. JETP Lett. 5 (1967) 24.

[44] Martin Schader und Michael Rundshagen
*Objektorientierte Systemanalyse, eine Einführung*
1994 – Springer-Verlag; Berlin, Heidelberg

[45] Wolfram Schiffmann und Robert Schmitz
*Technische Informatik (Grundlagen der Computertechnik)*
1994 – Springer-Verlag; Berlin, Heidelberg

[46] Martin A. Schinkmann
*Das Betriebssystem des Neurocomputers Synapse*
Diplomarbeit im Studiengang Physik der Ruprecht-Karls-Universität Heidelberg
April 1992 – Lehrstuhl für Informatik V, Universität Mannheim

[47] Hans Albrecht Schmid
*Systematic Framework Design by Generalization*
Communications of the ACM, vol. 40/No10, 48-51 (Oct.1997)

[48] Hans Albrecht Schmid
*Objektorientierte Entwurfsmuster und Frameworks in der Informatik-Ausbildung an der Fachhochschule Konstanz*
Informatik Spektrum 20:364-371 (1997)

[49] Walter Schmidt-Parzefall
*HERA-B - Ein neues Experiment bei DESY*
Phys. Bl. 53 (1997) Nr.4 319.

[50] Joachim Spengler
*HERA-B: overview and 1996 enineering run*
Nucl. Instr. and Meth. A384 (1996) 106.

[51] Richard M. Stallman
*Using and Porting GNU CC for version 2.7.2*
November 1995 – ftp://phi.sinica.edu.tw/pub/aspac/gnu/ps/gcc-2.7.2.full.ps

[52] W. Richard Stevens
*Programmierung in der UNIX-Umgebung: die Referenz für Fortgeschrittene*
Addison-Wesley (1995)

[53] Andrew S. Tanenbaum
*Computer networks*
(Second Edition)
1989 – Prentice-Hall; Englewood Cliffs (New Jersey (USA))

[54] Andrew S. Tanenbaum
*Moderne Betriebssysteme*
(Zweite Auflage)
Hanser (1995))

[55] Andrew S. Tanenbaum
*Structured Computer Organization*
(Third Edition)
1990 – Prentice-Hall; Englewood Cliffs (New Jersey (USA))

[56] Brent Welch
*Practical Programming in Tcl and Tk*
Prentice Hall 1995.

[57] Andreas Wurz
*Hera-B FLT – Test-Board*
November 1995 – Mannheim

[58] Thomas Wolf
*Software Concept for the HERA-B First-Level-Trigger*
Januar 1996 – Mannheim

[59] John Zweizig
*Data aquisition in the HERA-B an LHC era*
Nucl. Instr. and Meth. A384 (1996) 147.

# Vielen Dank !

An alle, die mir während meiner Promotion durch ihre Unterstützung eine wichtige Hilfe waren. Besonderer Dank gilt meinen Eltern, die mir immer einen wertvollen Rückhalt gegeben haben.

Für ihre vielfältige Unterstützung möchte ich mich auch besonders bei Maria und Herbert Gabriel, der durch seinen plötzlichen Tod dies leider nicht mehr selbst lesen kann, bedanken.

Meiner, während der Promotion größer gewordenen, Familie danke ich für das Mittragen der vor allem in der Schlußphase aufgetretenen Belastungen.

Herrn Prof. Dr. R. Männer gilt mein Dank dafür, daß ich die Promotion an seinem Lehrstuhl durchführen konnte und für seine immer vorhandene, offene Gesprächsbereitschaft.

Bei den Mitgliedern der Arbeitsgruppe Dr. Joachim Gläß, Alexander Gröpel, Carsten Hageböke, Christian Hähnel und Andreas Wurz bedanke ich mich für die gute Zusammenarbeit.

Bedanken möchte ich mich auch bei Andrea Seeger für ihre logistische Unterstützung und bei Martin Schinkmann für die interessanten und hilfreichen Diskussionen hauptsächlich in der Anfangsphase der Arbeit.