Technische Universität Chemnitz
Straße der Nationen 62
09111 Chemnitz

FPGA-basierte Fail-safe-Schnittstellen für eingebettete Systeme

Studienarbeit

Fakultät für Informatik
Studiengang Informatik
Professur Technische Informatik
Vertiefungsrichtung Eingebettete Systeme

von
Marcel Flade
Dorfstraße 81
09526 Pfaffroda-Hallbach

Betreuer: Dipl. Inf. Stefan Ihmor, Universität Paderborn
Prüfer: Prof. Dr. habil. Wolfram Hardt, TU Chemnitz

im
November 2003

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Aufgabenstellung

FPGAs sind konfigurierbare Hardwarekomponenten, die mittlerweile beachtliche Rechenleistung bereitstellen und eine große Anzahl an I/Os vorweisen. Die Herausforderung besteht im Einsatz von FPGA-Boards in eingebetteten Systemen unter Echtzeitbedingungen. Da Kommunikation nicht immer Echtzeitfähig gestaltet werden kann (z.B. Internet) soll ein FPGA als Kommunikationsmodul in einer nicht echtzeitfähigen Kette von Elementen eingebaut werden, um auch Komponenten mit harten Echtzeitbedingungen in eingebetteten Systemen einzubinden. Dabei übernimmt das FPGA-Board die Funktion, im Fall von zu großen Verzögerungen oder des Ausfalls des Kommunikationspartners, automatisch Fail-safe-Daten für die Komponenten mit harter Echtzeit zu generieren. Die Neuheit dieses Ansatzes besteht darin, das Fail-safe-Verhalten nicht erst in der Endkomponente sondern bereits in der verbindenden Schnittstelle und damit transparent für die Komponente zu erzeugen. Im Gegensatz zu Softwarelösungen bietet ein FPGA-Board die Möglichkeit des Rapid-Prototyping in Hardware.

1.2 Gliederung der Arbeit

Das 1. Kapitel enthält die Aufgabenstellung und einen kurzen Überblick über die Gliederung der Arbeit. Kapitel 2 betrachtet die Grundlagen, die zur Lösung der Aufgabenstellung notwendig waren. Dabei wird auf das Modell des Interfaceblocks und den Protokollbegriff eingegangen. Des Weiteren wird der Aufbau von FPGA's erklärt. Das 3. Kapitel betrachtet das Fail-safe-Verhalten und zeigt verschiedene Möglichkeiten der Fehlererkennung allgemein und an verschiedenen Protokollen. Außerdem erklärt es, wie Fail-safe-Verhalten in Schnittstellen integriert werden kann. Kapitel 4 stellt die Implementierung eines Demonstrators vor, in welchem die in dieser Arbeit vorgestellten Konzepte und Ergebnisse kombiniert und validiert werden. Das 5. Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf Weiterentwicklungsmöglichkeiten.

2 Grundlagen

In diesem Kapitel werden die Grundlagen betrachtet, die zur Lösung der Aufgabenstellung notwendig sind und zum Grundverständnis der folgenden Kapitel beitragen sollen. Als erstes wird das Modell des Interfaceblocks allgemein betrachtet und erklärt. Im Anschluß erfolgt die Erläuterung des Protokollbegriffs. Der Aufbau von FPGAs wird schließlich im letzten Teil des Kapitels beschrieben.

2.1 Das Modell des Interfaceblock (IFB)

Der Interfaceblock (IFB) ist ein Modell, dass erstmals in der Diplomarbeit von Stefan Ihmor [Ihm01] vorgestellt wurde.

Das Konzept des IFB wurde erdacht, um Schnittstellen zwischen unterschiedlichen Kommunikationskomponenten automatisiert generieren zu können, ohne Änderungen an den Komponenten selbst vorzunehmen. Kommunikationskomponenten können einerseits komplexe Kommunikationsstrukturen, wie Bussystem, oder andererseits funktionale Komponenten wie Algorithmen sein. Diese werden in der IFB-Terminologie entsprechend als Medium bzw. Task bezeichnet.

Als Beispiele für ein Medium könnte der Anschluss eines Joysticks, die RS-232-Schnittstelle oder der PCI-Bus aufgeführt werden. Der Begriff Medium wird in diesem Zusammenhang nicht für die simple Verdrahtung der Schnittstellen von Task und Medium verwendet. Unterschiedliche Beispiele für Tasks könnten ein Analog-Digital-Wandler, ein JPEG-Encoder, oder auch die Steuerungen eines technischen Systems sein. Der Interfaceblock bietet die Möglichkeit Medien, Tasks oder eine Mischung aus Medien und Tasks miteinander zu verbinden. Da in einem Interfaceblock Daten zwischen Sender und Empfänder transformiert werden, kann dieser als Adapter zwischen physisch inkompatibel bzw. sematisch inkompatibel Kommunikationskomponenten eingesetzt werden. Dabei werden die physikalische Struktur, elektrische Eigenschaften und die Protokolle der Schnittstelle berücksichtigt.

Der Interfaceblock ist modular aufgebaut. Dies bringt einige entscheidende Vorteile mit sich. Beim Entwurf bietet es die Möglichkeit, die Arbeit zu unterteilen und diese dann später zusammen zu fügen. Weiterhin erhöht dieser

Ansatz die Wiederverwendbarkeit für andere Projekte. Dies spart Entwicklungszeit und -kosten. Eine weiterer Vorteil besteht darin, Intellectual Properties (IPs), also Komponenten von Drittanbietern, als Teil des IFB einzubinden. Die Vision des vorgestellten Konzepts ist eine voll-automatische Generierung von Schnittstellen in Form von Interfaceblöcken basierend auf einer XML-Beschreibung die in [Fic03] vorgestellt wird.

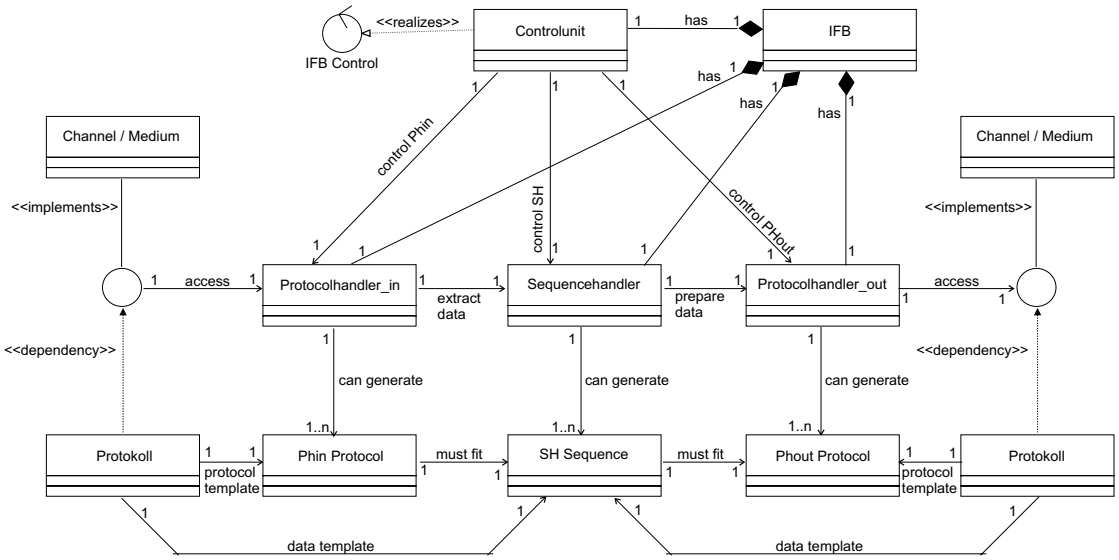


Abbildung 2.1: Klassendiagramm des Interfaceblocks

Der strukturelle und funktionale Aufbau eines IFB ist im Klassendiagramm in Abbildung 2.1 veranschaulicht. Die Struktur des Interfaceblock wird als Makrostruktur bezeichnet. Sie ist in Abbildung 2.2 dargestellt. Bestandteile des Interfaceblocks sind die Kontrolleinheit, zwei Protokollhandler und ein Sequenzhandler. Jeder Handler enthält mindestens einen Modus, der das dynamische Verhalten des Interfaceblocks implementiert. Im den folgenden Abschnitten werden diese einzelnen Komponenten in ihrem Aufbau und ihrer Funktionalität näher erläutert.

2.1.1 Die Kontrolleinheit (CU)

Die Kontrolleinheit (vgl. Abbildung 2.2) übernimmt die Steuerungsaufgaben innerhalb des Interfaceblocks. Sie koordiniert und kontrolliert die Funktionalität der Protokollhandler und des Sequenzhandlers. Dazu bestimmt

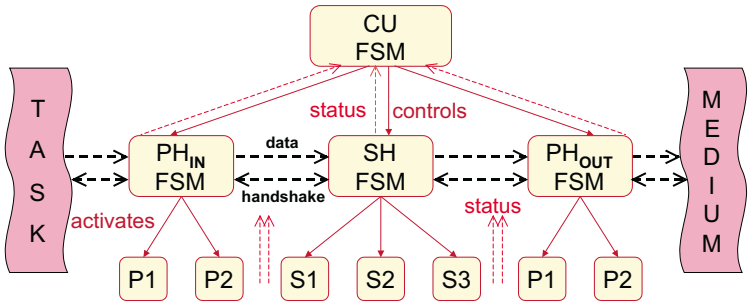


Abbildung 2.2: Makrostruktur des Interfaceblocks

die Kontrolleinheit den jeweils aktiven Modus eines Handlers, welcher die auszuführende Funktionalität implementiert, die mittels einer Automatenbeschreibung modelliert wurde. Das Wissen über den aktuellen Zustand der Schnittstelle bezieht die Kontrolleinheit aus Rückmeldungen in Form von Statussignalen aus den Handlern. Ein Teil der Steuerlogik in der Kontrolleinheit sind Timer, die die Zeitbasis der Schnittstelle implementieren. Dadurch ist es möglich den Interfaceblock als Echtzeitschnittstelle zu betreiben.

2.1.2 Die Protokollhandler (PH)

Die Protokollhandler sind die externen Schnittstellen des Interfaceblocks. Hier erfolgt die eigentliche Kommunikation mit allen verbundenen Tasks und Medien. Ein Protokollhandler kann, basierend auf seinen Modi, die Protokolle der jeweiligen Kommunikationskomponenten verarbeiten. Das beinhaltet das sowohl das Senden als auch das Empfangen von Daten.

Durch Umschalten zwischen unterschiedlichen Modi ist es möglich, verschiedene Protokolle zu verarbeiten. Der Handler fungiert hier als "Schalter", wobei die Modi für die eigentliche Generierung oder das Auslesen der Protokolle zuständig sind. Hierbei übernimmt der Protokollhandler die Funktion der Schnittstelle zwischen Kontrolleinheit und Modi sowie zwischen Kommunikaitonskomponente und Sequenzhandler. Dabei werden in einem Modus die Nutzdaten von dem redundanten Teil des Protokolls getrennt. Diese Nutzdaten werden dann an den Sequenzhandler weitergeleitet.

2.1.3 Der Sequenzhandler (SH)

Der Sequenzhandler ist das Bindeglied zwischen den Protokollhandlern des Interfaceblocks. Basierend auf einer Abbildungsvorschrift werden hier die eingehenden Nutzdaten konform zum ausgehenden Protokoll transformiert. Dazu nimmt der Sequenzhandler Daten vom eingehenden Protokollhandler entgegen und kann diese in Abhängigkeit vom ausgehenden Protokollhandler modifizieren. Dazu verfügt der Sequenzhandler über entsprechende Modi, die neben einer strukturellen Änderung von Daten auch eine semantische Änderung ermöglichen. So weden für den ausgehenden Protokollhandler fertige Datenpakete vorbereitet, die dieser dann nur noch in das Protokoll integrieren und versenden muss.

Wie schon der Protokollhander, bildet auch der Sequenzhandler die Schnittstelle zwischen seinen Modi und der Kontrolleinheit und führt den Wechsel dieser Modi aus. Die Steuerung übernimmt auch hier die Kontrolleinheit.

2.1.4 Der Modus

Modi werden sowohl im Protokoll- als auch im Sequenzhandler benötigt. Sie realisieren das dynamische Verhalten des Interfaceblocks. Als frei definierbarer Automat verfügt ein Modus generell über keine festgelegte Funktionalität, es gibt allerdings einige Restriktionen, der den Aufbau des endlichen Automaten (FSM) reglementiert.

Als Instanz übernimmt ein Modus dann eine spezialisierte Aufgabe in einem Handler, wie oben beschrieben. Die direkte Steuerung eines Modus übernimmt der jeweilige Handler. Die eigenlichen Steuersignale dazu enstammen aber sämtlich der Kontrolleinheit. Trotz der einheitlichen Beschreibung als FSM realisiert ein Modus unterschiedliche Funktionalität als Protokollhandler bzw. Sequenzhandler.

Als Protokollhandlermodus übernimmt er die Verarbeitung eines Protokolls eines Mediums oder einer Task. Dabei implementiert der Modus einen Kommunikationsautomaten, der genau ein Protokoll verarbeiten kann. Dazu wird jeder Kommunikationsautomat als komplementäre Schnittstelle zur verbundenen Kommunikationskomponente erzeugt. Zur Protokollverarbeitung muss der Automat die empfangenen Nutzdaten über den Protokollhandler an den Sequenzhandler leiten und abgehende Daten, die er vom Sequenzhandler erhalten hat, konform in das entsprechende Protokoll integrieren. Durch die Anordnung mehrerer Modi in einem Protokollhandler und der Möglichkeit,

diese einzeln zu aktivieren, ist es möglich auf unterschiedliche Protokolle zuzugreifen.

Der Modus eines Sequenzhandlers muß nicht mit der externen Welt kommunizieren. Er übernimmt Transformationen auf den zwischen den Protokollhandlern transferierten Nutzdaten. Ein Sequenzhandlermodus kann Daten strukturell umwandeln, Berechnungen darauf ausführen und sogar neue Daten nach einem vorgegebenem Schema generieren. Die kurzzeitige Speicherung von Daten ist hier ebenfalls möglich. Ausgehend von einem einfachen Durchreichen der Daten bis hin zu komplexen Algorithmen, ist eine vielfältige Funktionalität in einem Sequenzgeneratormodus modellierbar.

2.2 Definition des Protokollbegriffs

Zur Definition des Protokollbegriffs, auf den in den folgenden Kapiteln noch häufiger eingegangen wird, ist zunächst die Betrachtung des Aufbaus von Netzwerken notwendig. In der Regel sind Netzwerke schichtenweise aufgebaut. Dies ist notwendig um die Komplexität bei der Entwicklung beherrschbar zu halten. Eine Standardisierung dieses Schichtenmodells, die von der

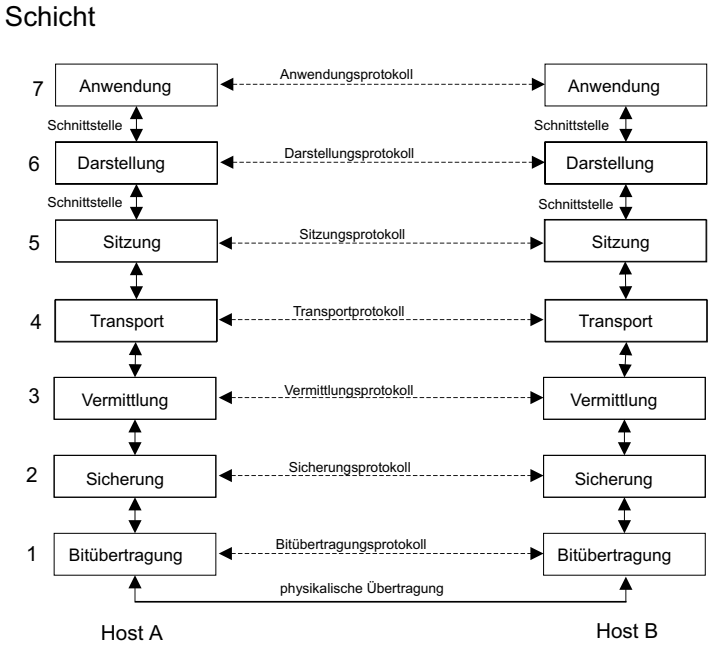


Abbildung 2.3: Das ISO-OSI-Referenzmodell

International Standards Organisation (ISO) entwickelt wurde, stellt das OSI-Referenzmodell (Abbildung 2.3) dar. OSI bedeutet Open System Interconnection, also Kommunikation offener Systeme. Das Modell beschreibt 7 hierarchische Schichten innerhalb eines Netzwerkes, die Bitübertragungs-, Sicherungs-, Vermittlungs-, Transport-, Sitzungs-, Darstellungs- und Anwendungsschicht. Eine sehr umfassende Darstellung des OSI-Refernzmodells ist in [Tan03] zu finden. Eine gute Zusammenfassung bietet [Wie]. In Abbildung 2.3 wird der Aufbau des OSI-Referenzmodells veranschaulicht. Allerdings ist es nicht immer möglich, eine Netzwerkarchitektur in die 7 Schichten zu unterteilen, da manchmal eine reale Schicht die Aufgaben von zwei oder mehr Modellschichten übernimmt.

Die Bitübertragungsschicht ist für die Übertragung der Bits über den Kommunikationskanal zuständig. In der Sicherungsschicht werden aus dem Bitstrom wieder Datenpakete gewonnen. Mit dem Routing der Pakete innerhalb des Subnetzes beschäftigt sich die Vermittlungsschicht. Die Transportschicht übernimmt Daten von der Sitzungsschicht und teilt sie, wenn nötig, in kleinere Teile auf. Sie wird auch dazu genutzt, um die Sitzungsschicht unabhängig von der Hardware zu halten. In der Sitzungsschicht können Vorkehrungen getroffen werden, um Verbindungen nach Abbrüchen wiederherstellen zu können. Außerdem steuert es die gerade laufende Sitzung hinsichtlich Dialogsteuerung, Token-Management und Synchronisation. In der Darstellungsschicht ist die Syntax und Semantik der übertragenen Daten von Bedeutung. Es können Datentypen definiert werden, Daten verschlüsselt oder entschlüsselt werden. In der letzten Schicht, der Anwendungsschicht, werden die Daten des Senderprozesses nun genutzt. In dieser Schicht sind eine Vielzahl von Anwendungsprotokollen definiert, wie zum Beispiel HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol) und SMTP (Simple Mail Transfer Protocol) auch besser bekannt als E-Mail.

Der theoretische Übertragungsweg erfolgt nur zwischen den gleichen Schichten bei Sender und Empfänger. Damit die Kommunikation innerhalb einer Schicht stattfinden kann, ist ein Protokoll nötig. Das Protokoll legt die Regel und Bestimmungen fest, nach denen die Kommunikation abläuft. Jede Schicht hat ihr eigenes Protokoll.

Die reale Kommunikation findet auf der Senderseite über die Schichten von oben nach unten statt. Die untere Schicht stellt jeweils für die übergeordnete Schicht einen Dienst bereit, der die Daten entgegennimmt und weiterleitet. Die Nachricht wird von der Anwendungsschicht bis zur Bitübertragungsschicht durchgereicht. Dabei werden die Daten von der oberen Schicht in das schichteigene Protokoll verpackt und als Nutzdaten an die darunterliegende

Schicht weitergegeben. Erst auf der Bitübertragungsschicht findet die eigentliche Datenübertragung statt. Auf der Empfängerseite wird die empfangene Nachricht dann Schicht für Schicht nach oben gereicht. Jede Schicht entfernt das aktuelle Transportprotokoll, extrahiert die Nutzdaten und reicht sie an die darüberliegende Schicht weiter.

Für die reale Kommunikation kann auch ein Interfaceblock (vgl. Abschnitt 2.1) zum Einsatz kommen. Er läßt sich in das ISO-OSI-Referenzmodell einordnen. Der Interfaceblock kann eine Verbindung zwischen den Schichten herstellen. Somit können Lücken zwischen den Schichten geschlossen werden, für die keine Implementierung existiert. Es ist aber auch möglich mit dem Interfaceblock ganze Schichten zu überbrücken.

2.3 FPGA

FPGA's (Field programmable Gate Array) sind programmierbare Hardwarebausteine und gehören zur Gruppe der anwenderprogrammierbaren Schaltungen. Mit ihnen ist die Realisierung von kombinatorischen und sequentiellen digitalen Schaltungen möglich. Im Allgemeinen bestehen FPGA's aus programmierbaren Logikblöcken (CLB) zur Realisierung von kombinatorischen

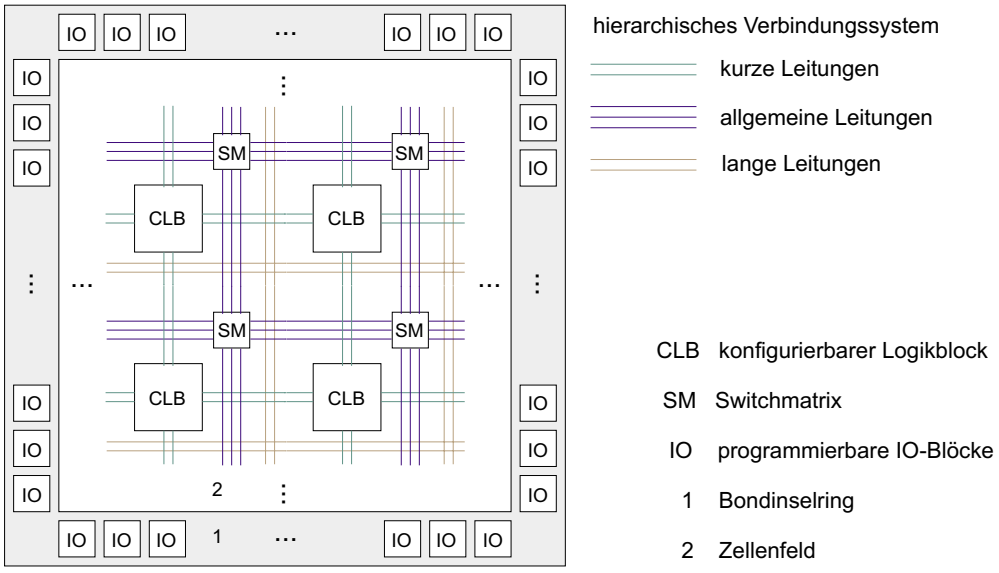


Abbildung 2.4: Der allgemeine Aufbau einer FPGA

und sequentiellen Schaltungen, konfigurierbaren I/O-Blöcken zur Kommunikation nach außen und einer programmierbaren hierarchischen Verbindungsstruktur, die die Logikblöcke untereinander und mit den I/O-Blöcken verbinden kann. Der Aufbau ist nocheinmal in Abbildung 2.4 veranschaulicht.

Zur Konfiguration der FPGA's existieren zwei verschiedene Technologien, die antifusebasierte und die SRAM-basierte. FPGA's mit Antifuses sind nur einmal konfigurierbar, da dort durch hohe Programmierspannungen Isolatorschichten zwischen zwei Leitungen aufgeschmolzen werden und somit eine leitende Verbindung entsteht. Der Vorteil dieser Technologie ist, das die Konfiguration nicht flüchtig ist. Wenn man den Strom abschaltet, dann bleibt sie bestehen. Außerdem sind Antifuses kleiner als SRAM-Zellen. Bei der SRAM-Technologie werden SRAM-Zellen als Speicher genutzt. Diese Zellen sind mit dem Gate eines Schalttransistor verbunden, der je nach Wert in der SRAM-Zelle leitend oder gesperrt ist. Der Nachteil dabei ist, dass die SRAM-Zellen ihren Wert verlieren, sobald der Strom abgeschalten wird. Um ein FPGA wieder in Betrieb zu nehmen, muss ein erneutes Laden der Konfiguration erfolgen. Allerdings ist es dadurch möglich, die FPGA mit anderen Konfigurationen zu laden. Dies eignet sich besonders zu Testzwecken in der Entwicklungsphase. Außerdem ist es vorteilhaft, wenn spätere Änderungen an der implementierten Funktionalität notwendig werden.

Ausführlichere Angaben zur Technologie von antifuse und SRAM-basierten FPGA's sind in [Wan98] und [Mül03] zu finden. Im Folgenden werden die Bestandteile einer SRAM-basierten FPGA näher erläutert. Dabei wird genauer auf die Komponenten der Spartan 2E von Xilinx eingegangen, da diese FPGA die Zielplattform des Demonstrators aus Kapitel 4 bildet.

2.3.1 Konfigurierbare Logikblöcke

Die konfigurierbaren Logikblöcke (CLB) realisieren die kombinatorischen und sequentiellen Funktionen eines FPGA. Dies geschieht auf der Grundlage von Look-up-tables (LUT) und Flipflops. Die LUT's bestehen aus SRAM-Zellen, die abhängig von der Eingangsbelegung ausgelesen werden. Damit können sehr komplexe kombinatorische Funktionen mit einer konstanten Laufzeit realisiert werden, da die LUT eine konstante Zeit zum Auslesen des Wertes braucht. Bei einigen FPGA-Typen, wie dem Virtex-II von Xilinx [Xil03b], ist es außerdem möglich, die LUT's als RAM zu benutzen. Mit den Flipflops lassen sich sequentielle Schaltungen realisieren.

Die CLB's der Spartan 2E bestehen aus zwei gleichen Teilen. Jeder dieser Teile besitzt zwei LUT's mit jeweils 4 Eingängen, einer Carry und Control Logik für jede LUT und zwei Flipflops. Der Aufbau ist in Abbildung 2.5 veranschaulicht. Die LUT's des Spartan 2E können als LUT, RAM oder Schieberegister genutzt werden. Die Flipflops lassen sich als flankengetriggerte D-Flipflops oder als pegelgesteuerte Latches benutzen. Außerdem besitzt jeder Teil des CLB einen synchron oder asynchron betreibbaren Setz- und Rücksetzeingang. Eine Carry und Control Logik ermöglicht außerdem eine schnelle Implementierung von arithmetischen Funktionen.

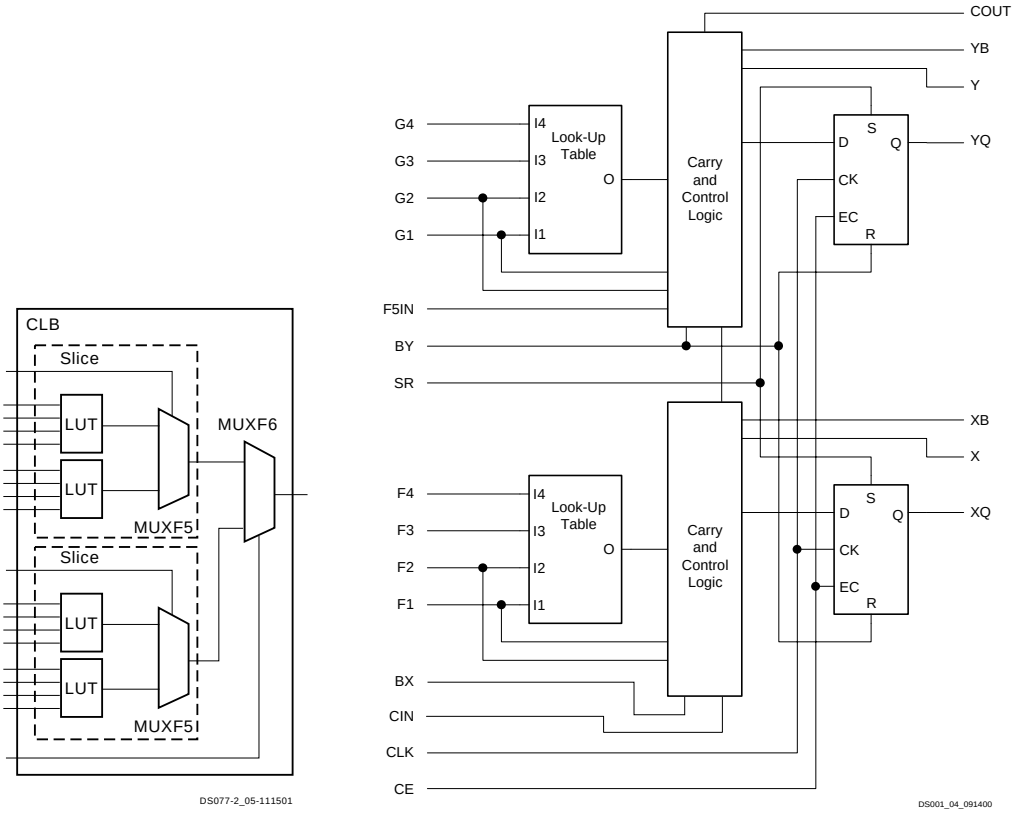


Abbildung 2.5: Konfigurierbarer Logikblock einer Xilinx Spartan 2E FPGA, Quelle [Xil03a]

2.3.2 IO-Blöcke

Die IO-Blöcke stellen die Verbindung zwischen den Logikblöcken und der Außenwelt her. Ein IO-Block ist mit einem Pin des Gehäuses verbunden

und hat verschiedene Betriebsmodi. Er kann so konfiguriert werden, dass er Eingang, Ausgang, oder Bidirektional ist. Es können verschiedene Pegel und Frequenzen konfiguriert werden. So kann man bei der Spartan 2E zum Beispiel direkt an einen PCI oder AGP-Bus gehen oder auch LVTTL (Low Voltage TTL) anbinden. Des Weiteren beinhalten IO-Blöcke Flipflops, womit es möglich ist, den Eingang zu puffern. Der Aufbau des IO-Blocks der Spartan 2E ist in Abbildung 2.6 zu sehen.

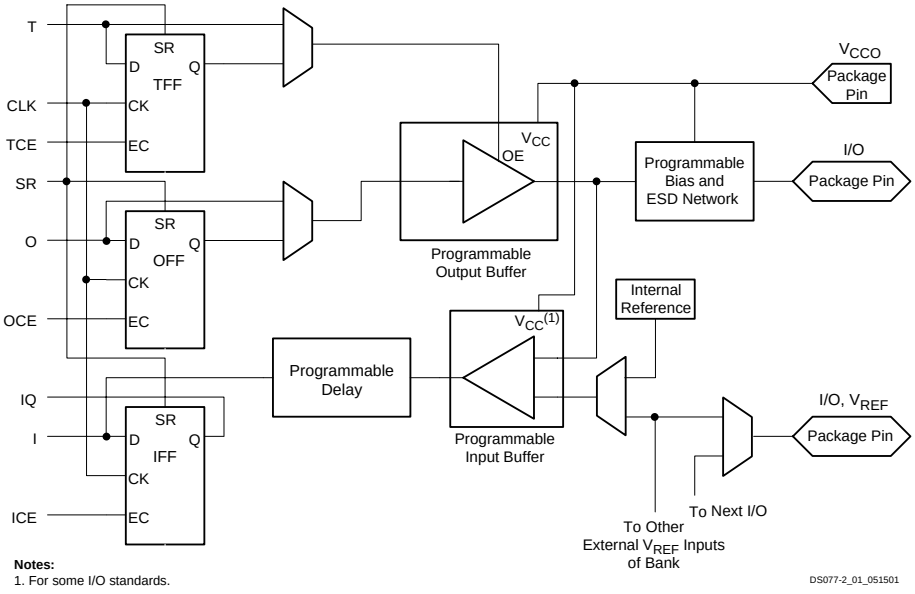


Abbildung 2.6: IO-Block einer Xilinx Spartan 2E FPGA, Quelle [Xil03a]

2.3.3 Verbindungsstruktur

Im Allgemeinen haben FPGAs drei Arten von Verbindungsleitungen. Es gibt lange, kurze und allgemeine Leitungen. Das Routing erfolgt über Schalttransistoren an denen sich eine SRAM-Zelle befindet. Durch den Wert in der Zelle wird festgelegt, ob zwei Leitungen miteinander verknüpft werden oder nicht. Der Nachteil der Schalttransistoren ist, dass sich mit jedem Transistor eine Verzögerungszeit zur Leitungsverzögerung addiert. Die Geschwindigkeit der Schaltung wird letzendlich durch die größte Leitungsverzögerung, die den kritischen Pfad bildet, bestimmt. Deshalb wird beim Routing versucht, die Leitungen so schnell wie möglich zu halten, also so wenige Schalttransistoren wie möglich zu benutzen. Die drei Leitungsarten des FPGA haben deshalb

auch unterschiedliche Eigenschaften ihre Länge und Routingmöglichkeiten betreffend.

Die **langen Leitungen** dienen dazu, Signale über große Distanzen zu routen. Sie verlaufen horizontal und vertikal durch das FPGA. Lange Leitungen haben wenige potentielle Verbindungspunkte zu den CLB und müssen somit nur wenige Schalttransistoren überwinden. Das wirkt sich positiv auf die Signallaufzeiten aus. Allerdings gibt es nur eine relativ geringe Anzahl dieser Leitungen innerhalb des FPGA.

Kurze Leitungen werden verwendet, um Verbindungen von einem CLB zu einem benachbarten herzustellen. Außerdem stellen sie Verbindungen zwischen CLB und Switchmatrix her, um Signale auch über benachbarte CLBs hinaus zu transportieren. Ihre Länge ist gering. Dafür sind aber schnelle Verbindugnen möglich.

Mit den **allgemeinen Leitungen** ist ein Routing zwischen den Switchmatrizen möglich. Switchmatrizen sind programmierbare Verbindungsstrukturen. Die Verbindungen werden wieder über Schalttransistoren realisiert. Aus diesem Grund sind die allgemeinen Leitungen die langsamsten aber auch die flexibelsten Verbindungsstrukturen eines FPGA.

Bei der Spartan 2E gibt es auch diese drei Arten von Leitungen. Die kurzen Leitungen, oder lokale Leitungen wie sie im Datenblatt [Xil03a] genannt werden, können Verbindungen zu benachbarten CLBs oder zur Switchmatrix (Routingmatrix) herstellen. Sie können aber auch rückgekoppelt werden. Die allgemeinen Leitungen verlaufen immer zwischen banachbarten Routingmatrizen. Von jeder Routingmatrix gehen je 24 Leitungen nach links, rechts, oben und unten. Zusätzlich gibt es noch gepufferte Leitungen, die über eine Länge von sechs CLB's verlaufen und auf die nur an den Endpunkten oder in der Mitte zugegriffen werden kann. Globale Leitungen gibt es je 12 Stück pro Spalte und Zeile. Sie verlaufen über die gesamte Länge des FPGA. Eine zusätzliche Verbindungsstruktur bildet der VersaRing. Er verbindet die IO-Blöcke mit dem CLB-Feld. Das Spartan 2E FPGA bietet des weiteren die Möglichkeit einen On-Chip 3-State-Bus zu realisieren. Außerdem gibt es 4 globale primäre Taktnetze, von denen jedes alle CLB's und IO-Blöcke treiben kann. Sie sind mit fest zugeordneten Eingangspins verbunden. Das sekundäre globale Netz ist nicht an bestimmte Pins gebunden. Es besteht aus 24 Leitungen, 12 oben und 12 unten im Chip, die zusammen mit den langen Leitungen auch alle CLB's erreichen können.

3 Fail-safe-Verhalten

In diesem Kapitel wird auf das Fail-safe-Verhalten einer Schnittstelle eingegangen. Der erste Teil befaßt sich mit der Begriffseinordnung und der Bedeutung des Themas. Anschließend werden mögliche Fehler bei Schnittstellen betrachtete und Maßnahmen zu ihrer Erkennung vorgestellt. Der Einsatz dieser Maßnahmen wird an einigen existierenden Schnittstellen vorgestellt. Als Abschluss wird die Integration von Fail-safe-Verhalten in Schnittstellen vorgestellt und am Modell des Interfaceblocks aus Abschnitt 2.1 beschrieben.

3.1 Einordnung und Bedeutung

Mit zunehmenden Fortschritt werden technische Systeme immer komplexer. Jedoch bringt eine höhere Komplexität auch meist eine größere Anfälligkeit für Fehler mit sich. Vor allem bei sicherheitskritischen Anwendungen, zum Beispiel in der Automobilelektronik, der Signalgebung und Kommunikation für die Eisenbahn, in Kernkraftwerken, bei der Steuerung von Flugzeugen, der Luftraumkontrolle, in der Medizin und für militärische Anwendungen, ist es deshalb wichtig, sich mit der Zuverlässigkeit solcher Systeme auseinanderzusetzen. Zuverlässigkeit ist nach DIN 40041 [iDuV90] Teil 1 die Gesamtheit derjenigen Eigenschaften einer Betrachtungseinheit, welche sich auf die Eignung zur Erfüllung gegebener Erfordernisse unter vorgegebenen Bedingungen für ein gegebenes Zeitintervall beziehen. Eine geringer Zuverlässigkeit erhöht die Wahrscheinlichkeit von Ausfällen. Ein Ausfall ist nach DIN 40041 Teil 3 das Aussetzen der Ausführung der festgelegten Aufgabe einer Betrachtungseinheit aufgrund einer in ihr selbst liegenden Ursache und im Rahmen der zulässigen Beanspruchung. Ein Fehler ist die Nichterfüllung vorgegebener Forderungen durch einen Merkmalswert (DIN 40041 Teil 3). Tritt ein Fehler auf, so findet eine unzulässige Abweichung eines Merkmals des Systems statt und es befindet sich somit in einem unzulässigen Zustand. Daraus folgt, dass ein Fehler auch ein Zustand des Systems ist.

Allgemein unterscheidet man bei Systemen verschiedene Arten von Betriebszuständen im Zusammenhang mit Fehlern. Tabelle 3.1 stellt eine Übersicht über die möglichen Zustände eines System und sein Verhalten im jeweiligen Zustand dar. Im Folgenden wird näher auf den Zustand Fail-safe eingegangen, zu dem auch Fail-stop-safe gehört.

Fail-safe bedeutet allgemein nach [Har03b] und [Gör89], ein System beim Auftreten eines Fehlers in einen sicheren Zustand zu bringen. Es wird nur

Systemzustand	**Verhalten des Systems**
go	System arbeitet sicher und korrekt
fail-operational	System arbeitet fehlertolerant ohne Leistungsverminderung
fail-soft	Systembetrieb ist sicher, aber Leistung ist vermindert
fail-safe, fail-stop-safe	nur Systemsicherheit gewährleistet, evtl. keine Systemleistung
fail-unsafe	unvorhersehbares Systemverhalten

Tabelle 3.1: Systemzustände und ihr Verhalten, nach [Gör89]

die Systemsicherheit gewährleistet, jedoch nicht zwingend die weitere Arbeit des Systems. Damit wird vermieden, dass ein unvorhersehbares Systemverhalten eintritt. Ein System wird in einen sicheren Zustand gebracht, wenn eine Fehlfunktion das System negativ beeinflußt und infolge der Beeinflussung Personenschäden, Materialschäden oder Zerstörungen am System entstehen können. Eine Fehlfunktion in einem System kann zum Beispiel durch den Ausfall eines Teilsystems, fehlerhafte Eingangssignale oder auch ausbleibende Signale verursacht werden.

Es gibt verschiedene Arten von sicheren Zuständen. Man kann das System einfach abschalten. Dies wird zum Beispiel bei den Bremsen eines Lkw gemacht. Die Bremsen werden durch Luftdruck gelöst und blockieren damit die Räder nicht. Falls der Luftdruck infolge eines Defektes abfällt, schließen sich die Bremsen und bringen den Lkw zum Stillstand.

Eine andere Möglichkeit wird bei Ampelanlagen eingesetzt. Sie besteht darin, ein Reservesystem zu aktivieren, welches die Steuerung übernimmt. An Ampelanlagen sind dies die zusätzlich angebrachten Verkehrsschilder, die die Vorfahrt regeln, wenn die Ampel nicht funktioniert.

Bei einem digitalen System könnten bei fehlerhaften Eingangssignalen von einer Steuerung undefinierte Zustände entstehen und somit ein unvorhergesehenes Verhalten auslösen. Zur Vermeidung dieses Problems können die Eingangssignale mit Prüfmechanismen versehen werden. Mit deren Hilfe ist eine Fehlererkennung auf den Eingangssignalen möglich. Fehlerhafte Signale werden nicht an das zu steuernde System weitergeleitet und somit undefinierte Zustände vermieden.

Der Einsatz von Fail-safe-Verhalten bietet sich bei Systemen an, bei denen der Einsatz redundanter Komponenten nicht möglich ist, sich nicht lohnt

oder zu teuer wäre. Sie bieten somit eine günstige Variante für Systeme, bei denen die Systemleistung im Fehlerfall nicht garantiert sein muss, sondern nur eine Vermeidung von undefinierten oder gefährlichen Zuständen gewährleistet werden soll. Für den Einsatz von Fail-safe-Verhalten ist allerdings eine gute Kenntnis des Systems notwendig. Dazu gehört, dass man weiß, welche Fehler auftreten können, wie die Anwesenheit und das Auftreten von Fehlern erkennbar ist und welche Maßnahmen im Fehlerfall zu treffen sind. Durch diese Erfordernisse ist Fail-safe-Verhalten anwendungsabhängig.

3.2 Arten von Fehlern bei Schnittstellen

Bei Fehlern unterscheidet man nach [Gör89] drei Arten. Die erste Art sind Entwurfsfehler. Sie entstehen vor der Inbetriebnahme eines Systems. Zu ihnen zählen Implementierungsfehler, Spezifikationsfehler und Dokumentationsfehler. Die zweite Gruppe sind die Herstellungsfehler. Sie entstehen bei der Fertigung eines Systems zum Beispiel durch Fehler in der Fertigungstechnologie. Die dritte Art von Fehlern sind Betriebsfehler. Sie treten erst in der Nutzungsphase eines Systems auf. Zu ihnen zählen zufällige physikalische Fehler, Verschleißfehler, störungsbedingte Fehler, Bedienfehler, Wartungsfehler und absichtliche Fehler. Die Entwurfsfehler und Herstellungsfehler müssen beim Entwurfs- und Fertigungsprozess analysiert und vermieden werden. Hier ist nur ein System in der Nutzungsphase von Interesse, da Fail-safe-Verhalten zur Verhinderung von Fehlern dieser Phase eingesetzt werden.

Um Fail-safe-Verhalten vernünftig einsetzen zu können, ist zunächst eine Untersuchung der Fehler anzustellen, die auftreten können. Das Untersuchungsfeld wird sich hier auf Schnittstellen und die damit verbundene Datenübertragung zwischen zwei Systemen beschränken.

Bei der Datenübertragung müssen nach DIN EN 61508 VDE [iDuV02] Übertragungsfehler, Wiederholung, Verlust, Einfügung, falsche Abfolge, Nachrichtenverfälschung, zeitliche Verzögerung und Maskierung als mögliche Fehler angenommen werden. Übertragungsfehler sind Fehler, die während der Übertragung einer Nachricht auftreten und zum Beispiel durch elektromagnetische Einflüsse hervorgerufen werden. Eine Wiederholung liegt dann vor, wenn eine bereits gesendete Nachricht zu einem späteren Zeitpunkt fälschlicherweise wiederholt wird. Ein Verlust tritt ein, wenn eine Nachricht komplett gelöscht wird. Die Ursachen können beim Sender, der eine Nachricht nicht verschickt, beim Empfänger, der die Nachricht nicht annimmt oder im Nichtbestehen einer Verbindung liegen. Das Einfügen stellt eine nicht erlaubte Er-

weiterung der Daten einer Nachricht dar. Der Fehler der falschen Abfolge ereignet sich, wenn eine zeitlich ältere Nachricht nach einer neueren ankommt. Wird eine Nachricht verfälscht bevor sie mit einem Sicherungsmechanismus versehen oder geprüft worden ist, dann liegt eine Nachrichtenverfälschung vor. Eine zeitliche Verzögerung liegt vor, wenn die Nachricht zu einem bestimmten Zeitpunkt eintreffen muss, sie diesen aber verfehlt und später den Empfänger erreicht. Bei der Fehlermaskierung findet eine Verfälschung der Empfängeradresse statt. Dadurch erhält der falsche Empfänger die Nachricht.

3.3 Möglichkeiten der Fehlererkennung

Im vorigen Abschnitt wurden Arten von Fehlern vorgestellt, die bei der Datenübertragung auftreten können. Für den Einsatz von Fail-safe-Verhalten ist es jetzt notwendig, Möglichkeiten der Fehlererkennung zu betrachten. Dadurch kann man die Anwesenheit und das Auftreten von Fehlern erkennen und darauf entsprechend reagieren. Tabelle 3.2 fasst die Ergebnisse des Abschnittes zusammen und gibt eine Übersicht darüber, welcher Fehler durch einen bestimmten Fehlererkennungsmechanismus auffindbar sind.

3.3.1 Parität

Die Paritätsprüfung dient zur Identifizierung von Übertragungsfehlern. Dabei wird beim Sender zusätzlich zu den Daten ein Prüfbit generiert, das anzeigt, wieviele Einsen der Datensatz enthält. Die Parität kann gerade (Summe der Bits gerade) oder ungerade (Summe der Bits ungerade) sein. Mit diesem Verfahren ist keine Fehlerkorrektur möglich. Es lassen sich nur Fehler erkennen, die eine ungerade Bitanzahl betreffen (1, 3, 5, ... Bit-Fehler).

3.3.2 Rechteckcode

Beim Rechteckcode werden mehrere Codewörter zugleich betrachtet. Aus den Codewörtern wird eine Matrix gebildet und darin für jede Spalte und Zeile ein Prüfbit generiert. Außerdem wird aus den Prüfbits ein weiteres Bit gebildet, das Eckbit. Bei diesem Schema verändert jeder Einzelfehler zwei Prüfbits. Somit lassen sich mit dem Rechteckcode 1-Bit Fehler korrigieren. Er kann aber auch 2-Bit und gewisse 3-Bit Fehler erkennen. Mit ihm lassen sich Übertragungsfehler identifizieren und auch Fehler durch Einfügung, da mehrere Codewörter betrachtet werden.

Mechanismus \ Fehler	Übertragungsfehler	Wiederholung	Verlust	Einfügung	falsche Abfolge	Nachrichtenverfälschung	zeitliche Verzögerung	Maskierung
Parität	X							
Rechteckcode	X			X				
Hammingcodes	X			X				
Zyklische Codes	X			X				
Frame-Check	X			X				
ACK-Fehler-Erk.		X	X		X			
Monitoring	X							
PID-Check	X			X				
MEDL		X	X		X		X	

Tabelle 3.2: Fehler und ihre Erkennung durch verschiedene Mechanismen

3.3.3 Hammingcodes

Hammingcodes verfolgen die Idee, dass die Prüfbits die Position des Fehlers im Codewort angeben. Das Codewort wird gespreizt und Prüfbits eingefügt. Die Prüfbits werden zur Paritätskontrolle von bestimmten Bits benutzt. Bei der Kontrolle des Codewortes wird eine Binärzahl, deren Länge der Anzahl Prüfbits entspricht, gebildet. Sie gibt die Position eines 1-Bit Fehlers genau an. Mit diesem Verfahren lassen sich Übertragungsfehler erkennen. Auch manche Einfügungen sind identifizierbar, da durch sie die Position der Prüfbits verändert wird.

3.3.4 Zyklische Codes (CRC)

Der Cyclic Redundancy Check basiert auf dem Prinzip Bitfolgen als Darstellung eines Polynoms P(x) mit den Koeffizienten 0 und 1 zu betrachten. Zusätzlich benötigt man noch ein Generatorpolynom G(x) vom Grad g. Das

Polynom T(x), das übertragen wird, besteht aus der Nachricht M(x) und einer angehängten Prüfsumme. Das Polynom P(x) wird mit x^g multipliziert. Das Produkt wird durch das Generatorpolynom G(x) dividiert. Es entsteht das Quotientenpolynom Q und ein Divisionsrest R. An die ursprüngliche Nachricht wird der Divisionsrest R angefügt und diese gesendet.
Der Empfänger teilt die Nachricht dann wieder durch G(x) und erhält den Rest R. Stimmt der Rest nicht mit den letzten Stellen der erhaltenen Nachricht überein, dann ist die Nachricht falsch.

Es gibt verschiedene standardisierte Generatorpolynome für unterschiedliche Codelängen. Zum Beispiel das CRC-16 Generatorpolynom nach ISO lautet $x^{16} + x^{15} + x^2 + 1$. Es erkennt alle einfachen und zweifachen Bitfehler, alle Fehler mit ungerader Bitanzahl und alle Fehlerbündel mit 16 oder weniger Bits. Außerdem ist es in der Lage 99,997 Prozent aller 17-Bit Fehlerbündel und 99,998 Prozent aller Fehlerbündel mit 18 oder mehr Bits zu erkennen. Ein weiterer Vorzug des CRC ist, dass die Prüfsummen hardwaremäßig mit einer einfachen Schieberegisterschaltung berechnet und ausgewertet werden können.

Mit zyklischen Codes lassen sich Übertragungsfehler sowie Einfügungen erkennen. Mehr Informationen zu zyklischen Codes bieten [Tan03] und [Gör89].

3.3.5 Frame-Check

Der Frame-Check basiert auf der Struktur der übertragenen Nachricht und erkennt Fehler im Nachrichtenformat. Diese können durch Übertragungsfehler oder Einfügung auftreten. Zur Fehlererkennung wird der Rahmen der beim Empfänger eingehenden Nachricht überprüft. Die Kontrolle beschränkt sich dabei auf die Länge des Rahmens sowie der Korrektheit des Rahmenformates. Der Nutzdatenbereich einer Nachricht wird mit diesem Verfahren nicht kontrolliert.

3.3.6 ACK-Fehler Erkennung

Acknowledgement Fehler (Bestätigungsfehler) sind Fehler, die bei Protokollen mit Quittierungsmechanismus auftreten können. Auf ein eingehendes Paket reagiert der Empfänger, indem er eine Antwort an den Sender schickt. Er quittiert also, dass er das Paket erhalten hat. Erhält der Sender keine Antwort auf ein gesendetes Paket kann das folgende Ursachen haben. Das

Paket ist nicht beim Empfänger angekommen infolge von zum Beispiel Leitungsstörungen. Das Paket ist beim Empfänger angekommen, jedoch ist die Antwort auf dem Weg zum Sender verlorengegangen oder gestört worden, so dass die Antwort nicht korrekt ist. Eine weitere Ursache kann darin liegen, dass das Paket zwar beim Empfänger angekommen ist, jedoch durch andere Fehlererkennungsmaßnahmen als fehlerhaft identifiziert wurde. In diesem Fall gibt es zwei Möglichkeiten. Der Empfänger kann zum Einen dem Sender eine Fehlernachricht schicken, zum Anderen auch nichts tun und sich darauf verlassen, dass der Sender die Nachricht noch einmal schickt. Der ACK-Fehler muss immer vom Sender identifiziert werden. Im Fehlerfall schickt der Sender die Nachricht erneut auf den Weg. Hat der Empfänger diese aber schon bekommen, ist also das ACK-Paket verlorengegangen, dann schickt der Empfänger eine Quittung, ignoriert aber die neue Nachricht.

3.3.7 Monitoring

Das Monitoring beruht auf der Beobachtung der Buspegel durch den Sender. Der Sender kann Differenzen zwischen dem gesendeten und empfangenen Bit erkennen. Mit diesem Mechanismus ist es möglich alle globalen Fehler zu registrieren. Zusätzlich erlaubt es das Erkennen aller lokal am Sender auftretenden Bitfehler.

3.3.8 Packet Identifier Check

Der Packet Identifier Check (PID-Check) überprüft die Paketidentifizierung auf Korrektheit. Dabei wird beim Sender das Einerkomplement des PID mit der Länge n gebildet und an sie angehangen. Es wird ein PID der Länge 2·n übertragen. Der Empfänger bildet wieder das Einerkomplement der letzen n Bits und vergleicht sie mit den ersten n Bits. Stimmen diese nicht überein, dann ist eine Übertargunsfehler aufgetreten. Dieses Verfahren kann aber nur bei paketorientierten Übertragungsverfahren mit Identifikatoren für jedes Paket angewendet werden. [USB00]

3.3.9 Message Descriptor List (MEDL)

Eine Message Descriptor List (MEDL) ist ein Ein- und Ausgangsspeicher, der verschiedene Informationen enthalten kann. In ihr werden die Nutzdaten gespeichert, die gesendet oder empfangen werden sollen. Die Auslösung des

Sendevorgangs kann durch einen Timer zu einer bestimmten Zeit oder mit dem Schreiben der Daten in die MEDL erfolgen.

Der Empfänger der Nachricht wird nicht direkt angegeben. Dies erfolgt über Descriptoren. Sie werden vor den Nutzdaten übertragen. In der MEDL der Empfänger ist der Descriptor auch gespeichert. Stimmt der Descriptor auf der Leitung mit einem in der MEDL überein, so wird die Nachricht entgegengenommen ansonsten ignoriert.

Benutzt man ein bestimmtes Protokoll, dann ist die Struktur der Message Descriptor List vorgegeben und somit auch ihre Fehlererkennungsmechanismen. Entwirft man eine MEDL, dann ist es dem Entwickler überlassen, welche Mechanismen er einbaut. Dadurch ist es möglich, Fehlerkennungsmechanismen für Wiederholungen, Verluste, falsche Abfolge und zeitliche Verzögerung einzubauen.

3.4 Fehlererkennungsmechanismen verschiedener Schnittstellen

In diesem Abschnitt wird eine Auswahl von Protokollen hinsichtlich ihrer Fehlererkennungsmechanismen untersucht. Somit kann man für die Fehlererkennung auf bereits in den Protokollen existierende Verfahren zurückgreifen, ohne Eigenentwicklungen vornehmen zu müssen.

3.4.1 RS-232

RS-232 ist ein Protokoll zur seriellen point-to-point Übertragung von Daten. Auf dieser Übertragungsart sind verschieden aufgebaute Datenpakete definiert. Die übertragenen Rahmen bestehen immer aus einem Startbit, das den logischen Pegel 1 hat. Anschließend folgen 7 oder 8 Datenbits an die sich ein Paritätsbit anschließen kann. Den Abschluss des Rahmens bilden 1 oder 2 Stoppbits mit dem logischen Pegel 0.

Bei der RS-232-Schnittstelle kann ein Paritätsbit eingesetzt werden, um Übertragungsfehler zu erkennen. Die Schnittstelle unterstützt dabei gerade und ungerade Parität.

3.4.2 RS-485 Schnittstelle

Die RS-485-Schnittstelle ist ein serielles Bussystem. In der Spezifikation [Thi94] wird kein spezielles Protokoll definiert, sondern nur Übertragungsparameter. Das System basiert auf der Datenübertragung mittels eines Leitungpaares. Daran werden die bis zu 32 Endgeräte parallel angeschlossen. Die Daten werden durch Spannungsdifferenzen übermittelt. Durch die fehlende Protokolldefinition sind auch keine Fehlererkennungsmechanismen in der Spezifikation verzeichnet. Für die Fehlererkennung ist der Anwender selbst verantwortlich, da sich diese nach dem Protokoll richted, welches der Anwender implementiert.

3.4.3 Enhanced Parallel Port (EPP)

Das EPP Protokoll wurde für die parallele Übertragung mit der Centronics Schnittstelle entwickelt. Es ist ein Teil des IEEE 1284 Standards. Es erlaubt eine Datenrate von bis zu 2 Megabyte pro Sekunde. Dabei werden 8 Bit parallel gesendet. Die Synchronisation zwischen Sender und Empfänger erfolgt über Handshakesignale. Das EPP Protokoll definiert 4 Transferarten. Sie heißen Address-write, Data-write, Address-read und Data-read. Die Auswahl erfolgt über unterschiedliche Handshakeleitungen. Fehlererkennungsmechanismen sind bei keinem der Transferarten in der Spezifikation vorgesehen. Mehr zu den Transferarten und zur parallelen Schnittstelle steht in [Axe97] und in [EPP].

3.4.4 USB

USB (Universal Serial Bus) ist ein serielles Bussystem von dem die Versionen 1.0, 1.1 und 2.0 definiert sind. In der grundlegenden Arbeitsweise unterscheiden sie sich jedoch nicht, weshalb hier nicht auf die einzelnen Versionen eingegangen wird. Die Daten werden seriell, also Bit für Bit, übertragen. Das Kabel besitzt dazu zwei Datenleitungen. Mit diesen wird das Signal differentiell übertragen, das heißt der Wert des Signals ist die Spannungsdifferenz zwischen den Datenleitungen. Durch dieses Verfahren wird die Störanfälligkeit des Kabels verringert. Die Topologie ist eine gemischte Stern- und Strang-Bus-Technologie. Sie wird als Tiered-Star-Topologie bezeichnet. An einen Host können so bis zu 127 Endgeräte teilweise direkt oder über Hubs angeschlossen werden.

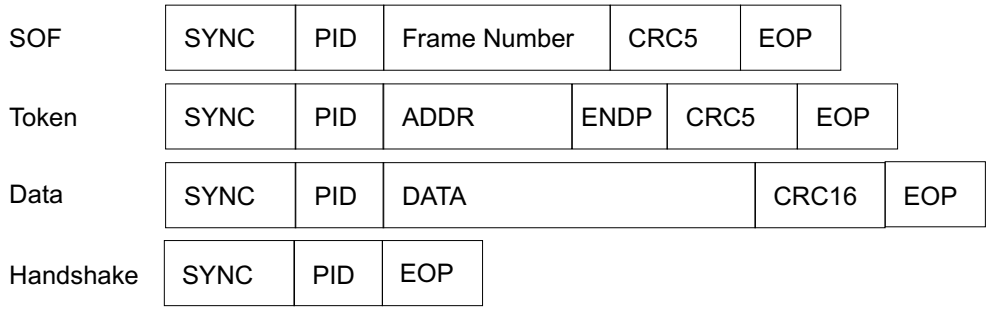


Abbildung 3.1: Aufbau der USB-Pakete

USB unterscheidet zwischen vier verschiedenen Paketen. Der Aufbau der einzelnen Pakete ist in Abbildung 3.1 veranschaulicht. Jedes Paket enthält immer ein SYNC-Feld, das zur Synchronisation dient. Des Weiteren ein PID (Packet Identifier) -Feld. Es gibt Auskunft über den Pakettyp, das Format des Paketes und den Typ der Fehlererkennung. Am Ende jedes Paketes befindet sich das EOP (End of Packet) -Feld, welches das Ende eines Paketes kennzeichnet. Als Synchronisationspaket dient das SOF (Start of Frame) -Paket. Es enthält die Nummer des nächsten Frames und eine 5-Bit Prüfsumme. Mit dem Token-Paket bestimmt der Host, welches Gerät als nächstes das Senderecht erhält. Es enthält die Adresse des Gerätes, das Endpoint-Feld, womit eine Adressierung einer Funktion des Gerätes möglich ist, und eine 5-Bit Prüfsumme. Das Data-Paket enthält die eigentlichen Nutzdaten und eine 16-Bit Prüfsumme. Zur Bestätigung eines Datenpaketes wird das Handshake-Paket eingesetzt.

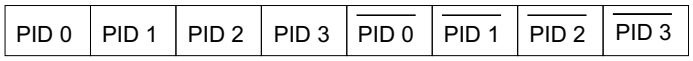


Abbildung 3.2: Format eines USB PID, Quelle [USB98]

Die Verfahren zur Fehlererkennung sind bei USB 1.0, 1.1 und 2.0 gleich. Zur Fehlererkennung wird der Cyclic Redundancy Check (CRC) eingesetzt. USB unterstützt zwei Arten des CRC. Für die SOF- und Token-Pakete wird CRC5 mit dem Generatorpolynom $G(x) = x^5 + x^2 + 1$ eingesetzt. Die Datenpakete werden durch CRC16 geschützt. Es verwendet das Generatorpolynom

$G(x) = x^{16} + x^{15} + x^2 + 1$. Der CRC schützt alle Felder eines Paketes außer dem PID-Feld.
Das PID-Feld hat einen eigenen Schutzmechanismus. Von dem 4-Bit langen PID wird das Einerkomplement gebildet und an das Original angehängt. So ergibt sich die in Abbildung 3.2 gezeigte Struktur des PID. Wird eine fehlerhafte PID von einem Empfänger erkannt, dann ignoriert sie dieser.
Der dritte Fehlererkennungsmechanismus bei USB richtet sich gegen Acknowledge-Fehler. Zu diesem Zweck werden die Handshake-Pakete eingesetzt. Sie übermitteln den Status einer Datenübertragung und können Informationen über einen erfolgreichen Datenempfang, über eine Kommandoakzeptanz oder -zurückweisung, zur Flusskontrolle und zur Unterbrechungsanforderung enthalten.

Weiterführende Informationen zu USB sind in [USB98] und [USB00] zu finden.

3.4.5 Firewire

Firewire ist ein serielles Hochgeschwindigkeitsprotokoll. Es kann aus bis zu 1023 Bussegmenten mit je 63 Geräten bestehen. Die Kommunikation zwischen den Geräten kann über Nachrichten (isochroner Modus) oder als verteilter Speicher (asynchroner Modus) erfolgen. Dabei können die Geräte direkt miteinander kommunizieren und müssen nicht über den Host Daten austauschen, wie bei USB.

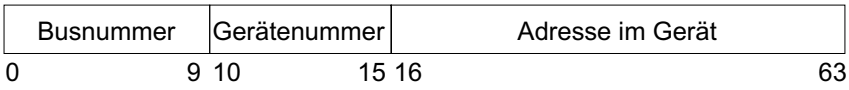


Abbildung 3.3: Aufbau der Adresse von Firewire

Die angesteuerte Adresse setzt sich aus 10 Bit für das Bussegment, 6 Bit für die Geräte (Nodes) im Busegment und 32 Bit für die Adresse im Gerät zusammen (Abbildung 3.3). Die Datenübertragung ist in Runden organisiert. Die Dauer einer Runde beträgt 125 μs. Den Start einer Runde signalisiert ein Cycle-Start Paket. Auf dieses Paket folgen dann isochrone Pakete. Diese Pakete dürfen maximal 80% einer Runde einnehmen. Anschließend können asynchrone Pakete verschickt werden bis zum Beginn der nächsten Runde.

Zur Feherlerkennung verwendet Firewire einen 32 Bit CRC. Dieser Mechanismus kommt sowohl bei isochroner als auch bei asynchroner Datenübertragung zum Einsatz. Im asynchronen Übertragungsmodus kommen, zusätzlich zum CRC, Acknowledgementpakete zum Einsatz. Das heißt, der Sender erwartet für eine verschickte Nachricht eine Bestätigung vom Empfänger über deren Erhalt. Der isochrone Modus unterstützt dies nicht, da bei ihm ein vorhersagbares Zeitverhalten im Vordergrund steht.

3.4.6 TTP/C

TTP/C [Kop97, AG02, TTP] ist ein Protokoll zur Echtzeitkommunikation zwischen elektronischen Komponenten von verteilten fehlertoleranten Systemen, die über einen Bus verbunden sind. Die einzelnen Knoten werden als Nodes bezeichnet. Ein Node (Abbildung 3.4) besteht aus einem TTP/C-Controller und einem Hostcomputer. Die Kommunikation zwischen Controller und Host findet über das Communication Network Interface (CNI) statt. Der Controller enthält einen Bus-Guardian, den Protokollprozessor und die TTP/C Message Descriptor List (MEDL).

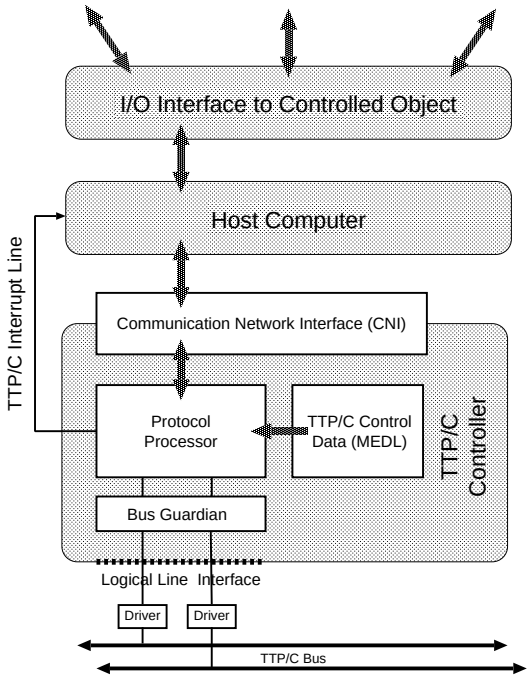


Abbildung 3.4: Aufbau einer TTP/C Node, Quelle [AG02]

		Attribute			
Zeit	Adresse	D	L	I	A
t1	0x0001	Out	8	I	0
t2	0x00A1	In	8	N	0
t3	0x003C	Out	16	N	0

Abbildung 3.5: Aufbau der TTP/C Message Descriptor List, Quelle [Kop97]

Der Bus-Guardian ist eine unabhängige Einheit, die den Bus vor Timingfehlern durch den Controller schützt. Der Bus besteht aus zwei identischen Kanälen. Beim Senden wird die Nachricht über jeden der Kanäle geschickt. Die MEDL ist eine statische Datenstruktur im TTP/C-Controller (Abbildung 3.5). Sie enthält den Zeitpunkt, wann eine Nachricht, deren Adresse im Adressfeld steht, gesendet oder empfangen werden muss. Die Adresse gibt an, wo die Nachricht im CNI steht. Die Attribute in der MEDL geben die Richtung (D - in/out), die Länge der Nachricht (L), die Art der Nachricht (I - Initialisierung / Normal) und zusätzliche Parameter (A) für Modus und Rollenwechsel an.

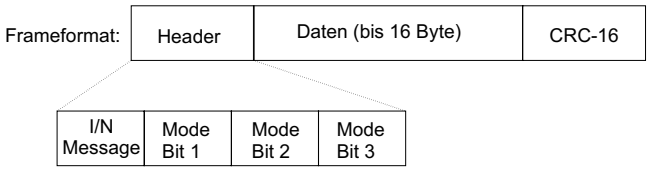


Abbildung 3.6: TTP/C Frameformat, Quelle [Kop97]

Der Frame (Abbildung 3.6), der gesendet wird, besteht aus einem Header, dem Nutzdatenfeld und einem 16 Bit CRC. Es gibt grundsätzlich zwei Nachrichtenarten. Initialisierungsnachrichten und normale Nachrichten. Bei Initialisierungsnachrichten wird der CRC über den Header und die Nutzdaten gebildet. Die CRC-Prüfsumme normaler Nachrichten setzt sich aber nicht nur aus dem Header und den Nutzdaten zusammen (Abbildung 3.7). Um eine Übereinstimmung der Controllerzustände zu bewirken, ohne dafür extra Nachrichten versenden zu müssen, bildet der Sender den CRC, zusätzlichen zu Header und Nutzdaten, auch aus dem aktuellen Controllerzustand (C-State). Der C-State enthält informationen über die aktuelle Zeit im Sender, die aktuelle Position in der MEDL, den aktuellen Modus, anstehende Moduswechsel und Zugehörigkeit. Zur Überprüfung der eingehenden Nach-

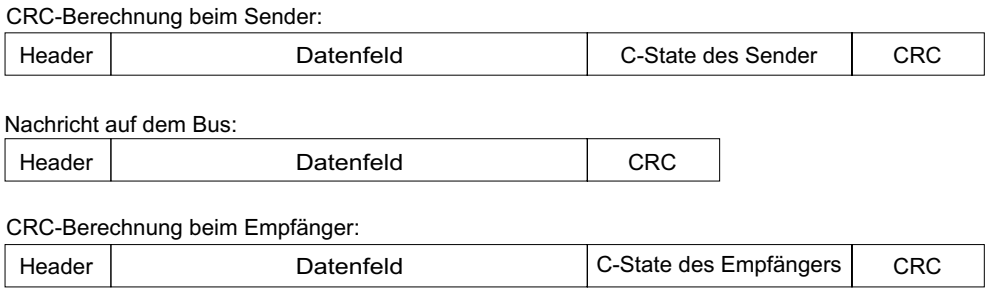


Abbildung 3.7: Berechnung des CRC bei TTP/C, Quelle [Kop97]

richt, bildet der Empfänger den CRC aus den empfangen Header und Nutzdaten und seinem eigenen C-State. Die Clock-Synchronisation findet auch ohne zusätzliche Nachrichten statt. Da der Zeitpunkt, zu dem ein Node eine bestimmt Nachricht sendet, Sender und Empfänger bekannt sind, wird darüber die Zeit synchron gehalten. Damit können Übertragungsfehler, Wiederholung, Verlust, Einfügungen, falsche Abfolge, zeitliche Verzögerung und Maskierung erkannt werden.

3.4.7 TTP/A

TTP/A ist eine eingeschränkte Version des TTP/C Protokolls. Es ist kein verteiltes sondern ein Multimasterprotokoll. Es wurde für günstige Feldbusanwendungen entwickelt. Zur Realisierung kann ein Standard-UART (Universal Asynchronous Receiver Transmitter) verwendet werden. Eine Nachricht besteht dort aus einem Startbit, 8 Bit Nutzdaten, einem Paritätsbit und einem Stopbit.

Bei TTPA/A gibt es zwei Nachrichtenarten, die Fireworks- und die Datennachricht. Zur Kennzeichnung dient bei Fireworksnachrichten ungerade Parität, bei Datennachrichten gerade Parität. Die Datenübertragung ist in Runden aufgeteilt. Jede Runde beginnt mit dem Senden einer Fireworksnachricht durch den aktiven Master. Die Nachricht enthält den Namen der aktiven Message Descriptor List (MEDL) für diese Runde. Außerdem dient sie zur globalen Synchronisation der Nodes. Anschließend folgt eine Sequenz von Datennachrichten, bis die aktive MEDL abgearbeitet ist. Danach kann eine neue Runde starten. Das Senden einer Nachricht muss zu einem vorbestimmten Zeitpunkt erfolgen und in einem bestimmten Zeitfenster abgeschlossen

sein. Diese Informationen müssen bei der Entwicklung der MEDL festgelegt werden.

Um einen UART für TTP/A einsetzen zu können, muss dieser gerade und ungerade Parität unterstützen. Gerade Parität dient zur Kennzeichnung und zur Fehlererkennung in Datennachrichten. Tritt ein Fehler in den Nutzdaten auf, bleiben die alten Daten erhalten und der Fehler wird innerhalb der Node dem Host mitgeteilt.
Zur Erkennung von Fehlern im zeitlichen Ablauf, dient die MEDL. Die Sequenz von Sende und Empfangsvorgängen ist in der MEDL festgelegt und wird von allen Nodes beobachtet. Wird ein RDI (receive data interrupt, Daten wurden empfangen) außerhalb des zulässigen Zeitfensters ausgelöst, liegt ein Kontrollfehler vor. Bleibt eine im Zeitfenster erwartete Nachricht aus, dann werden die alten Daten nicht verändert und der Fehler zum Host gemeldet. Beim Auftreten eines Kontrollfehlers in einer Node, beendet diese für sich die aktuelle Runde. Die Node wartet dann auf die nächste Fireworksnachricht. Sendet der aktive Master nicht innerhalb eines bestimmten Timeouts, dann wird ein Reservemaster aktiviert.

Mit den Fehlererkennungsmechanismen lassen sich Nachrichtenwiederholung, Verlust, falsche Abfolge, zeitliche Verzögerung und Maskierung erkennen. Auch fehlerhafte Daten, bei denen eine ungerade Anzahl von Bits verfälscht ist, lassen sich erkennen. Mehr Informationen zum TTP/A Protokoll beinthalten [Kop97], [AG01] und [TTP].

3.4.8 LVDS

LVDS (Low Voltage Differential Signaling) ist eine Technologie die im IEEE Standard 1596.3 definiert ist. Sie erlaubt es, Signale mit einem geringen Spannungsunterschied (250 ... 400 mV) zwischen Low und High-Pegel zu übertragen. Durch den geringen Spannungshub sind hohe Datenübertragugsraten bei geringem Leistungsverbrauch möglich. Die Spezifikation [Sem00, LVD] umfaßt nur die Definition der elektrischen Eigenschaften auf Bitübertragungsebene. Ein festes Protokoll zur Datenübertragung wird, wie bei RS-485, nicht definiert.

In den meisten LVDS Empfängern sind interne Failsafe-Schaltungen eingebaut. Sie zwingen den Ausgang unter gewissen Fehlerbedingungen in einen bekannten logischen Pegel, so dass keine undefinierten Ausgangsspannungen auftreten. Diese Failsafe-Mechanismen treten bei offenen Eingangspins, unterbrochenen Eingängen und unzureichender Energiezufuhr in Kraft.

3.4.9 Controller Area Network (CAN)

Das Controller Area Network (CAN) bezeichnet einen seriellen Bus, mit einer Datenübertragungsrate von 1 Mbit/s bei maximal Leitungslänge von 40 m. Größere Distanzen erfordern geringere Übertragunsraten. Bei der Datenübertragung findet keine Adressierung statt. Jeder Knoten am Bus kann anhand eines Identifiers, der in einer Message Descriptor List szeht, entscheiden, ob er die Nachricht annimmt oder nicht. Bei CAN 1.0 können pro Nachricht 8 Byte Nutzdaten verschickt werden. Der Identifier ist 11 Bit lang. Es gibt 4 Arten von Paketen, die beim CAN Frame genannt werden. Es sind der Daten-, Remote-, Error- und Overload-Frame. Die CAN 2.0-Spezifikation verwendet einen 29 Bit langen Identifier. Damit ist eine größere Anzahl Geräte ansprechbar. Außerdem wurde die Nutzlast der Datenpakete auf bis zu 64 Byte erhöht.

Die Fehlererkennungsmechanismen sind bei beiden Spezifikationen gleich. Das CAN benutzt zur Fehlererkennung CRC16. Dabei wird aus dem Start-, Arbitration-, Control- und Datenfeld die Prüfsumme gebildet. Das Generatorpolynom lautet $G(x) = x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$. Der

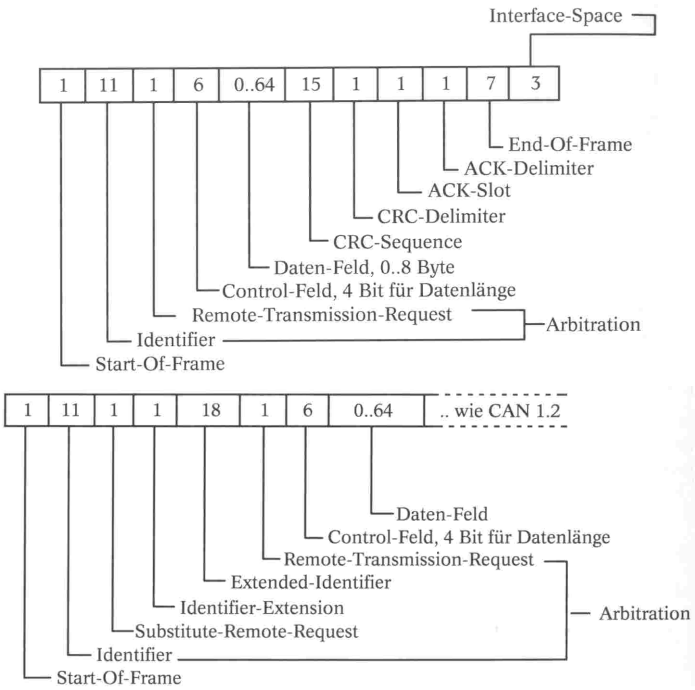


Abbildung 3.8: Datenrahmen für CAN 1.0 und 2.0, Quelle [Dem93]

Sender überträgt im ACK-Bit ein High. Stellt ein Empfänger eine Übereinstimmung mit der CRC-Sequenz fest, überschreibt er das Bit mit Low. In diesem Fall erfolgte die Übertragung ohne Fehler. Falls die Übertragungs nicht fehlerfrei verlief, dann sendet ein Busteilnehmer ein Error-Frame. Es wird von den anderen Einheiten erkannt, worauf diese die letzten empfangenen Daten verwerfen. Die Busteilnehmer besitzen je zwei Zähler. Dabei ist einer für das Senden und einer für das Empfangen zuständig. Mit diesen Zählern ist es möglich defekte Busteilnehmer zu erkennen. Bei einem Fehler wird der Zähler um eins erhöht, bei einer erfolgreichen Übertragungs dagegen wieder um eins verringert. Übersteigt ein Zähler den Wert von 128 dann wird der entsprechende Busteilnehmer in den „Error passiv "Staus gebracht. Dadurch ist es ihm nur noch möglich zu senden und empfangen, wenn kein anderer Busteilnehmer den Bus benutzt. Treten weitere Fehler auf, schaltet sich der Knoten bei einem Zählerstand von 256 vom Bus ab. Weiterführende Informationen zu diesem Standard stehen in [CANa], [CANb] und [Dem93].

3.4.10 Ethernet

Ethernet [Hei95, San] ist ein weit verbreitetes Netzwerkprotkoll. Sein Einsatzspektrum reicht von der Verbindung zweier PC's bis hin zum Aufbau komplexer Netzwerksysteme. Dabei können als Übertragungsmedium für die Daten Kabel, Glasfaserkabel oder Funk eingesetzt werden. Die Topologie kann als Token Ring oder als Bus, an dem mehrere Geräte angeschlossen sind, aufgebaut sein.

Ethernet verwendet zum Zugriff auf das Übertragungsmedium das CSMA/CD Zugriffsverfahren (Carrier Sense Multiple Access/Collision Detection). Vor dem Senden prüft die Station ob schon eine Nachricht einer anderen Station auf dem Bus liegt. Ist das nicht der Fall dann legt sie ihre Nachricht auf den Bus. Haben allerdings mehrere Stationen gleichzeitig geprüft und versuchen gleichzeitig Daten auf den Bus zu legen, dann kommt es zu einer Kollision. Diese muss von den einzelnen Stationen erkannt werden. Daraufhin ziehen die Stationen ihre Signale zurück und versuchen es zeitversetzt mit einem Fairnessintervall noch einmal.

Zur Fehlererkennung benutzt der Ethernet-Standard den Cyclic Redundancy Check. Dabei wird CRC32 mit dem Generatorpolynom $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ eingesetzt. Das Erzeugen sowie das Auswerten des CRC-Wertes erfolgt in der Sicherungsschicht.

3.5 Integration von Fail-safe-Verhalten in Schnittstellen

Systeme mit harten oder weichen Echtzeitbedingungen verlangen ein vorhersagbares Systemverhalten. Die gilt sowohl für den fehlerfreien Betrieb, als auch für den Betrieb im Fehlerfall. Um dies zu erreichen wird Fail-safe-Verhalten eingesetzt.

Bisher wurde Fail-safe-Verhalten in die Systeme direkt integriert. Der hier vorgestellte Ansatz besteht darin, diesen Mechanismus schon in die verbindende Schnittstelle zweier Systeme einzubinden. Ein großer Vorteil dieses Ansatzes ist, dass an den Systemen selbst keine Änderungen vorgenommen werden müssen. Der Einsatz von verschiedenen Intellectual Properties (IP) wird somit vereinfacht. Der Entwurf von Systemen aus IP's beschränkt sich somit auf die Entwicklung der Schnittstellen zwischen den Systemen. Änderungen an den Komponenten selbst sind dadurch nicht notwendig. Um gegen eventuell auftetende Fehler bei der Kommunikation gewappnet zu sein, wird Fail-safe-Verhalten in die Schnittstelle integriert.

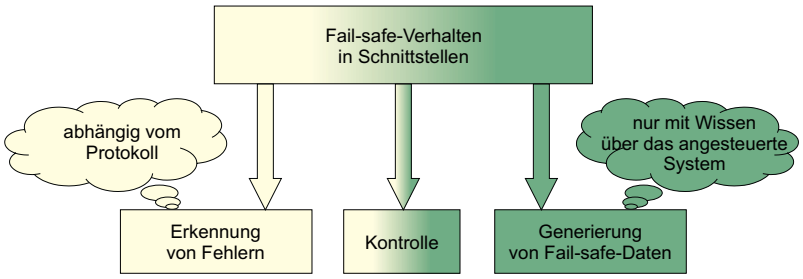


Abbildung 3.9: Aufgaben zur Erfüllung von Fail-safe-Verhalten in Schnittstellen

Zur Realisierung von Fail-safe-Verhalten sind allgemein drei Aufgaben (Abbildung 3.9) zu erfüllen. Das sind die Fehlererkennung, die Generierung von Fail-safe-Daten und die Überwachung und Kontrolle der Erkennung und Generierung.

Die Aufgabe der Fehlererkennung realisiert der **Protokollguard**. Durch ihn werden Mechanismen zur Fehlererkennung, wie sie in Abschnitt 3.3 vorgestellt wurden, in die Schnittstelle integriert. Der Protokollguard überprüft das eingehende Protokoll auf Korrektheit. Dabei ist die Art der Fehlererkennung abhängig vom Protokoll, das empfangen wird, wie Abschnitt 3.4 gezeigt

hat. Im Fehlerfall generiert der Protokollguard ein Fehlersignal, das von einer Kontrolleinheit interpretiert werden kann. Es ist durchaus vorstellbar, dass der Protokollguard verschiedene Fehlersituationen erkennen kann und für jede Fehlersituation ein individuelles Fehlersignal erzeugt. Somit kann die Kontrolleinheit auf unterschiedliche Fehlersituationen angemessen reagieren.

Der **Generator für die Fail-safe-Daten** erzeugt im Fehlerfall Daten, um ein System in einen fehlersicheren Zustand zu bringen. Um Fail-safe-Daten für ein System erzeugen zu können, muss man wissen, durch welche Daten ein System in einen fehlersicheren Zustand gebracht werden kann. Es ist also Wissen über das angesteuerte System notwendig. Dieses Wissen wird in den Generator bei der Entwicklung der Schnittstelle mit eingebunden. Der Generator ist in der Lage eine festgelegte Datensequenz zu erzeugen oder auch eine komplexe Steuerung zu realisieren. Dadurch ist es möglich Fail-safe-Datengeneratoren für ein breites Anwendungsspektrum einzusetzen.

Zur Überwachung und Kontrolle der Erkennung und Generierung dient eine **Kontrolleinheit**. Sie hat die Aufgabe, die Fehlersignale auszuwerten und entsprechende Maßnahmen zu treffen. Die Maßnahme besteht darin, einen Generator für Fail-safe-Daten zu aktivieren. Dabei kann die Kontrolleinheit durchaus auf mehr als einen Generator zurückgreifen. Dies ermöglicht eine individuelle Reaktion auf Fehlersituationen. Die Einbindung der Kontrolleinheit in das Fail-safe-Verhalten bringt den Vorteil mit sich, dass der Protokollguard und der Fail-safe-Datengenerator unabhängig voneinander entwickelt werden können und zwei eigenständige Komponenten bilden.

Ein Interfaceblock, wie er in Kapitel 2.1 vorgestellt wurde, erfüllt alle Voraussetzungen zur Integration von Fail-safe-Verhalten in Schnittstellen. Im Modell des Interfaceblocks gibt es vier Komponenten. Die Kontrolleinheit, zwei Protokollhandler, wobei einer zum Empfangen und einer zum Senden benötigt wird (im folgenden als PH_{in} und PH_{out} bezeichnet) und den Sequenzhandler. Im Folgenden wird beschrieben, wie sich die bisherigen Konzepte in den Interfaceblock integrieren lassen. Zur Veranschaulichung zeigt Abbildung 3.10 ein mögliches Fail-safe-Verhalten in einem Interfaceblock.

Die Feherlerkennung sollte so nah wie möglich an der Datenquelle sitzen, damit die Verzögerungszeiten zwischen dem Auftreten eines Fehlers und dessen Erkennung so gering wie möglich sind. Aus diesem Grund bewerkstelligt die Fehlererkennung der aktive Modus des PH_{in}. Innerhalb des Interfaceblocks ist er der Erste, der die Daten erhält. Außerdem ist in diesem Modus der Kommunikationsautomat zur Erkennung das aktuell empfangenen Protokolls implementiert, der zum Empfangen von Daten nötig ist. Ein weiterer

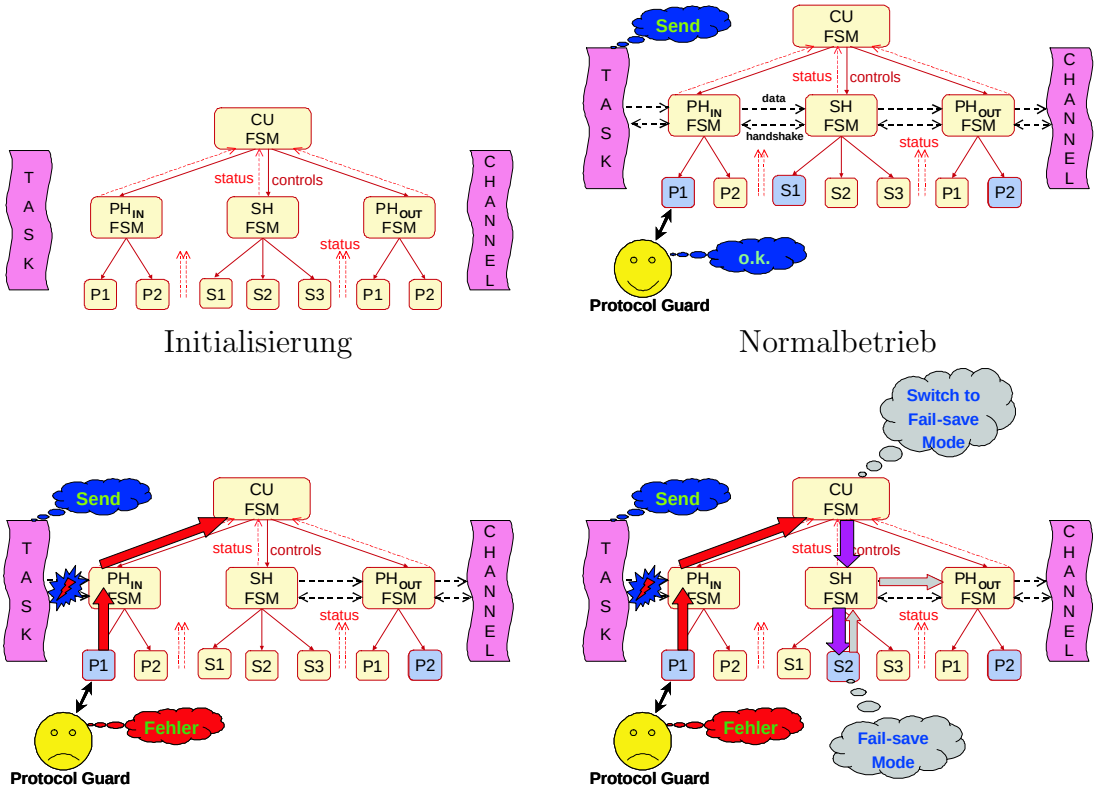


Abbildung 3.10: Fail-safe-Verhalten in einem Interfaceblock

Vorteil ist, dass man die Erkennung von Übertragungsfehlern parallel zum Datenempfang stattfinden lassen kann, wenn man die Fehlererkennung in den Modus implementiert. Dabei kann man auf die protokollspezifischen Fehlererkennungsmechanismen der Protokolle (Abschnitt 3.4) zurückgreifen. Es ist aber zu beachten, dass beim Auftreten eines Fehlers die Daten nicht an den Sequenzhandler weitergereicht werden und ein Statussignal für die Kontrolleinheit generiert wird, dass diese über den Fehler informiert.

Zur Erkennung von Zeitüberschreitungsfehlern ist ein Timer notwendig. Dabei gibt es zwei verschiedene Implementierungsvarianten. Falls das eingehende Protokoll schon Echtzeitanforderungen unterliegt, ist es sinnvoll den Timer in den Modus zu integrieren. Damit ist man bezüglich verschiedener Protokolle flexibel, die unterschiedlichen Echtzeitanforderungen unterliegen. Wenn der Interfaceblock allerdings zur Umwandlung eines ereignisgesteuerten (event-triggered) Protokolls in ein zeitgesteuertes (time-triggered) benutzt wird, muß der Timer in die Kontrolleinheit integriert werden. Zusätz-

lich muss der Modus im PH_{in} ein Statussignal generieren für den Fall, dass er Daten empfangen hat. Liegt der Zeitpunkt innerhalb der vorgeschriebenen Zeit, dann können die Daten normal weitergegeben werden. Wurde allerdings das Ende des Timers erreicht und sind keine neuen Daten eingegangen, dann muss der Fail-safe-Datengenerator aktiviert werden, um Daten für den nächsten Sendezyklus zu generieren.

Der Generator der Fail-safe-Daten wird als Modus des Sequenzhandlers implementiert. Bei der Implementierung ist ein genaues Wissen über die Komponente notwendig, für die die Fail-safe-Daten generiert werden sollen. Zum Beispiel ein reines Weiterschicken von alten Daten könnte zu unbeabsichtigten Ergebnissen führen. Das wäre unter anderem der Fall, wenn relative Positionierungsdaten für einen Roboter gesendet werden. Die alten Daten enthalten Differenzwerte zur aktuellen Position. Werden diese geschickt, würde sich der Roboter weiterbewegen, was vom Anwender aber nicht beabsichtigt ist. Deshalb muss der Fail-safe-Datengenerator in solch einem Fall einen Nullvektor erzeugen. Aus diesem Grund ist vor dem Einsatz von Fail-safe-Verhalten zu untersuchen, welche Daten eine Komponente in einen sicheren Zustand bringen und dieses Wissen in den Modus zu implementieren. Im Interfaceblock ist es auch möglich, auf unterschiedliche Fehler mit verschiedenem Fail-safe-Verhalten zu reagieren. Wenn die Fehlererkennung mehrere Fehler unterscheiden kann oder es mehrere Fehlerquellen gibt, kann für jede Fehlersituation ein anderer Modus für den Sequenzhandler implementiert werden. Je nach Fehlersituation wird der entsprechende Modus aktiviert.

Zur Kontrolle und Überwachung der Fehlererkennung und der Generierung von Fail-safe-Daten ist die Kontrolleinheit zuständig. In ihr laufen alle Statusleitungen der Handler und ihrer Modi zusammen. Wenn sie vom Protokollguard eines PH_{in}-Modus einen Fehler gemeldet bekommt, dann kann sie sofort reagieren und im Sequenzhandler den entsprechenden Fail-safe-Modus aktivieren. Wenn dann in einem weiteren Durchlauf des PH_{in}-Modus kein Fehler mehr festgestellt wird, dann schaltet die Kontrolleinheit den Sequenzhandler wieder auf Normalbetrieb.

Der Einsatz von Fail-safe-Verhalten in einem IFB ist auch bei dynamisch rekonfigurierbaren FPGA's denkbar, bei denen in der Rekonfigurationsphase das FPGA nicht angehalten wird. Dazu muss die Kontrolleinheit über den Beginn der Rekonfigurationsphase informiert werden. Sie aktiviert dann einen Fail-safe-Modus für die Dauer der Rekonfiguration. Nach dem Abschluss der Phase schaltet die Kontrolleinheit den IFB wieder auf Normalbetrieb bzw. in einen Sequenzhandlermodus der der neuen Konfiguration entspricht.

Das Einsatzfeld von Fail-safe-Verhalten in einem Interfaceblock ist groß. Es reicht von einfachen Kommunikationsaufgaben über die Umwandlung von ereignisgesteuerten in zeitgesteuerte Signale für Echtzeitanforderungen bis hin zu komplexen Systemen, die während des Betriebs an neue Funktionen angepasst werden können. Der IFB bietet dabei eine gute Grundlage, um unterschiedlichsten Anforderungen gerecht zu werden.

4 Demonstrator

In diesem Kapitel wird ein Demonstrator vorgestellt, an dem die Konzepte, die in dieser Arbeit vorgestellt wurden, umgesetzt werden. Zunächst erfolgt eine kurze Betrachtung über den Nutzen eines Demonstrators. Anschließend wird dessen Aufbau und Funktionsweise erläutert. Zum Schluss werden die Konzepte zur Implementierung beschrieben und umgesetzt.

4.1 Bedeutung

Ein Demonstrator bietet den Vorteil, erarbeitete theoretische Konzepte im praktischen Einsatz zu erproben und zu validieren. Dadurch können Schwachstellen und Fehler im theoretischen Modell aufgedeckt und korrigiert werden. Außerdem kann die Praxisrelevanz der Konzepte gezeigt werden. Liegt nur ein theoretisches Modell vor, so kann es durchaus sein, dass es sich nicht oder nur schwer in die Praxis umsetzen lässt und damit nie zum Einsatz kommen kann. Umgekehrt kann eine rein praktische Implementierung eventuell nicht allgemein modelliert werden und ist damit auf ein spezielles Anwendungsgebiet beschränkt. Deshalb werden für den Demonstrator zunächst die Modellierungsansätze vorgestellt, bevor die Implementierung erfolgt.

4.2 Aufbau und Funktionsweise

Der Demonstrator (Abbildung 4.1) ist aus zwei PC's, einem Roboterarm und einem Digilab 2E FPGA Board mit einem Xilinx Spartan 2E XC2S200E FPGA [Inc02] aufgebaut. Die Verbindung zwischen PC1 und FPGA-Board wird über die serielle Schnittstelle hergestellt. Die Verbindung von FPGA-Board und PC2 geschieht über die parallel Schnittstelle.

Auf PC1 werden Daten zur Steuerung des Roboters eingegeben, deshalb wird im Folgenden PC1 als Steuerung bezeichnet. Auf PC2 läuft die Roboterkontrolle. Der Roboter verfügt über 4 Freiheitsgrade, die gesteuert werden können. Je 2 Bit sind für die Steuerung eines Freiheitsgrades zuständig. Ein Freiheitsgrad kann beibehalten, um eine Einheit verringert oder vergrößert werden. Zusätzlich kann eine durch die Roboterkontrolle vorgegebene Ausgangsposition angefahren werden. Abbildung 4.2 veranschaulicht die Freiheitsgrade des Roboters und stellt dar, durch welche Daten diese gesteuert werden können.

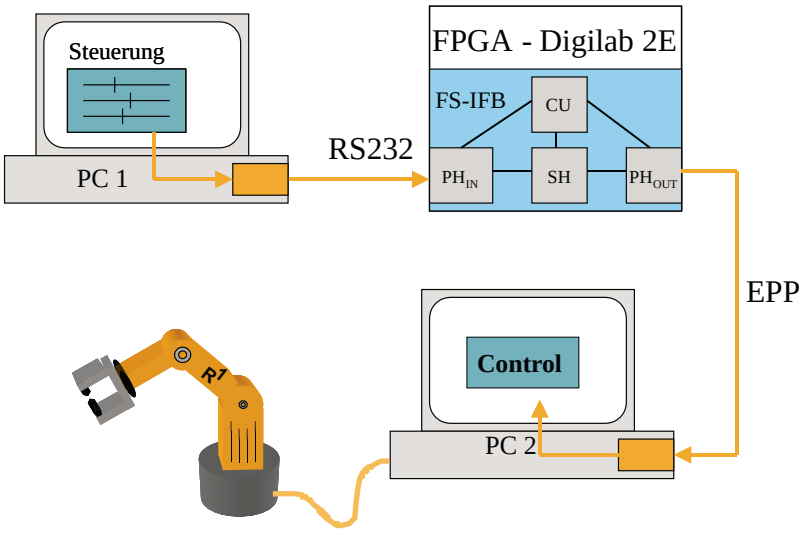
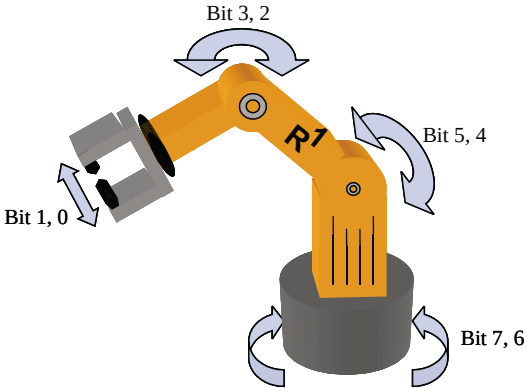


Abbildung 4.1: Der Aufbau des Demonstrators

Die Kommunikation zwischen der Steuerung und der Roboterkontrolle findet über einen Fail-safe-Interfaceblock (FS-IFB) statt. PC1 schickt Steuerdaten über die serielle RS-232-Schnittstelle. Die Nutzdaten bestehen aus 8 Datenbits und einem Paritätsbit für ungerade Parität. Der FS-IFB empfängt die Daten und wertet das Paritätsbit aus. Sind die Daten fehlerfrei, speichert sie der FS-IFB zwischen. Die Roboterkontrolle liest einmal je Sekunde die Steuerdaten aus dem FS-IFB mit Hilfe des EPP-Protokoll.



Wert	Funktion
00	Keine Veränderung
01	Freiheitsgrad plus eine Einheit
10	Freiheitsgrad minus eine Einheit
11	Zurück auf Ausgangsposition

Abbildung 4.2: Funktionen des Funktionen

Aus diesem Aufbau ergibt sich die Liste der Komponenten und ihrer Funktionen, die zur Implementierung des Demonstrators notwendig sind:

- PH_{in} im FPGA, Empfang von seriellen Daten über RS-232-Schnittstelle
- PH_{out} im FPGA, Senden von parallelen Daten über parallele Schnittstelle mit EPP-Protokoll
- Normalmodus im FPGA, Parallelisierung der seriellen Eingangsdaten
- Fail-safe-Modus im FPGA, Erzeugung des Nullvektors
- Kontrolleinheit im FPGA, Kontrolle des FS-IFB inklusive Fail-safe-Verhalten und dessen rechtzeitiger Aktivierung
- Steuerung auf PC1, zur Eingabe von Steuerdaten
- Roboterkontrolle auf PC2, zur Steuerung des Roboters

Im folgenden Abschnitt werden die Konzepte und Vorüberlegungen zu diesen Komponenten besprochen. Anschließend folgt die Umsetzung der Konzepte in der Implementierung.

4.3 Implementierungskonzepte

4.3.1 Modellierung von Fail-safe-Verhalten

Zur Beschreibung von Fail-safe-Verhalten für konkrete Anwendungen benötigt man eine Modellierungssprache. Ein adäquates Beschreibungsmittel hierfür stellen endliche Automaten (FSM) dar. Mit FSM's können einfache Sequenzen aber auch komplexe Steuerungen beschrieben werden. Somit bilden sie eine gute Grundlage zur Modellierung von Fail-safe-Verhalten. Der Protokollguard und der Fail-safe-Datengenerator werden durch unabhängig voneinander arbeitende Automaten beschrieben. Die Verbindung der Komponenten übernimmt die Kontrolleinheit, der ebenfalls eine FSM zu Grunde liegt.

4.3.2 Konstruktion des Interfaceblocks

Der Interfaceblock (IFB) aus Kapitel 2 realisiert die Hülle zur Implementierung des Fail-safe-Verhalten in Schnittstellen. Um die Konzepte des IFB, wiederverwendbares Design und Intellectual Properties, in diese Arbeit mit einfließen zu lassen, wird zunächst ein wiederverwendbares Muster des IFB erstellt, das als Grundlage der Implementierung dienen soll.

Um ein wiederverwendbares Muster zu erstellen, sind zuvor ein paar Fragestellungen (Abbildung 4.3) notwendig. Dazu gehören die Fragen:

- Wie beschreibe ich die Muster?
- Welche Eingangssignale werden gebraucht?
- Welche Ausgangssignale werden gerbaucht?
- Wie ist alles flexibel gestaltbar?

Die Muster werden durch Templates realisiert. Zur Beschreibung der Templates wird die Hardwarebeschreibungssprache VHDL verwendet. Sie bietet die Möglichkeit, die Templates nach der Entwicklung zu synthetisieren und auf einem FPGA die Funktionen zu testen.

Nach der Auswahl der Beschreibungsart müssen die Eingänge und Ausgänge des IFB beschrieben werden. Da der IFB zum Großteil aus synchronen Automaten aufgebaut ist, benötigt man Takt- und Reseteingänge. Des Weiteren müssen Daten durch den IFB transportiert werden. Um den Datentransfer zu gewährleisten und verschiedenen Protokollen gerecht zu werden, sind zusätzliche Handshakeleitungen notwendig.

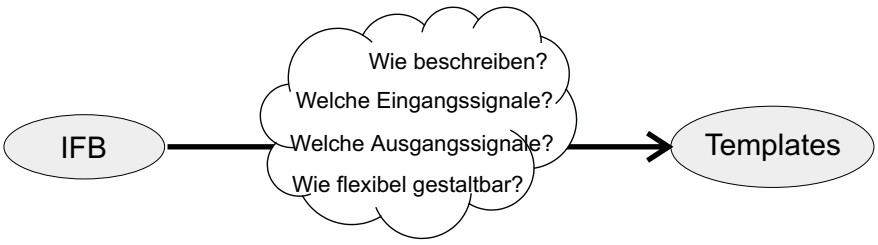


Abbildung 4.3: Vom Modell zum Template des IFB

Ein weitere Grundgedanke ist die Allgemeinheit des Konzeptes. Dazu werden die Templates generisch gestaltet. Das bedeutet, dass die Anzahl von Daten-, Handshake-, Takt- und Resetleitungen nicht von vornherein festgelegt wird, sondern an einer zentralen Stelle definiert werden kann.

4.4 Implementierung

Zunächst wird die Implementierung der Templates besprochen, da sie die Grundlage der Implementierung bildet. Anschließend erfolgt eine Betrachtung der Schnittstellen im Demonstrator und danach die Implementierung des Fail-safe-Verhaltens. Zur Implementierung wurden folgende Programme verwendet:

- Xilinx ISE - Entwicklungsumgebung für Xilinx FPGA's
- Modellsim XE Starter - VHDL-Simulator
- Microsoft Visual C++ - Entwicklungsumgebung für PC-Programme

4.4.1 Templates

Interfaceblock

Das Template des Interfaceblocks hat die Aufgabe, eine Hülle für die Teilkomponenten (Kontrolleinheit, Protokollhandler, Sequenzhandler) bereitzustellen. Die internen Verbindungsleitungen der Komponenten werden definiert, aber nicht in ihrer Breite beschränkt. Somit werden keine Restriktionen bezüglich der Verarbeitungsbreite auferlegt. Das Gleiche gilt für die externen Ports. Alle Parameter, die den IFB und seine Komponenten betreffen, können in diesem Template festgelegt werden, so dass in den anderen Komponenten diesbezüglich keine Änderungen notwendig sind. Das bringt den Vorteil, dass man eine zentrale Stelle besitzt, an der Parameter eingestellt werden können.

Kontrolleinheit

In der Kontrolleinheit werden die Steuerungsautomaten für die Handler integriert. Diese Automaten übernehmen das Starten der Handler und die Auswahl des aktiven Modus. Da diese Automaten abhängig vom konkreten Einsatz des Interfaceblocks sind, wird hier nur eine Hülle bereitgestellt, die die externen Anschlüsse der Komponente definiert. Darin werden dann bei der Implementierung die Automaten zur Steuerung der Handler eingefügt.

Handler

Die Handler haben innerhalb des IFB, zusammen mit ihren eingebundenen Modi, die Aufgabe der Datenverarbeitung. Von ihnen gibt es 3 Stück, den **Protokollhandler für eingehende Signale ($\mathbf{PH}_{in}$)**, den **Protokollhandler für ausgehende Signale ($\mathbf{PH}_{out}$)** und den **Sequenzhandler (SH)**. Die Protokollhandler sind dafür zuständig, die Nachrichten richtig zu dekodieren bzw. zu kodieren und die Nutzdaten aus dem Protokoll zu extrahieren bzw. richtig in ein Protokoll zu verpacken. Der Aufbau von Protokoll- und Sequenzhandler ist im wesentlichen gleich. Deshalb beschränke ich mich im Weiteren auf den allgemeinen Aufbau.

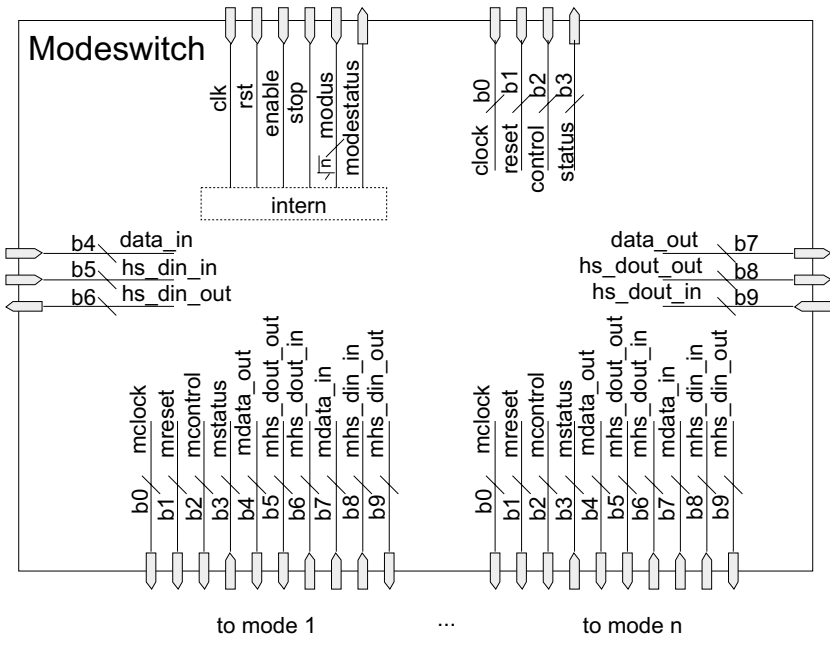


Abbildung 4.4: Aufbau des Modeswitch

Der Handler hat die Aufgabe, die Daten- und Handshakesignale zum aktiven Modus zu routen sowie die Kontrollsignale von beziehungsweise Statussignale zur Kontrolleinheit zu leiten. Herzstück des Handlers ist der **Modeswitch** (Abbildung 4.4). Durch ihn werden die Signale gerouted. Dies geschieht mittels Multiplexern und Demultiplexern. Zur Auswahl der richtigen Leitungen wird an den Multiplexern und Demultiplexern eine Adresse benötigt. Die Adresse steht in einem Register und entspricht dem gerade aktiven Modus. Um die Komponente hinsichtlich der Verarbeitungsbreite flexibel zu halten, sind nur die Signale zur Steuerung des Modeswitch fest vorgegeben. Die internen Verbindungen werden generisch definiert. Auch die Anzahl der Modi, die angeschlossen werden können, ist variabel gestaltet.

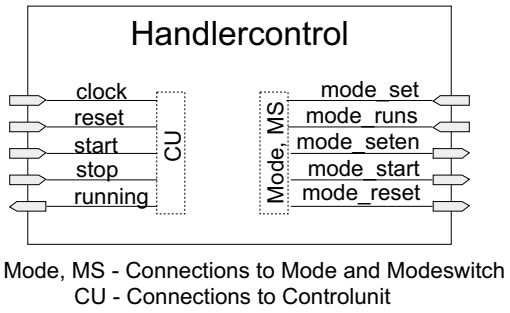


Abbildung 4.5: Ports der Handlercontrol

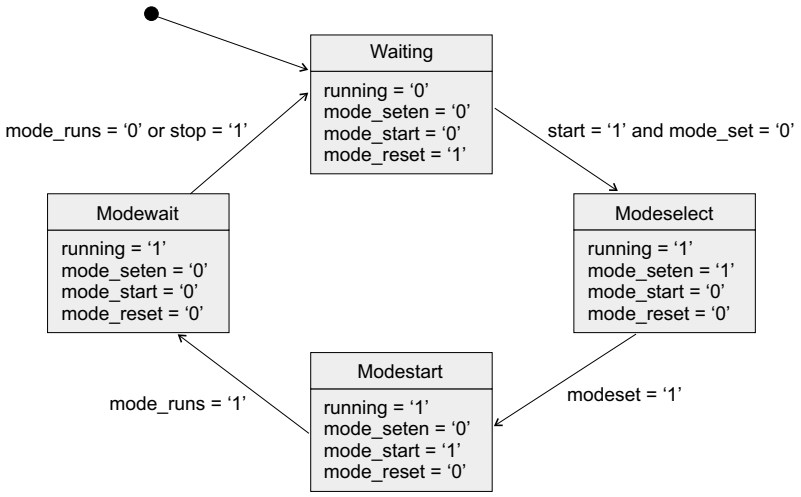


Abbildung 4.6: Automat zur Kontrolle der Handlerfunktionen

Um die Vorgänge im Handler zu steuern wurde die Komponente **Handlercontrol** entwickelt, deren Aufbau in Abbildung 4.5 veranschaulicht ist. Sie besteht aus einem Automaten (Abbildung 4.6), der zum Einen den gewählten Modus starten und stoppen kann und zum Anderen den Modeswitch kontrolliert. Die Kontrolle des Modeswitch beschränkt sich dabei auf ein Signal zum Setzen des Modusregisters und zum Rücksetzen des Modusregisters. Die Übermittlung des eigentlichen Modus geschieht durch Kontrollleitungen von der Kontrolleinheit. Damit brauch die Komponente Handlercontrol bei der Implementierung einer konkreten Funktion des IFB nicht verändert werden.

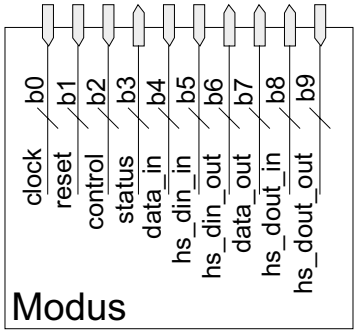


Abbildung 4.7: Ports des Modus

Das Template für den **Modus** wurde als Komponente für einen Handler angelegt. Es stellt ebenso wie das Template der Kontrolleinheit nur eine Hülle bereit, in die die konkrete Funktionalität implementiert werden kann. Im Handler selbst muss nur die Komponente eingebunden werden. Dies ist einfach möglich. Man nimmt nur die vordefinierte Komponentendeklaration und ändert die Namen der Komponenten und ihre Modusnummern entsprechend. Diese Nummer gibt an, welcher Wert im Modusregister des Modeswitch stehen muss, damit dieser Modus aktiv wird.

In den Protokollhandlern kann außerdem eine Komponente **Interruptswitch** integriert werden. Sie hat die Aufgabe, externe Interruptsignale an die Kontrolleinheit weiterzuleiten. Die Interruptsignale können dazu dienen, die Kontrolleinheit darüber zu informieren, dass ein Moduswechsel notwendig ist, um den Empfang eines anderen Datenformates oder eine andere Art der Datenübertragung zu ermöglichen. Eine direkte Verbindung von der Außenwelt zur Kontrolleinheit ist nicht möglich, da dies nur für die Protokollhandler erlaubt ist. Die Funktion des Interruptswitch ist nicht festgelegt und kann je nach Bedarf angepasst werden.

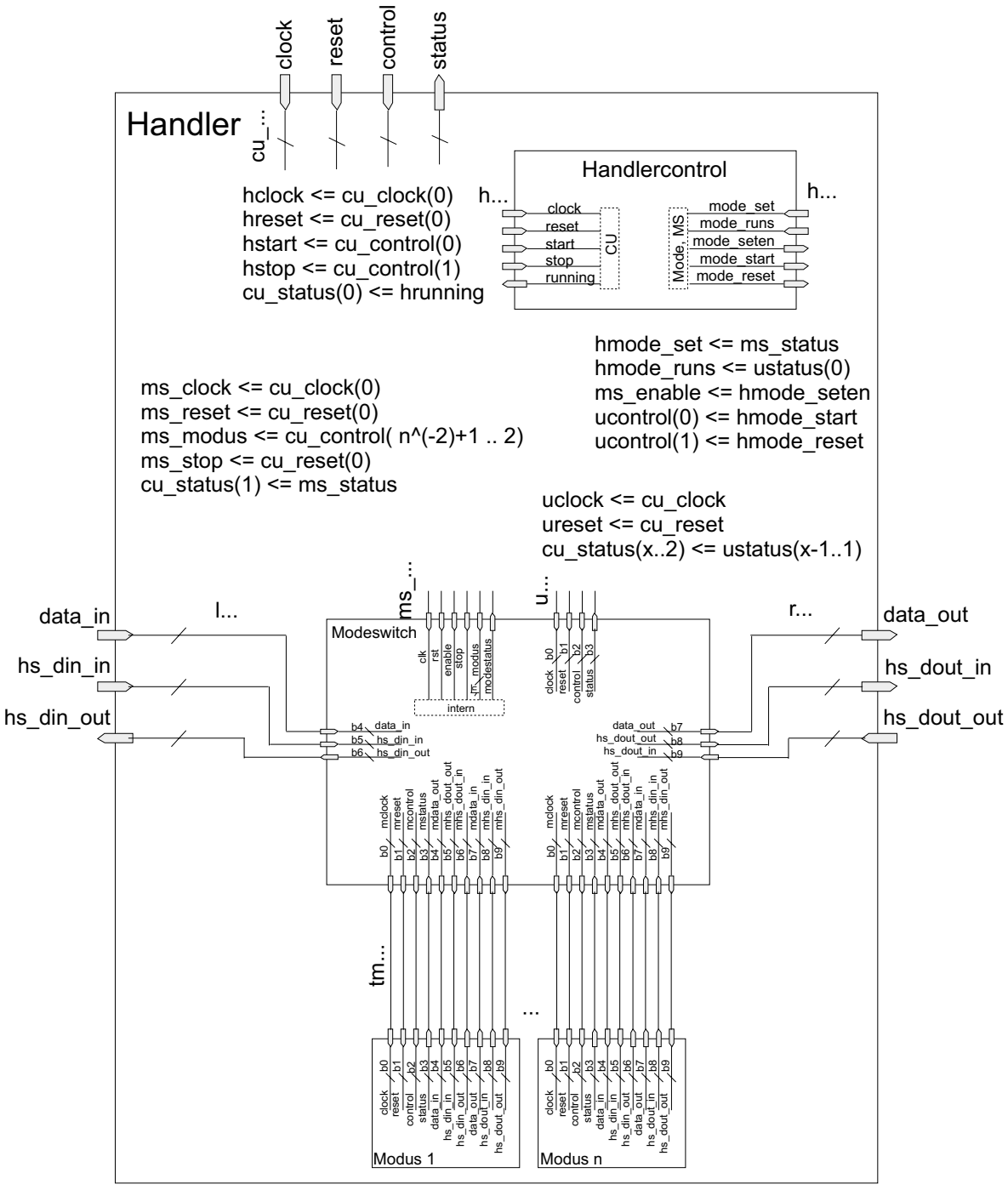


Abbildung 4.8: Aufbau des Handlers. Internen Signalen sind die Buchstaben mit drei Punkten vorangestellt (z.B. u. . . bedeutet uclock, ureset, . . .). Auf die Verdrahtung im oberen Teil wurde aus Gründen der Übersichtlichkeit verzichtet und dafür die Abildungsvorschriften der Signale aufgeführt.

4.4.2 Schnittstellen und ihre Verbindung

EPP

Der Kommunikationsautomat für das EPP-Protokoll wurde anhand von Timingdiagrammen aus [Axe97] erstellt. Er ist in Abbildung 4.9 dargestellt. Der Startzustand ist *Init*. Das Signal *nWrite* gibt an, ob ein Lese- oder Schreibzyklus stattfinden soll. Führt das Signal einen logischen Pegel von 0, wird in den Zustand *Writecycle* gewechselt. Bei einem logischen Pegel von 1 in den Zustand *Readcycle*. Befindet sich der Automat in diesen Zuständen, gibt das Signal *nDataStrobe* mit einem logischen Pegel von 0 an, dass Daten gelesen (Zustand *DataRcycle*) oder geschrieben (Zustand *DataWcycle*) werden sollen. In diesen Zuständen müssen die Daten am Ausgang anliegen oder die vom Sender geschriebenen Daten entgegen genommen werden. Das Signal *nAddrStrobe* dagegen leitet einen Adresslese bzw. -schreibzyklus ein. Dies erfolgt in den Zuständen *AddressRcycle* und *AddressWcycle*, in denen die zu lesende Adresse am Ausgang liegen muss bzw. die zu empfangende Adresse entgegen genommen werden muss. Anschließend wird in den Zustand *Cycleend* gewechselt. Das Signal *nWait* wird auf den logischen Pegel 1

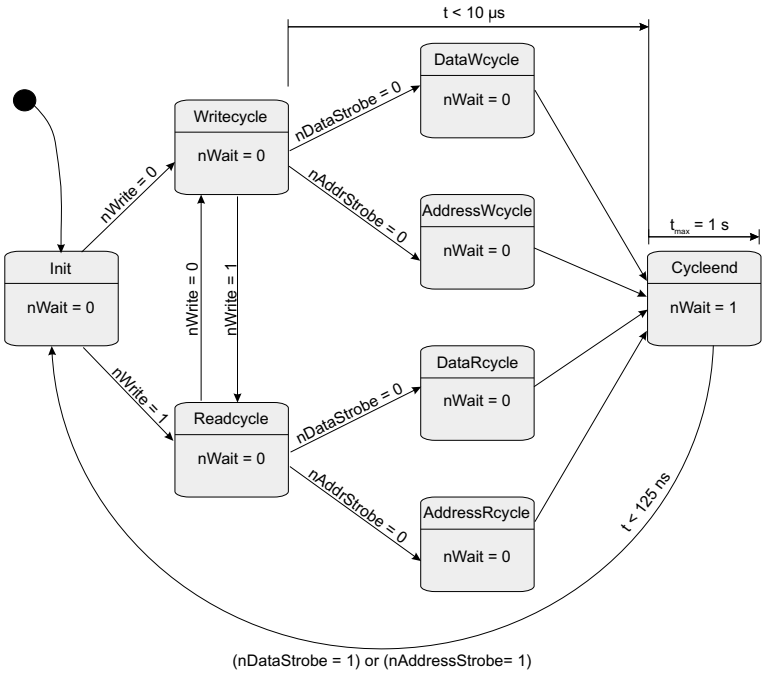


Abbildung 4.9: Kommunikationsautomat des EPP-Protokolls

gesetzt. Dies signalisiert der Gegenstelle, dass der Zyklus abgeschlossen ist. Daraufhin setzt die Gegenstelle das Signal *nDataStrobe* oder *nAddrStrobe*, je nach Art des Zyklus der gerade beendet wurde, auf 1. Damit ist ein Sende- oder Empfangszyklus abgeschlossen und der Automat befindet sich wieder im Startzustand *Init*.

Bei der Implementierung des EPP-Protokolls muss auf ein paar Restriktionen hinsichtlich Zeitlimits geachtet werden. Die Zeitspanne zwischen einem Signalwechsel zwischen *nDataStrobe* bzw. *nAddrStrobe* und *nWait* darf nicht länger als 10 μs betragen. Die Reaktion auf einen logischen Pegel von 1 an *nWait* muss in einem Zeitraum von 1 s erfolgen. Sie besteht darin, dass das Signal *nDataStrobe* oder *nAddrStrobe* auf den logiscen Pegel 1 gesetzt wird. Daraufhin muss das *nWait*-Signal innerhalb von 125 ns wieder auf 0 gesetzt werden.

RS-232

Die Aufgabe des Kommunikationsautomaten für die RS-232-Schnittstelle (Abbildung 4.10) besteht darin, einen Datenrahmen bestehend aus einem Startbit, 8 Datenbits, einem Paritätsbit für ungerade Parität und einem Stopbit zu empfangen. Der Vollständigkeit halber ist in Abbildung 4.10 der

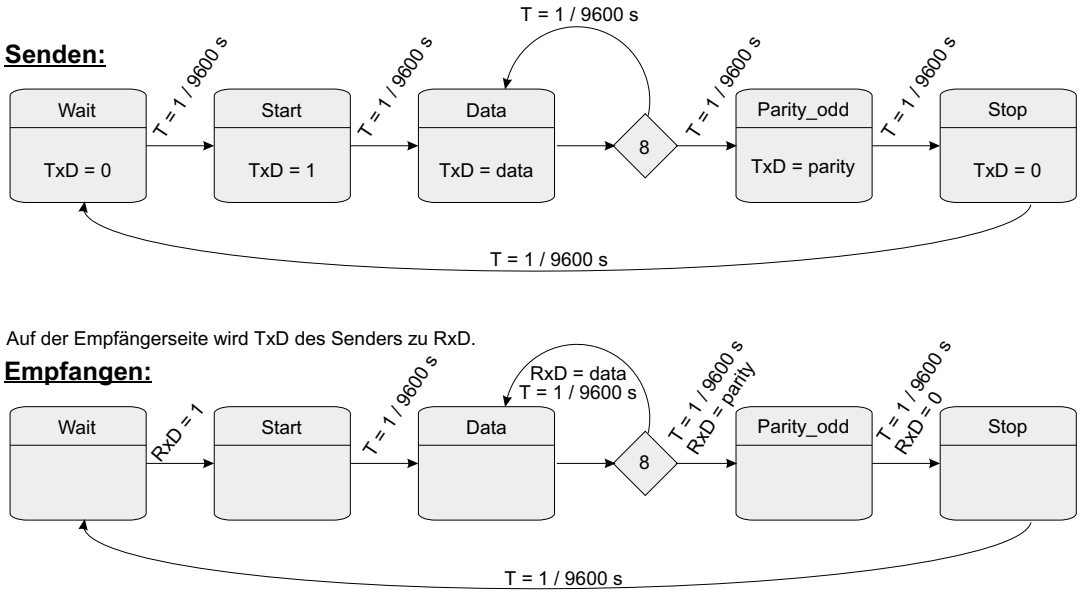


Abbildung 4.10: Kommunikationsautomat der RS-232-Schnittstelle

Kommunikationsautomat zum Senden eines solchen Datenrahmens mit angegeben.

Der Empfang läuft wie folgt ab. Der Automat befindet sich nach dem Start im Zustand *Wait*. Sobald das Signal *RxD* den logischen Pegel 1 annimmt, ist dies das Zeichen, dass eine Übertragung eingeleitet wird. Es wird in den Zustand *Start* gewechselt. Danach folgen 8 Zustände, in denen Daten empfangen werden. Das Signal *RxD* führt dabei den Wert des Bits. Nach den Datenbits erfolgt der Empfang des Paritätsbits im Zustand *Parity_odd*. Anschließend wird in den Zustand *Stop* gewechselt, wenn das Signal *RxD* den logischen Pegel 0 führt und damit das Ende des Datenrahmens anzeigt.

Zur Implementierung ist auch hier eine zeitliche Restriktion zu beachten. Die Frequenz, mit der der Automat getaktet wird, muss 9600 Hz betragen. Dadurch wird eine Übertragungsrate von 9600 Bit/s ermöglicht, die der Übertragungsgeschwindigkeit der RS-232-Schnittstelle bei dem vorliegenden Demonstrator entspricht.

Normalmodus des Sequenzhandlers

Das Bindeglied zwischen der EPP- und der RS-232-Schnittstelle stellt ein Modus des Sequenzhandlers dar. Er wird hier als *Normalmodus* bezeichnet, da er aktiv ist, wenn die Kommunikation ohne Fehler verläuft. Die Funktion des Normalmodus besteht darin, die seriell empfangenen Daten des PH_{in} zu parallelisieren und sie für den PH_{out} in einem Register bereitzustellen, das sich im Sequenzhandler am Ausgang zum PH_{out} befindet. Damit realisiert der Normalmodus eine Transformation zwischen RS-232- und EPP-Schnittstelle.

4.4.3 Implementierung von Fail-safe-Verhalten

Protokollguard

Zur Prüfung der Daten an der RS-232-Schnittstelle ist es notwendig das Pritätsbit auszuwerten. Dafür wurde ein Automat entworfen, der in Abbildung 4.11 dargestellt ist. Er wird in den Modus des PH_{in} integriert und erzeugt zwei Statussignale für die Kontrolleinheit, die angeben, ob die Daten korrekt empfangen wurden und ob das Ergebnis gültig ist.

Die Auswertung des Paritätsbits geschieht folgendermaßen. Im Initialzustand *wait* wird keine Prüfung durchgeführt. Der Automat wartet darin auf den Start der Auswertung. Mit dem Startsignal beginnt die Protokollprüfung. In

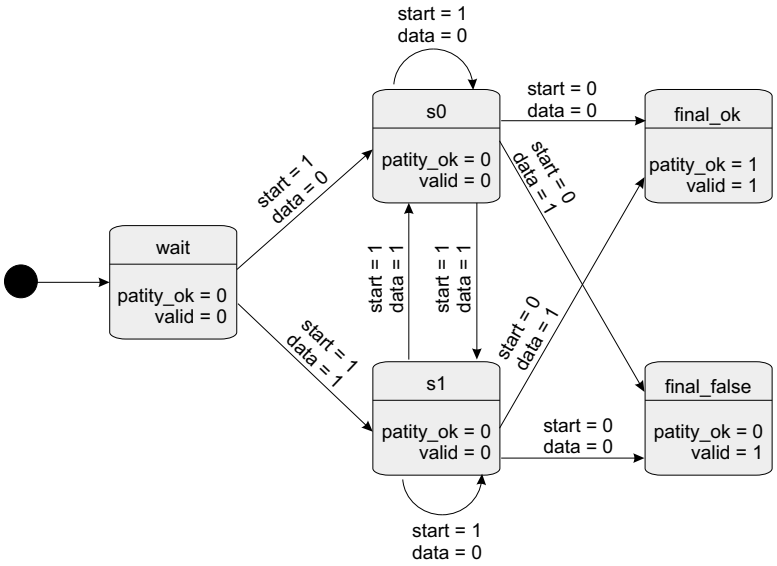


Abbildung 4.11: Automat zur Paritätsprüfung

den Zuständen *s0* und *s1* wird eine XOR-Funktion realisiert, die die ungerade Parität der Datenbits generiert. Eine Änderung des Startsignals auf den Wert 0 zeigt dem Automaten an, dass das nachfoglende Datenbit das Paritätsbit der empfangenen Daten ist. Sind das im Automaten berechnete Paritätsbit und das Paritätsbit der Daten gleich, so sind die Daten korrekt empfangen worden und der Automat wechselt in den Zustand *final_ok*. In diesem Zustand wird ausgegeben, dass die Prüfung erfolgreich war und das Ergebnis gültig ist. Stimmen die beiden Paritätsbits nicht überein, so wechselt der Automat in den Zustand *final_false* und gibt zurück, dass die Prüfung fehlgeschlagen ist und das Ergebnis gültig.

Fail-safe-Datengenerator

Zum Entwerfen des Fail-safe-Datengenerators muss man betrachten, für welches System die Daten generiert werden sollen und welche Daten das System in einen sicheren Zustand bringen. Die Datengenerierung soll für die Roboterkontrolle erfolgen. Deshalb betrachten wir zunächst die Semantik der Daten. Es werden 4 Freiheitsgrade gesteuert. Dabei werden relative Positionierungsdaten verwendet. Sie bestehen aus je 2 Bit pro Freiheitsgrad. Die Vektoren *01*, *10* und *11* führen zu einer Bewegung eines Freiheitsgrades und somit des Roboters. Nur durch den Vektor *00* verändert sich der jeweilige Freiheitsgrad nicht (vgl. Abbildung 4.2).

Anhand der Semantik der Daten erkennt man, dass der sichere Zustand für den Roboter, im Falle eines Fehlers, der bewegungslose Zustand ist. Für Zustände in denen sich der Roboter bewegt, kann man nicht vorhersagen, in welcher Position sich der Roboter nach dem Verlassen des Fail-safe-Zustandes befindet. Dies liegt darin begründet, dass von vornherein nicht bekannt ist, wie lange der Fail-safe-Datengenerator aktiv ist. Demzufolge ist die Anzahl der ausgeführten Bewegung nicht vorhersagbar.

Der bewegungslose Zustand wird hergestellt, indem die Daten aus einem 8 Bit langen Vektor von Nullen bestehen. Der Fail-safe-Datengenerator muss diesen Vektor erzeugen und auf ein Signal der Kontrolleinheit hin, den Vektor in das Ausgangsregister des Sequenzhandlers schreiben. Abbildung 4.12 zeigt diesen Automaten.

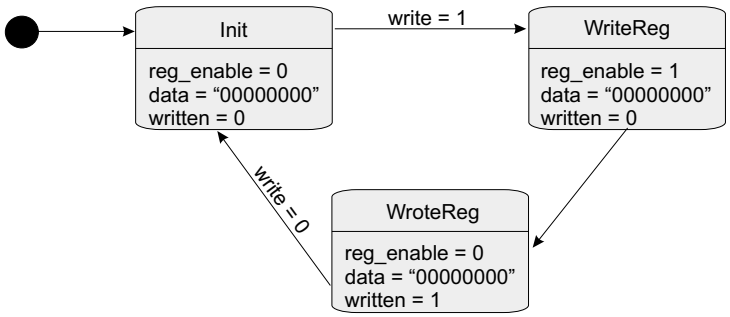


Abbildung 4.12: Automat für den Fail-safe-Datengenerator

Kontrolleinheit

Die Kontrolleinheit setzt sich aus drei Automaten (Abbildung 4.13, einem Timer und einem Taktteiler zusammen. Jeweils ein Automat ist für die Kontrolle eines Handler zuständig. Der Timer erzeugt ein Signal, wenn ein neuer Sendezyklus für das EPP-Protokoll eintreten soll und keine neuen Daten vorliegen. Damit ist er ein Hilfsmittel zur Aktivierung des Fail-safe-Datengenerators. Das Signal wird erzeugt, wenn innerhalb von einer Sekunde keine neuen Steuerungsdaten für den Roboter vorliegen. Der Taktteiler hat die Aufgabe vom Systemtakt einen Takt von 9600 Hz abzuleiten, um den Kommunikationsautomaten für die RS-232-Schnittstelle und den Protokollguard zu takten.

Die Verbindung zwischen Protokollguard und Fail-safe-Datengenerator stellt der Automat zur Kontrolle des Sequenzhandlers (Abbildung 4.13, mittlerer

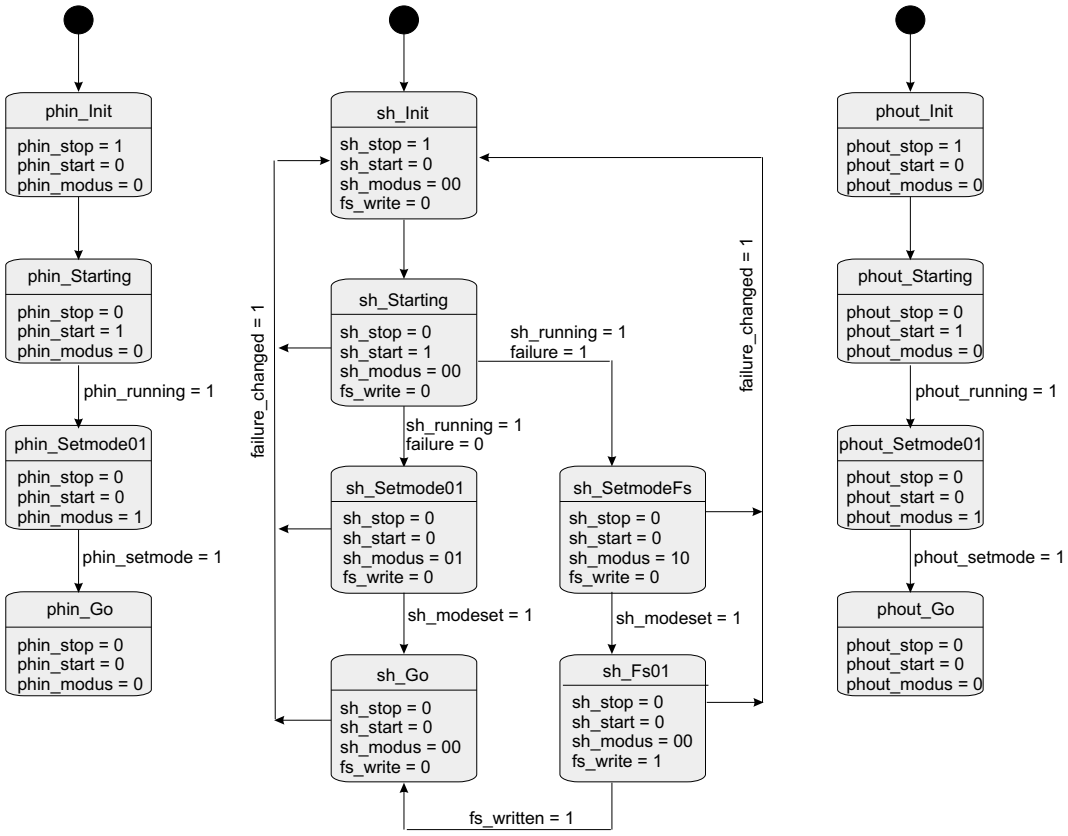


Abbildung 4.13: Automaten der Kontrolleinheit

Automat) her. Er übernimmt diese Aufgabe, weil er für das Umschalten der Modi des Sequenzhandlers verantwortlich ist und der Fail-safe-Datengenerator als ein Modus des Sequenzhandlers implementiert ist. Das Signal *failure_changed* löst das Umschalten des Modus aus. Es wird aus den Fehlersignalen des Protokollguards und dem Signal des Timers erzeugt. Seine Priorität ist am höchsten im Vergleich zu den anderen Eingangssignalen. Es wirkt wie ein synchrones Resetsignal für den Kontrollautomaten. Dadurch kann eine schnelle Reaktion auf Fehlersituationen aus jedem Zustand des Automaten erfolgen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurden Konzepte zur Integration von Fail-safe-Verhalten in Schnittstellen vorgestellt. Zunächst wurden einige Grundlagen beschrieben, auf denen diese Arbeit aufbaut. Im Haupteil erfolgte eine Begriffseinordnung und die Darlegung der Bedeutung von Fail-safe-Verhalten. Außerdem wurde auf die Möglichkeiten der Fehlererkennung eingegangen und gezeigt wie verschiedene Protokolle diese Fehlererkennungsmechanismen nutzen. Anschließend erfolgte die Aufstellung eines Modells zur Integration von Fail-safe-Verhalten in Schnittstellen. Dieses Modell wurde am Demonstrator umgesetzt und validiert. Damit ist auch die praktische Anwendbarkeit nachgewiesen.

Zum Abschluss der Arbeit möchte ich noch einen kleinen Ausblick für die vorgestellten Konzepte geben. In Zukunft könnte Fail-safe-Verhalten in Schnittstellen besonders für dynamisch rekonfigurierbare FPGA's von Bedeutung sein. Bei diesen FPGA's können Teile der Konfiguration während des Betriebes verändert werden. Zwischen den einzelnen Teilen existieren Schnittstellen. Um den sicheren Betrieb auch in der Rekonfigurationsphase zu gewährleisten, kann in den verbindenden Schnittstellen Fail-safe-Verhalten integriert werden.
Die Templates können in Zukunft als Vorlage für einen Codegenerator dienen, der einen Interfaceblock automatisch erstellt. In diesem Zusammenhang ist auch die automatische Generierung von Fail-safe-Verhalten für Schnittstellen denkbar.
Zur Realisierung dieser Punkte sind aber weitergehende Betrachtungen und Untersuchungen notwendig.

A Quelltexte der Templates

In diesem Kapitel sind die Quelltexte der Templates aus Abschnitt 4.4.1 aufgeführt. Sie sind in VHDL beschrieben. Die Quelltexte für den Demonstrator sind im Anhagng B zu finden. Die Trennung wurde vorgenommen, da diese Templates als Grundlage für andere Implementierungen dienen können.

A.1 Interfaceblock (ifb.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   ifb.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity ifb is
11    Generic (
12      ------------------------------------------------------
13      -- generic parameter for incoming data
14      ------------------------------------------------------
15      clock_width       : integer := 1;
16      reset_width       : integer := 1;
17      ------------------------------------------------------
18      -- generic parameter for incoming data
19      ------------------------------------------------------
20      incom_data_width      : integer := 1;
21      incom_hshake_in_width : integer := 1;
22      incom_hshake_out_width: integer := 1;
23      incom_irq_width       : integer := 1;
24      ------------------------------------------------------
25      -- generic parameter for outgoing data
26      ------------------------------------------------------
27      outgo_data_width      : integer := 1;
28      outgo_hshake_in_width : integer := 1;
29      outgo_hshake_out_width: integer := 1;
30      outgo_irq_width       : integer := 1 );
31    Port (
32      ------------------------------------------------------
33      -- Ports for generell control signals
34      ------------------------------------------------------
35      clock : in std_logic_vector(clock_width-1 downto 0);
36      reset : in std_logic_vector(reset_width-1 downto 0);
```

```
37      ------------------------------------------------------
38      -- Ports for incoming data
39      ------------------------------------------------------
40      incom_data      : in  std_logic_vector(incom_data_width-1 downto 0);
41      incom_hshake_in : in  std_logic_vector(incom_hshake_in_width-1 downto 0);
42      incom_hshake_out: out std_logic_vector(incom_hshake_out_width-1 downto 0);
43      incom_irq       : in  std_logic_vector(incom_irq_width-1 downto 0);
44      ------------------------------------------------------
45      -- Ports for outgoing data
46      ------------------------------------------------------
47      outgo_data      : out std_logic_vector(outgo_data_width-1 downto 0);
48      outgo_hshake_in : in  std_logic_vector(outgo_hshake_in_width-1 downto 0);
49      outgo_hshake_out: out std_logic_vector(outgo_hshake_out_width-1 downto 0);
50      outgo_irq       : out std_logic_vector(outgo_irq_width-1 downto 0) );
51  end ifb;
52
53  architecture Behavioral of ifb is
54    constant nom_phi : integer := 1; -- number of modes of phandler_in
55    constant nom_sh  : integer := 1; -- number of modes of shandler
56    constant nom_pho : integer := 1; -- number of modes of phandler_out
57
58    constant cuphi_clock_width  : integer := 1;
59    constant cuphi_reset_width  : integer := 1;
60    constant cuphi_control_width: integer := 3; -- minimun = 3
61    constant cuphi_status_width : integer := 2; -- minimum = 2
62    constant cuphi_irq_width    : integer := 1;
63
64    constant cush_clock_width   : integer := 1;
65    constant cush_reset_width   : integer := 1;
66    constant cush_control_width : integer := 3; -- minimun = 3
67    constant cush_status_width  : integer := 2; -- minimun = 2
68
69    constant cupho_clock_width  : integer := 1;
70    constant cupho_reset_width  : integer := 1;
71    constant cupho_control_width: integer := 3; -- minimun = 3
72    constant cupho_status_width : integer := 2; -- minimun = 2
73    constant cupho_irq_width    : integer := 1;
74
75    constant phish_data_width      : integer := 1; -- minimun = 1
76    constant phish_hshake_in_width : integer := 1;
77    constant phish_hshake_out_width: integer := 1;
78
79    constant shpho_data_width      : integer := 1; -- minimun = 1
80    constant shpho_hshake_in_width : integer := 1;
81    constant shpho_hshake_out_width: integer := 1;
82
83     -- Definition of internal signals
84    signal cuphi_clock  : std_logic_vector(cuphi_clock_width-1 downto 0);
85    signal cuphi_reset  : std_logic_vector(cuphi_reset_width-1 downto 0);
```

```
86    signal cuphi_control: std_logic_vector(cuphi_control_width-1 downto 0);
87    signal cuphi_status : std_logic_vector(cuphi_status_width-1 downto 0);
88    signal cuphi_irq    : std_logic_vector(cuphi_irq_width-1 downto 0);
89
90    signal cush_clock   : std_logic_vector(cush_clock_width-1 downto 0);
91    signal cush_reset   : std_logic_vector(cush_reset_width-1  downto 0);
92    signal cush_control : std_logic_vector(cush_control_width-1 downto 0);
93    signal cush_status  : std_logic_vector(cush_status_width-1 downto 0);
94
95    signal cupho_clock  : std_logic_vector(cupho_clock_width-1 downto 0);
96    signal cupho_reset  : std_logic_vector(cupho_reset_width-1 downto 0);
97    signal cupho_control: std_logic_vector(cupho_control_width-1 downto 0);
98    signal cupho_status : std_logic_vector(cupho_status_width-1 downto 0);
99    signal cupho_irq    : std_logic_vector(cupho_irq_width-1 downto 0);
100
101   signal phish_data      : std_logic_vector(phish_data_width-1 downto 0);
102   signal phish_hshake_in : std_logic_vector(phish_hshake_in_width-1 downto 0);
103   signal phish_hshake_out: std_logic_vector(phish_hshake_out_width-1 downto 0);
104
105   signal shpho_data      : std_logic_vector(shpho_data_width-1 downto 0);
106   signal shpho_hshake_in : std_logic_vector(shpho_hshake_in_width-1 downto 0);
107   signal shpho_hshake_out: std_logic_vector(shpho_hshake_out_width-1 downto 0);
108
109   -- local signals
110   signal lclock : std_logic_vector(clock_width-1 downto 0);
111   signal lreset : std_logic_vector(reset_width-1 downto 0);
112
113   -- componentdedefinitions for cu, phandler_in, shandler and phandler_out
114   component cu
115    Generic(
116     ------------------------------------------------
117     -- generic parameters for external control signals from IFB
118     ------------------------------------------------
119     clock_width : integer;
120     reset_width : integer;
121     ------------------------------------------------
122     -- generic parameters for CU to PHin
123     ------------------------------------------------
124     phi_clock_width   : integer;
125     phi_reset_width   : integer;
126     phi_control_width : integer;
127     phi_status_width  : integer;
128     phi_irq_width     : integer;
129     ------------------------------------------------
130       -- generic parameters for CU to SH
131     ------------------------------------------------
132     sh_clock_width    : integer;
133     sh_reset_width    : integer;
134     sh_control_width  : integer;
```

```
135       sh_status_width   : integer;
136       ---------------------------------------------
137         -- generic parameters for CU to PHout
138       ---------------------------------------------
139       pho_clock_width   : integer;
140       pho_reset_width   : integer;
141       pho_control_width : integer;
142       pho_status_width  : integer;
143       pho_irq_width     : integer );
144      Port(
145       ---------------------------------------------------
146       -- Ports for external control signals from IFB
147       ---------------------------------------------------
148       clock   : in std_logic_vector(clock_width -1 downto 0);
149       reset   : in std_logic_vector(reset_width -1 downto 0);
150       ---------------------------------------------
151       -- Ports for CU to PHin
152       ---------------------------------------------
153       phi_clock  : out std_logic_vector(phi_clock_width-1 downto 0);
154       phi_reset  : out std_logic_vector(phi_reset_width-1 downto 0);
155       phi_control: out std_logic_vector(phi_control_width-1 downto 0);
156       phi_status : in  std_logic_vector(phi_status_width-1 downto 0);
157       phi_irq    : in  std_logic_vector(phi_irq_width-1 downto 0);
158       ---------------------------------------------
159        -- Ports for CU to SH
160       ---------------------------------------------
161       sh_clock  : out std_logic_vector(sh_clock_width-1 downto 0);
162       sh_reset  : out std_logic_vector(sh_reset_width-1 downto 0);
163       sh_control: out std_logic_vector(sh_control_width-1 downto 0);
164       sh_status : in  std_logic_vector(sh_status_width-1 downto 0);
165       ---------------------------------------------
166        -- Ports for CU to PHout
167       ---------------------------------------------
168       pho_clock  : out std_logic_vector(pho_clock_width-1 downto 0);
169       pho_reset  : out std_logic_vector(pho_reset_width-1 downto 0);
170       pho_control: out std_logic_vector(pho_control_width-1 downto 0);
171       pho_status : in  std_logic_vector(pho_status_width-1 downto 0);
172       pho_irq    : out std_logic_vector(pho_irq_width-1 downto 0) );
173     end component;
174
175     component phandler_in
176      Generic(
177       nom : integer; -- number of modes
178       ---------------------------------------------
179       -- generic parameter for PHin to CU
180       ---------------------------------------------
181       cu_clock_width  : integer;
182       cu_reset_width  : integer;
183       cu_control_width: integer;
```

```
184       cu_status_width : integer;
185       cu_irq_width    : integer;
186       ------------------------------------------------
187       -- generic parameter for PHin to IFB incoming borders
188       ------------------------------------------------
189       ifb_data_width      : integer;
190       ifb_hshake_in_width : integer;
191       ifb_hshake_out_width: integer;
192       ifb_irq_width       : integer;
193       ------------------------------------------------
194       -- generic parameter for PHin to SH
195       ------------------------------------------------
196       sh_data_width       : integer;
197       sh_hshake_in_width  : integer;
198       sh_hshake_out_width : integer );
199      Port(
200       ------------------------------------------------
201       -- Ports for PHin to CU
202       ------------------------------------------------
203       cu_clock  : in  std_logic_vector(clock_width-1 downto 0);
204       cu_reset  : in  std_logic_vector(cu_reset_width-1 downto 0);
205       cu_control: in  std_logic_vector(cu_control_width-1 downto 0);
206       cu_status : out std_logic_vector(cu_status_width-1 downto 0);
207       cu_irq    : out std_logic_vector(cu_irq_width-1 downto 0);
208       ------------------------------------------------
209       -- Ports for PHin to IFB incoming borders
210       ------------------------------------------------
211       ifb_data      : in  std_logic_vector(ifb_data_width-1 downto 0);
212       ifb_hshake_in : in  std_logic_vector(ifb_hshake_in_width-1 downto 0);
213       ifb_hshake_out: out std_logic_vector(ifb_hshake_out_width-1 downto 0);
214       ifb_irq       : in  std_logic_vector(ifb_irq_width-1 downto 0);
215       ------------------------------------------------
216       -- Ports for PHin to SH
217       ------------------------------------------------
218       sh_data       : out std_logic_vector(sh_data_width-1 downto 0);
219       sh_hshake_in  : in  std_logic_vector(sh_hshake_in_width-1 downto 0);
220       sh_hshake_out : out std_logic_vector(sh_hshake_out_width-1 downto 0) );
221     end component;
222
223     component shandler
224      Generic(
225       nom : integer;
226       ------------------------------------------------------
227       -- generic parameter for CU to SH
228       ------------------------------------------------------
229       cu_clock_width  : integer;
230       cu_reset_width  : integer;
231       cu_control_width: integer;
232       cu_status_width : integer;
```

```
233       ------------------------------------------------------
234       -- ports for PHin to SH
235       ------------------------------------------------------
236       phi_data_width      : integer;
237       phi_hshake_in_width : integer;
238       phi_hshake_out_width: integer;
239       ------------------------------------------------------
240       -- ports for PHout to SH
241       ------------------------------------------------------
242       pho_data_width      : integer;
243       pho_hshake_in_width : integer;
244       pho_hshake_out_width: integer );
245      Port(
246       ------------------------------------------------------
247       -- ports for CU to SH
248       ------------------------------------------------------
249       cu_clock  : in  std_logic_vector(clock_width-1 downto 0);
250       cu_reset  : in  std_logic_vector(cu_reset_width-1 downto 0);
251       cu_control: in  std_logic_vector(cu_control_width-1 downto 0);
252       cu_status : out std_logic_vector(cu_status_width-1 downto 0);
253       ------------------------------------------------------
254       -- ports for PHin to SH
255       ------------------------------------------------------
256       phi_data      : in  std_logic_vector(phi_data_width-1 downto 0);
257       phi_hshake_in : in  std_logic_vector(phi_hshake_in_width-1 downto 0);
258       phi_hshake_out: out std_logic_vector(phi_hshake_out_width-1 downto 0);
259       ------------------------------------------------------
260       -- ports for PHout to SH
261       ------------------------------------------------------
262       pho_data      : out std_logic_vector(pho_data_width-1 downto 0);
263       pho_hshake_in : in  std_logic_vector(pho_hshake_in_width-1 downto 0);
264       pho_hshake_out: out std_logic_vector(pho_hshake_out_width-1 downto 0) );
265     end component;
266
267     component phandler_out
268      Generic(
269       nom : integer;
270       ------------------------------------------------
271       -- generic parameter for PHin to CU
272       ------------------------------------------------
273       cu_clock_width  : integer;
274       cu_reset_width  : integer;
275       cu_control_width: integer;
276       cu_status_width : integer;
277       cu_irq_width    : integer;
278       ------------------------------------------------
279       -- generic parameter for PHin to IFB incoming borders
280       ------------------------------------------------
281       ifb_data_width      : integer;
```

```
282       ifb_hshake_in_width : integer;
283       ifb_hshake_out_width: integer;
284       ifb_irq_width       : integer;
285       ---------------------------------------------
286       -- generic parameter for PHin to SH
287       ---------------------------------------------
288       sh_data_width       : integer;
289       sh_hshake_in_width  : integer;
290       sh_hshake_out_width : integer );
291      Port(
292       ---------------------------------------------
293       -- Ports for PHin to CU
294       ---------------------------------------------
295       cu_clock  : in  std_logic_vector(clock_width-1 downto 0);
296       cu_reset  : in  std_logic_vector(cu_reset_width-1 downto 0);
297       cu_control: in  std_logic_vector(cu_control_width-1 downto 0);
298       cu_status : out std_logic_vector(cu_status_width-1 downto 0);
299       cu_irq    : in std_logic_vector(cu_irq_width-1 downto 0);
300       ---------------------------------------------
301       -- Ports for PHin to IFB incoming borders
302       ---------------------------------------------
303       ifb_data      : out  std_logic_vector(ifb_data_width-1 downto 0);
304       ifb_hshake_in : in  std_logic_vector(ifb_hshake_in_width-1 downto 0);
305       ifb_hshake_out: out std_logic_vector(ifb_hshake_out_width-1 downto 0);
306       ifb_irq       : out  std_logic_vector(ifb_irq_width-1 downto 0);
307       ---------------------------------------------
308       -- Ports for PHin to SH
309       ---------------------------------------------
310       sh_data      : in  std_logic_vector(sh_data_width-1 downto 0);
311       sh_hshake_in : in  std_logic_vector(sh_hshake_in_width-1 downto 0);
312       sh_hshake_out: out std_logic_vector(sh_hshake_out_width-1 downto 0));
313     end component;
314     -- end of component definitions
315
316   begin
317     lclock <= clock;
318     lreset <= reset;
319
320   CONTROLUNIT: cu
321       GENERIC MAP(
322         ---------------------------------------------------
323         -- generic parameters for external control signals from IFB
324         ---------------------------------------------------
325         clock_width => clock_width,
326         reset_width => reset_width,
327         ---------------------------------------------
328         -- generic parameters for CU to PHin
329         ---------------------------------------------
330         phi_clock_width   => cuphi_clock_width,
```

```
331         phi_reset_width   => cuphi_reset_width,
332         phi_control_width => cuphi_control_width,
333         phi_status_width  => cuphi_status_width,
334         phi_irq_width     => cuphi_irq_width,
335         ------------------------------------------------
336          -- generic parameters for CU to SH
337         ------------------------------------------------
338         sh_clock_width    => cush_clock_width,
339         sh_reset_width    => cush_reset_width,
340         sh_control_width  => cush_control_width,
341         sh_status_width   => cush_status_width,
342         ------------------------------------------------
343          -- generic parameters for CU to PHout
344         ------------------------------------------------
345         pho_clock_width   => cupho_clock_width,
346         pho_reset_width   => cupho_reset_width,
347         pho_control_width => cupho_control_width,
348         pho_status_width  => cupho_status_width,
349         pho_irq_width     => cupho_irq_width )
350       PORT MAP(
351         ------------------------------------------------------
352         -- Ports for external control signals from IFB
353         ------------------------------------------------------
354         clock => lclock,
355         reset => lreset,
356         ------------------------------------------------
357         -- Ports for CU to PHin
358         ------------------------------------------------
359         phi_clock   => cuphi_clock,
360         phi_reset   => cuphi_reset,
361         phi_control => cuphi_control,
362         phi_status  => cuphi_status,
363         phi_irq     => cuphi_irq,
364         ------------------------------------------------
365          -- Ports for CU to SH
366         ------------------------------------------------
367         sh_clock   => cush_clock,
368         sh_reset   => cush_reset,
369         sh_control => cush_control,
370         sh_status  => cush_status,
371         ------------------------------------------------
372          -- Ports for CU to PHout
373         ------------------------------------------------
374         pho_clock   => cupho_clock,
375         pho_reset   => cupho_reset,
376         pho_control => cupho_control,
377         pho_status  => cupho_status,
378         pho_irq     => cupho_irq );
379
```

```
380  PHin:   phandler_in
381      GENERIC MAP(
382        nom => nom_phi,
383        ---------------------------------------------
384        -- generic parameter for PHin to CU
385        ---------------------------------------------
386        cu_clock_width   => cuphi_clock_width,
387        cu_reset_width   => cuphi_reset_width,
388        cu_control_width => cuphi_control_width,
389        cu_status_width  => cuphi_status_width,
390        cu_irq_width     => cuphi_irq_width,
391        ---------------------------------------------
392        -- generic parameter for PHin to IFB incoming borders
393        ---------------------------------------------
394        ifb_data_width       => incom_data_width,
395        ifb_hshake_in_width  => incom_hshake_in_width,
396        ifb_hshake_out_width => incom_hshake_out_width,
397        ifb_irq_width        => incom_irq_width,
398        ---------------------------------------------
399        -- generic parameter for PHin to SH
400        ---------------------------------------------
401        sh_data_width       => phish_data_width,
402        sh_hshake_in_width  => phish_hshake_in_width,
403        sh_hshake_out_width => phish_hshake_out_width )
404      PORT MAP(
405        ---------------------------------------------
406        -- Ports for PHin to CU
407        ---------------------------------------------
408        cu_clock   => cuphi_clock,
409        cu_reset   => cuphi_reset,
410        cu_control => cuphi_control,
411        cu_status  => cuphi_status,
412        cu_irq     => cuphi_irq,
413        ---------------------------------------------
414        -- Ports for PHin to IFB incoming borders
415        ---------------------------------------------
416        ifb_data       => incom_data,
417        ifb_hshake_in  => incom_hshake_in,
418        ifb_hshake_out => incom_hshake_out,
419        ifb_irq        => incom_irq,
420        ---------------------------------------------
421        -- Ports for PHin to SH
422        ---------------------------------------------
423        sh_data       => phish_data,
424        sh_hshake_in  => phish_hshake_in,
425        sh_hshake_out => phish_hshake_out );
426
427  SH:   shandler
428      GENERIC MAP(
```

```
429         nom => nom_sh,
430         ------------------------------------------------------
431         -- generic parameter for CU to SH
432         ------------------------------------------------------
433         cu_clock_width   => cush_clock_width,
434         cu_reset_width   => cush_reset_width,
435         cu_control_width => cush_control_width,
436         cu_status_width  => cush_status_width,
437         ------------------------------------------------------
438         -- ports for PHin to SH
439         ------------------------------------------------------
440         phi_data_width       => phish_data_width,
441         phi_hshake_in_width  => phish_hshake_out_width,
442         phi_hshake_out_width => phish_hshake_in_width,
443         ------------------------------------------------------
444         -- ports for PHout to SH
445         ------------------------------------------------------
446         pho_data_width       => shpho_data_width,
447         pho_hshake_in_width  => shpho_hshake_in_width,
448         pho_hshake_out_width => shpho_hshake_out_width )
449       PORT MAP(
450         ------------------------------------------------------
451         -- ports for CU to SH
452         ------------------------------------------------------
453         cu_clock   => cush_clock,
454         cu_reset   => cush_reset,
455         cu_control => cush_control,
456         cu_status  => cush_status,
457         ------------------------------------------------------
458         -- ports for PHin to SH
459         ------------------------------------------------------
460         phi_data       => phish_data,
461         phi_hshake_in  => phish_hshake_out,
462         phi_hshake_out => phish_hshake_in,
463         ------------------------------------------------------
464         -- ports for PHout to SH
465         ------------------------------------------------------
466         pho_data       => shpho_data,
467         pho_hshake_in  => shpho_hshake_in,
468         pho_hshake_out => shpho_hshake_out );
469
470   PHout:  phandler_out
471       GENERIC MAP(
472         nom => nom_pho,
473         ---------------------------------------------
474         -- generic parameter for PHin to CU
475         ---------------------------------------------
476         cu_clock_width    => cupho_clock_width,
477         cu_reset_width    => cupho_reset_width,
```

```
478         cu_control_width  => cupho_control_width,
479         cu_status_width   => cupho_status_width,
480         cu_irq_width      => cupho_irq_width,
481         ---------------------------------------------
482         -- generic parameter for PHin to IFB incoming borders
483         ---------------------------------------------
484         ifb_data_width       => outgo_data_width,
485         ifb_hshake_in_width  => outgo_hshake_in_width,
486         ifb_hshake_out_width => outgo_hshake_out_width,
487         ifb_irq_width        => outgo_irq_width,
488         ---------------------------------------------
489         -- generic parameter for PHin to SH
490         ---------------------------------------------
491         sh_data_width       => shpho_data_width,
492         sh_hshake_in_width  => shpho_hshake_out_width,
493         sh_hshake_out_width => shpho_hshake_in_width )
494       PORT MAP(
495         ---------------------------------------------
496         -- Ports for PHin to CU
497         ---------------------------------------------
498         cu_clock   => cupho_clock,
499         cu_reset   => cupho_reset,
500         cu_control => cupho_control,
501         cu_status  => cupho_status,
502         cu_irq     => cupho_irq,
503         ---------------------------------------------
504         -- Ports for PHin to IFB incoming borders
505         ---------------------------------------------
506         ifb_data       => outgo_data,
507         ifb_hshake_in  => outgo_hshake_in,
508         ifb_hshake_out => outgo_hshake_out,
509         ifb_irq        => outgo_irq,
510         ---------------------------------------------
511         -- Ports for PHin to SH
512         ---------------------------------------------
513         sh_data       => shpho_data,
514         sh_hshake_in  => shpho_hshake_out,
515         sh_hshake_out => shpho_hshake_in );
516
517   end Behavioral;
```

A.2 Controlunit (cu.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   cu.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity cu is
11     Generic (
12      ------------------------------------------------------
13      -- generic parameters for external control signals from IFB
14      ------------------------------------------------------
15      clock_width : integer := 1;
16      reset_width : integer := 1;
17      ------------------------------------------------
18      -- generic parameters for CU to PHin
19      ------------------------------------------------
20      phi_clock_width   : integer := 1;
21      phi_reset_width   : integer := 1;
22      phi_control_width : integer := 3;
23      phi_status_width  : integer := 2;
24      phi_irq_width     : integer := 1;
25      ------------------------------------------------
26       -- generic parameters for CU to SH
27      ------------------------------------------------
28      sh_clock_width   : integer := 1;
29      sh_reset_width   : integer := 1;
30      sh_control_width : integer := 3;
31      sh_status_width  : integer := 2;
32      ------------------------------------------------
33       -- generic parameters for CU to PHout
34      ------------------------------------------------
35      pho_clock_width   : integer := 1;
36      pho_reset_width   : integer := 1;
37      pho_control_width : integer := 3;
38      pho_status_width  : integer := 2;
39      pho_irq_width     : integer := 1 );
40     Port (
41      ------------------------------------------------------
42      -- Ports for external control signals from IFB
43      ------------------------------------------------------
44      clock : in std_logic_vector(clock_width-1 downto 0);
45      reset : in std_logic_vector(reset_width-1 downto 0);
46      ------------------------------------------------
47      -- Ports for CU to PHin
```

```
48      ---------------------------------------------
49      phi_clock   : out std_logic_vector(phi_clock_width-1 downto 0);
50      phi_reset   : out std_logic_vector(phi_reset_width-1 downto 0);
51      phi_control : out std_logic_vector(phi_control_width-1 downto 0);
52      phi_status  : in  std_logic_vector(phi_status_width-1 downto 0);
53      phi_irq     : in  std_logic_vector(phi_irq_width-1 downto 0);
54      ---------------------------------------------
55       -- Ports for CU to SH
56      ---------------------------------------------
57      sh_clock    : out std_logic_vector(sh_clock_width-1 downto 0);
58      sh_reset    : out std_logic_vector(sh_reset_width-1 downto 0);
59      sh_control  : out std_logic_vector(sh_control_width-1 downto 0);
60      sh_status   : in  std_logic_vector(sh_status_width-1 downto 0);
61      ---------------------------------------------
62       -- Ports for CU to PHout
63      ---------------------------------------------
64      pho_clock   : out std_logic_vector(pho_clock_width-1 downto 0);
65      pho_reset   : out std_logic_vector(pho_reset_width-1 downto 0);
66      pho_control : out std_logic_vector(pho_control_width-1 downto 0);
67      pho_status  : in  std_logic_vector(pho_status_width-1 downto 0);
68      pho_irq     : out std_logic_vector(pho_irq_width-1 downto 0) );
69  end cu;
70
71  architecture Behavioral of cu is
72    -- definition of local signals
73    signal lclock : std_logic_vector(clock_width-1 downto 0);
74    signal lreset : std_logic_vector(reset_width-1 downto 0);
75
76    signal lphi_clock  : std_logic_vector(phi_clock_width-1 downto 0);
77    signal lphi_reset  : std_logic_vector(phi_reset_width-1 downto 0);
78    signal lphi_control: std_logic_vector(phi_control_width-1 downto 0);
79    signal lphi_status : std_logic_vector(phi_status_width-1 downto 0);
80    signal lphi_irq    : std_logic_vector(phi_irq_width-1 downto 0);
81
82    signal lsh_clock   : std_logic_vector(sh_clock_width-1 downto 0);
83    signal lsh_reset   : std_logic_vector(sh_reset_width-1 downto 0);
84    signal lsh_control : std_logic_vector(sh_control_width-1 downto 0);
85    signal lsh_status  : std_logic_vector(sh_status_width-1 downto 0);
86
87    signal lpho_clock  : std_logic_vector(pho_clock_width-1 downto 0);
88    signal lpho_reset  : std_logic_vector(pho_reset_width-1 downto 0);
89    signal lpho_control: std_logic_vector(pho_control_width-1 downto 0);
90    signal lpho_status : std_logic_vector(pho_status_width-1 downto 0);
91    signal lpho_irq    : std_logic_vector(pho_irq_width-1 downto 0);
92
93  begin
94    -- mapping of local signals with global signals
95    phi_clock   <= lphi_clock;
96    phi_reset   <= lreset;
```

```
97     phi_control <= lphi_control;
98     lphi_status <= phi_status;
99     lphi_irq    <= phi_irq;
100
101    sh_clock   <= lsh_clock;
102    sh_reset   <= lreset;
103    sh_control <= lsh_control;
104    lsh_status <= sh_status;
105
106    pho_clock   <= lpho_clock;
107    pho_reset   <= lreset;
108    pho_control <= lpho_control;
109    lpho_status <= pho_status;
110    pho_irq     <= lpho_irq;
111
112    lphi_clock <= clock;
113    lsh_clock  <= clock;
114    lpho_clock <= clock;
115
116          -- implementation of control fsm's
117  end Behavioral;
```

A.3 Handler

Da der Aufbau der Protokollhandler und Sequenzhandler gleich ist, wird hier nur der Sequenzhandler ausführlich angegeben. Bei den Protokollhandlern werden nur die Portdeklarationen und die Erweiterungen, die nicht im Sequenzhandler vorkommen, angegeben.

A.3.1 Sequencehandler (shandler.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   shandler.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9   use WORK.ifb_functions.ALL;
10
11  entity shandler is
12    Generic (
13      nom : integer := 1;
```

```
14      ------------------------------------------------------
15      -- generic parameter for CU to SH
16      ------------------------------------------------------
17      cu_clock_width   : integer := 1;
18      cu_reset_width   : integer := 1;
19      cu_control_width : integer := 3;
20      cu_status_width  : integer := 2;
21      ------------------------------------------------------
22      -- ports for PHin to SH
23      ------------------------------------------------------
24      phi_data_width       : integer := 1;
25      phi_hshake_in_width  : integer := 1;
26      phi_hshake_out_width : integer := 1;
27      ------------------------------------------------------
28      -- ports for PHout to SH
29      ------------------------------------------------------
30      pho_data_width       : integer := 1;
31      pho_hshake_in_width  : integer := 1;
32      pho_hshake_out_width : integer := 1 );
33    Port (
34      ------------------------------------------------------
35      -- ports for CU to SH
36      ------------------------------------------------------
37      cu_clock  : in  std_logic_vector(cu_clock_width-1 downto 0);
38      cu_reset  : in  std_logic_vector(cu_reset_width-1 downto 0);
39      cu_control: in  std_logic_vector(cu_control_width-1 downto 0);
40      cu_status : out std_logic_vector(cu_status_width-1 downto 0);
41      ------------------------------------------------------
42      -- ports for PHin to SH
43      ------------------------------------------------------
44      phi_data : in  std_logic_vector(phi_data_width-1 downto 0);
45      phi_hshake_in : in  std_logic_vector
46                              (phi_hshake_in_width-1 downto 0);
47      phi_hshake_out: out std_logic_vector
48                              (phi_hshake_out_width-1 downto 0);
49      ------------------------------------------------------
50      -- ports for PHout to SH
51      ------------------------------------------------------
52      pho_data : out std_logic_vector(pho_data_width-1 downto 0);
53      pho_hshake_in : in  std_logic_vector
54                              (pho_hshake_in_width-1 downto 0);
55      pho_hshake_out: out std_logic_vector
56                              (pho_hshake_out_width-1 downto 0));
57  end shandler;
58
59  architecture Behavioral of shandler is
60    -- definition of constants ---------------------------------------
61    constant mcontrol_width: integer:= cu_control_width-lenofint(nom)+1;
62    constant mstatus_width : integer:= cu_status_width;
```

```
63    constant mclock_width  : integer:= cu_clock_width;
64    constant mreset_width  : integer:= cu_reset_width;
65
66    -- This constants are defined to save work with readressing
67    -- in the modus-portmap
68    -- only the names of the constants have to be changed in the portmap
69    constant mode_nr01 : integer := 1;
70    constant mode_nr02 : integer := 2;
71    -- ... for each mode that is used
72
73    -- definition of local signals from and to modeswitch -------------
74    signal ms_clock : std_logic;
75    signal ms_reset : std_logic;
76    signal ms_modus : std_logic_vector(lenofint(nom)-1 downto 0);
77    signal ms_enable: std_logic;
78    signal ms_stop  : std_logic;
79    signal ms_status: std_logic;
80
81    signal uclock  : std_logic_vector(mclock_width-1 downto 0);
82    signal ureset  : std_logic_vector(mreset_width-1 downto 0);
83    signal ucontrol: std_logic_vector(mcontrol_width-1 downto 0);
84    signal ustatus : std_logic_vector(mstatus_width-1 downto 0);
85
86    signal ldata  : std_logic_vector(phi_data_width-1 downto 0);
87    signal lhsin  : std_logic_vector(phi_hshake_in_width-1 downto 0);
88    signal lhsout : std_logic_vector(phi_hshake_out_width-1 downto 0);
89
90    signal rdata : std_logic_vector(pho_data_width-1 downto 0);
91    signal rhsin : std_logic_vector(pho_hshake_in_width-1 downto 0);
92    signal rhsout: std_logic_vector(pho_hshake_out_width-1 downto 0);
93    -- ports for connection to modes
94    signal tmclock      : std_logic_vector
95                          (((mclock_width * nom)-1) downto 0);
96    signal tmreset      : std_logic_vector
97                          (((mreset_width * nom)-1) downto 0);
98    signal tmdata_out   : std_logic_vector
99                          (((phi_data_width * nom)-1) downto 0);
100   signal tmhs_dout_in : std_logic_vector
101                         (((phi_hshake_out_width * nom)-1) downto 0);
102   signal tmhs_dout_out: std_logic_vector
103                         (((phi_hshake_in_width * nom)-1) downto 0);
104   signal tmdata_in    : std_logic_vector
105                         (((pho_data_width * nom)-1) downto 0);
106   signal tmhs_din_in  : std_logic_vector
107                         (((pho_hshake_out_width * nom)-1) downto 0);
108   signal tmhs_din_out : std_logic_vector
109                         (((pho_hshake_in_width * nom)-1) downto 0);
110   signal tmcontrol    : std_logic_vector
111                         (((mcontrol_width * nom)-1) downto 0);
```

```
112     signal tmstatus     : std_logic_vector
113                           (((mstatus_width * nom)-1) downto 0);
114
115     -- local signals for handlercontrol
116     signal hreset     : std_logic; -- from cu
117     signal hclock     : std_logic; -- from cu
118     signal hstart     : std_logic; -- from cu
119     signal hstop      : std_logic; -- from cu
120     signal hmode_set  : std_logic; -- from a mode of modeswitch
121     signal hmode_runs : std_logic; -- from a mode of modeswitch
122     signal hrunning   : std_logic; -- to cu
123     signal hmode_seten: std_logic; -- to a mode of modeswitch
124     signal hmode_start: std_logic; -- to a mode of modeswitch
125     signal hmode_reset: std_logic; -- to a mode of modeswitch
126
127     -- component declaration ---------------------------------
128     component handlercontrol
129      Port (
130       reset     : in  std_logic;
131       clock     : in  std_logic;
132       start     : in  std_logic;
133       stop      : in  std_logic;
134       mode_set  : in  std_logic;
135       mode_runs : in  std_logic;
136       running   : out std_logic;
137       mode_seten: out std_logic;
138       mode_start: out std_logic;
139       mode_reset: out std_logic );
140     end component;
141
142     component modeswitch
143      Generic (
144       nom              : integer;
145       clock_width      : integer;
146       reset_width      : integer;
147       modus_width      : integer;
148       control_width    : integer;
149       status_width     : integer;
150       data_in_width    : integer;
151       hs_din_in_width  : integer;
152       hs_din_out_width : integer;
153       data_out_width   : integer;
154       hs_dout_in_width : integer;
155       hs_dout_out_width: integer );
156      Port (
157       clk        : in  std_logic;
158       rst        : in  std_logic;
159       modus      : in  std_logic_vector(modus_width -1 downto 0);
160       enable     : in  std_logic;
```

```
161       stop       : in std_logic;
162       modestatus : out std_logic;
163       clock      : in  std_logic_vector(clock_width-1 downto 0);
164       reset      : in  std_logic_vector(reset_width-1 downto 0);
165       control    : in  std_logic_vector(control_width-1 downto 0);
166       status     : out std_logic_vector(status_width-1 downto 0);
167       data_in    : in  std_logic_vector(data_in_width-1 downto 0);
168       hs_din_in  : in  std_logic_vector(hs_din_in_width-1 downto 0);
169       hs_din_out : out std_logic_vector(hs_din_out_width-1 downto 0);
170       data_out   : out std_logic_vector(data_out_width-1 downto 0);
171       hs_dout_in : in  std_logic_vector(hs_dout_in_width-1 downto 0);
172       hs_dout_out: out std_logic_vector(hs_dout_out_width-1 downto 0);
173       -- ports for connection to modes
174       mclock      : out std_logic_vector(((clock_width * nom)-1) downto 0);
175       mreset      : out std_logic_vector(((reset_width * nom)-1) downto 0);
176       mdata_out   : out std_logic_vector(((data_in_width * nom)-1) downto 0);
177       mhs_dout_in : in  std_logic_vector(((hs_din_out_width * nom)-1) downto 0);
178       mhs_dout_out: out std_logic_vector(((hs_din_in_width * nom)-1) downto 0);
179       mdata_in    : in  std_logic_vector(((data_out_width * nom)-1) downto 0);
180       mhs_din_in  : in  std_logic_vector(((hs_dout_out_width * nom)-1) downto 0);
181       mhs_din_out : out std_logic_vector(((hs_dout_in_width * nom)-1) downto 0);
182       mcontrol    : out std_logic_vector(((control_width * nom)-1) downto 0);
183       mstatus     : in  std_logic_vector(((status_width * nom)-1) downto 0) );
184     end component;
185
186     component modus
187       Generic (
188         -------------------------------------------------
189         -- parameter for communication with the handler
190         -------------------------------------------------
191         clock_width       : integer;
192         reset_width       : integer;
193         control_width     : integer;
194         status_width      : integer;
195         data_in_width     : integer;
196         hs_din_in_width   : integer;
197         hs_din_out_width  : integer;
198         data_out_width    : integer;
199         hs_dout_in_width  : integer;
200         hs_dout_out_width : integer );
201       Port (
202         -------------------------------------------------
203         -- ports for communication with the handler
204         -------------------------------------------------
205         clock      : in  std_logic_vector(clock_width-1 downto 0);
206         reset      : in  std_logic_vector(reset_width-1 downto 0);
207         control    : in  std_logic_vector(control_width-1 downto 0);
208         status     : out std_logic_vector(status_width-1 downto 0);
209         data_in    : in  std_logic_vector(data_in_width-1 downto 0);
```

```
210         hs_din_in  : in  std_logic_vector(hs_din_in_width-1 downto 0);
211         hs_din_out : out std_logic_vector(hs_din_out_width-1 downto 0);
212         data_out   : out std_logic_vector(data_out_width-1 downto 0);
213         hs_dout_in : in  std_logic_vector(hs_dout_in_width-1 downto 0);
214         hs_dout_out: out std_logic_vector(hs_dout_out_width-1 downto 0));
215     end component;
216 
217   ------------------------------------------------------------------
218   begin
219     -- modeswitch - handlercontrol
220     ms_enable <= hmode_seten;
221     hmode_set <= ms_status;
222     ucontrol(0) <= hmode_start;
223     ucontrol(1) <= hmode_reset;
224     hmode_runs <= ustatus(0);
225     ms_stop <= hmode_reset;
226 
227     -- modeswitch - cu-border
228     ms_clock <= cu_clock(0);
229     ms_reset <= cu_reset(0);
230     cu_status(1) <= ms_status;
231     ms_modus <= cu_control((lenofint(nom)+1) downto 2);
232     uclock <= cu_clock;
233     ureset <= cu_reset;
234 
235     -- handlercontrol - cu_border
236     hreset <= cu_reset(0);
237     hclock <= cu_clock(0);
238     hstart <= cu_control(0);
239     hstop  <= cu_control(1);
240     cu_status(0) <= hrunning;
241 
242     -- modeswitch with left and right border
243     ldata <= phi_data;
244     lhsin <= phi_hshake_in;
245     phi_hshake_out <= lhsout;
246     pho_data <= rdata;
247     rhsin <= pho_hshake_in;
248     pho_hshake_out <= rhsout;
249 
250     shctrl: handlercontrol
251         PORT MAP (
252           reset      => hreset,
253           clock      => hclock,
254           start      => hstart,
255           stop       => hstop,
256           mode_set   => hmode_set,
257           mode_runs  => hmode_runs,
258           running    => hrunning,
```

```
259           mode_seten => hmode_seten,
260           mode_start => hmode_start,
261           mode_reset => hmode_reset);
262
263       ms01: modeswitch
264         GENERIC MAP (
265           nom               => nom,
266           clock_width       => mclock_width,
267           reset_width       => mreset_width,
268           modus_width       => lenofint(nom),
269           control_width     => mcontrol_width,
270           status_width      => mstatus_width,
271           data_in_width     => phi_data_width,
272           hs_din_in_width   => phi_hshake_in_width,
273           hs_din_out_width  => phi_hshake_out_width,
274           data_out_width    => pho_data_width,
275           hs_dout_in_width  => pho_hshake_in_width,
276           hs_dout_out_width => pho_hshake_out_width )
277         PORT MAP (
278           clk         => ms_clock,
279           rst         => ms_reset,
280           modus       => ms_modus,
281           enable      => ms_enable,
282           stop        => ms_stop,
283           modestatus  => ms_status,
284           clock       => uclock,
285           reset       => ureset,
286           control     => ucontrol,
287           status      => ustatus,
288           data_in     => ldata,
289           hs_din_in   => lhsin,
290           hs_din_out  => lhsout,
291           data_out    => rdata,
292           hs_dout_in  => rhsin,
293           hs_dout_out => rhsout,
294           -- ports for connection to modes
295           mclock       => tmclock,
296           mreset       => tmreset,
297           mdata_out    => tmdata_out,
298           mhs_dout_in  => tmhs_dout_in,
299           mhs_dout_out => tmhs_dout_out,
300           mdata_in     => tmdata_in,
301           mhs_din_in   => tmhs_din_in,
302           mhs_din_out  => tmhs_din_out,
303           mcontrol     => tmcontrol,
304           mstatus      => tmstatus );
305
306       -- Modus -- the name of the modus must be modified and for
307       -- each mode there must be an own vhdl-component
```

```
308     modus01:  modus
309       GENERIC MAP (
310         clock_width       => mclock_width,
311         reset_width       => mreset_width,
312         control_width     => mcontrol_width,
313         status_width      => mstatus_width,
314         data_in_width     => phi_data_width,
315         hs_din_in_width   => phi_hshake_in_width,
316         hs_din_out_width  => phi_hshake_out_width,
317         data_out_width    => pho_data_width,
318         hs_dout_in_width  => pho_hshake_in_width,
319         hs_dout_out_width => pho_hshake_out_width )
320       PORT MAP (
321         clock      => tmclock(((mode_nr01 * mclock_width)- 1)
322                       downto ((mode_nr01 -1)* mclock_width)),
323         reset      => tmreset(((mode_nr01 * mreset_width)- 1)
324                       downto ((mode_nr01 -1)* mreset_width)),
325         control    => tmcontrol(((mode_nr01 * mcontrol_width)- 1)
326                       downto ((mode_nr01 -1)* mcontrol_width)),
327         status     => tmstatus(((mode_nr01 * mstatus_width)- 1)
328                       downto ((mode_nr01 -1)* mstatus_width)),
329         data_in    => tmdata_out(((mode_nr01 * phi_data_width)- 1)
330                       downto ((mode_nr01 -1)* phi_data_width)),
331         hs_din_in  => tmhs_dout_out(((mode_nr01 * phi_hshake_in_width)- 1)
332                       downto ((mode_nr01 -1)* phi_hshake_in_width)),
333         hs_din_out => tmhs_dout_in( ((mode_nr01 * phi_hshake_out_width)- 1)
334                       downto ((mode_nr01 -1)* phi_hshake_out_width)),
335         data_out   => tmdata_in(((mode_nr01 * pho_data_width)- 1)
336                       downto ((mode_nr01 -1)* pho_data_width)),
337         hs_dout_in => tmhs_din_out(((mode_nr01 * pho_hshake_in_width)- 1)
338                       downto ((mode_nr01 -1)* pho_hshake_in_width)),
339         hs_dout_out=> tmhs_din_in(  ((mode_nr01 * pho_hshake_out_width)- 1)
340                       downto ((mode_nr01 -1)* pho_hshake_out_width)) );
341   end Behavioral;
```

A.3.2 Protocolhandler_in (phandler_in.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   phandler_in.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9   use WORK.ifb_functions.ALL;
10
```

```
11  entity phandler_in is
12    Generic (
13      nom  : integer := 1; -- number of modes
14      ---------------------------------------------
15      -- generic parameter for PHin to CU -- all values are default
16      ---------------------------------------------
17      cu_clock_width   : integer := 1;
18      cu_reset_width   : integer := 1;
19      cu_control_width : integer := 3;
20      cu_status_width  : integer := 2;
21      cu_irq_width     : integer := 1;
22      ---------------------------------------------
23      -- generic parameter for PHin to IFB incoming borders
24      ---------------------------------------------
25      ifb_data_width       : integer := 1;
26      ifb_hshake_in_width  : integer := 1;
27      ifb_hshake_out_width : integer := 1;
28      ifb_irq_width        : integer := 1;
29      ---------------------------------------------
30      -- generic parameter for PHin to SH
31      ---------------------------------------------
32      sh_data_width       : integer := 1;
33      sh_hshake_in_width  : integer := 1;
34      sh_hshake_out_width : integer := 1 );
35    Port (
36      ---------------------------------------------
37      -- Ports for PHin to CU
38      ---------------------------------------------
39      cu_clock  : in  std_logic_vector(cu_clock_width-1 downto 0);
40      cu_reset  : in  std_logic_vector(cu_reset_width-1 downto 0);
41      cu_control: in  std_logic_vector(cu_control_width-1 downto 0);
42      cu_status : out std_logic_vector(cu_status_width-1 downto 0);
43      cu_irq    : out std_logic_vector(cu_irq_width-1 downto 0);
44      ---------------------------------------------
45      -- Ports for PHin to IFB incoming borders
46      ---------------------------------------------
47      ifb_data      : in  std_logic_vector(ifb_data_width-1 downto 0);
48      ifb_hshake_in : in  std_logic_vector(ifb_hshake_in_width-1 downto 0);
49      ifb_hshake_out: out std_logic_vector(ifb_hshake_out_width-1 downto 0);
50      ifb_irq       : in  std_logic_vector(ifb_irq_width-1 downto 0);
51      ---------------------------------------------
52      -- Ports for PHin to SH
53      ---------------------------------------------
54      sh_data      : out std_logic_vector(sh_data_width-1 downto 0);
55      sh_hshake_in : in  std_logic_vector(sh_hshake_in_width-1 downto 0);
56      sh_hshake_out: out std_logic_vector(sh_hshake_out_width-1 downto 0));
57  end phandler_in;
58
59  architecture Behavioral of phandler_in is
```

```
60    ...
61
62    -- component for irqswitch ------------------------------------------
63    component irqswitch
64      GENERIC (
65        irq_in_width  : integer;
66        irq_out_width : integer );
67      PORT (
68        irq_in : in  std_logic_vector(irq_in_width-1 downto 0);
69        irq_out: out std_logic_vector(irq_out_width-1 downto 0) );
70    end component;
71
72    ...
73  begin
74    ...
75
76    irqswitch01: irqswitch
77      GENERIC MAP (
78        irq_in_width  =>  ifb_irq_width,
79        irq_out_width => cu_irq_width )
80      PORT MAP (
81        irq_in  => ifb_irq,
82        irq_out => cu_irq );
83
84    ...
85  end Behavioral;
86
```

A.3.3 Protocolhandler_out (phandler_out.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   phandler_out.vhd
3   -- Datum:  11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9   use WORK.ifb_functions.ALL;
10
11  entity phandler_out is
12    Generic (
13      nom : integer := 1;
14      ---------------------------------------------
15      -- generic parameter for PHin to CU
16      ---------------------------------------------
17      cu_clock_width   : integer := 1;
```

```
18      cu_reset_width   : integer := 1;
19      cu_control_width : integer := 2;
20      cu_status_width  : integer := 2;
21      cu_irq_width     : integer := 1;
22      ---------------------------------------------
23      -- generic parameter for PHin to IFB incoming borders
24      ---------------------------------------------
25      ifb_data_width       : integer := 1;
26      ifb_hshake_in_width  : integer := 1;
27      ifb_hshake_out_width : integer := 1;
28      ifb_irq_width        : integer := 1;
29      ---------------------------------------------
30      -- generic parameter for PHin to SH
31      ---------------------------------------------
32      sh_data_width       : integer := 1;
33      sh_hshake_in_width  : integer := 1;
34      sh_hshake_out_width : integer := 1 );
35    Port (
36      ---------------------------------------------
37      -- Ports for PHin to CU
38      ---------------------------------------------
39      cu_clock  : in  std_logic_vector(cu_clock_width-1 downto 0);
40      cu_reset  : in  std_logic_vector(cu_reset_width-1 downto 0);
41      cu_control: in  std_logic_vector(cu_control_width-1 downto 0);
42      cu_status : out std_logic_vector(cu_status_width-1 downto 0);
43      cu_irq    : in  std_logic_vector(cu_irq_width-1 downto 0);
44      ---------------------------------------------
45      -- Ports for PHin to IFB incoming borders
46      ---------------------------------------------
47      ifb_data      : out std_logic_vector(ifb_data_width-1 downto 0);
48      ifb_hshake_in : in  std_logic_vector(ifb_hshake_in_width-1 downto 0);
49      ifb_hshake_out: out std_logic_vector(ifb_hshake_out_width-1 downto 0);
50      ifb_irq       : out std_logic_vector(ifb_irq_width-1 downto 0);
51      ---------------------------------------------
52      -- Ports for PHin to SH
53      ---------------------------------------------
54      sh_data      : in  std_logic_vector(sh_data_width-1 downto 0);
55      sh_hshake_in : in  std_logic_vector(sh_hshake_in_width-1 downto 0);
56      sh_hshake_out: out std_logic_vector(sh_hshake_out_width-1 downto 0)
57    );
58  end phandler_out;
59
60  architecture Behavioral of phandler_out is
61    ...
62
63    component irqswitch
64      GENERIC (
65        irq_in_width  : integer;
66        irq_out_width : integer );
```

```
67      PORT (
68        irq_in : in  std_logic_vector(irq_in_width-1 downto 0);
69        irq_out: out std_logic_vector(irq_out_width-1 downto 0) );
70    end component;
71
72    ...
73  begin
74    ...
75
76    irqswitch01: irqswitch
77      GENERIC MAP (
78        irq_in_width  =>  cu_irq_width,
79        irq_out_width => ifb_irq_width )
80      PORT MAP (
81          irq_in  => cu_irq,
82          irq_out => ifb_irq );
83
84     ...
85  end Behavioral;
```

A.4 Interruptswitch (intswitch.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   intswitch.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity irqswitch is
11    Generic (
12      irq_in_width  : integer := 1;
13      irq_out_width : integer := 1 );
14    Port (
15      irq_in : in  std_logic_vector(irq_in_width-1 downto 0);
16      irq_out: out std_logic_vector(irq_out_width-1 downto 0) );
17  end irqswitch;
18
19  architecture Behavioral of irqswitch is
20  begin
21
22  end Behavioral;
```

A.5 Handlercontrol (handlercontrol.vhd)

```
1  -- Author:  Marcel Flade
2  -- File:    handlercontrol.vhd
3  -- Date:    11.11.2003
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 entity handlercontrol is
11   Port (
12     reset      : in  std_logic;
13     clock      : in  std_logic;
14     start      : in  std_logic;
15     stop       : in  std_logic;
16     mode_set   : in  std_logic;
17     mode_runs  : in  std_logic;
18     running    : out std_logic;
19     mode_seten : out std_logic;
20     mode_start : out std_logic;
21     mode_reset : out std_logic );
22 end handlercontrol;
23
24 architecture Behavioral of handlercontrol is
25   TYPE states IS (waiting, modeselect, modestart, modewait);
26   signal cs, ns : states;   -- states of the FSM
27 begin
28 G: process (start, mode_set, mode_runs, cs)
29   begin
30     if cs = waiting then
31       if start = '1' and mode_set = '0' then
32         ns <= modeselect;
33       end if;
34     elsif cs = modeselect then
35       if mode_set = '1' then
36         ns <= modestart;
37       end if;
38     elsif cs = modestart then
39       if mode_runs = '1' then
40         ns <= modewait;
41       end if;
42     elsif cs = modewait then
43       if mode_runs = '0' or stop = '1' then
44         ns <= waiting;
45       end if;
46     end if;
47   end process;
```

```
48
49  SYNCH: process (clock, reset, ns)
50    begin
51      if reset = '1' then
52        cs <= waiting;
53      elsif clock'EVENT and clock='1' then
54        cs <= ns;
55      end if;
56    end process;
57
58  F: process (cs)
59    begin
60      case cs is
61        when waiting    => running    <= '0';
62                           mode_seten <= '0';
63                           mode_start <= '0';
64                           mode_reset <= '1';
65        when modeselect => running    <= '1';
66                           mode_seten <= '1';
67                           mode_start <= '0';
68                           mode_reset <= '0';
69        when modestart  => running    <= '1';
70                           mode_seten <= '0';
71                           mode_start <= '1';
72                           mode_reset <= '0';
73        when modewait   => running    <= '1';
74                           mode_seten <= '0';
75                           mode_start <= '0';
76                           mode_reset <= '0';
77        when others     => running    <= '0';
78                           mode_seten <= '0';
79                           mode_start <= '0';
80                           mode_reset <= '0';
81      end case;
82    end process;
83  end Behavioral;
```

A.6 Modeswitch (modeswitch.vhd)

```
1  -- Author: Marcel Flade
2  -- File:   modeswitch.vhd
3  -- Date:   11.11.2003
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9  use WORK.ifb_functions.ALL;
10
11 entity modeswitch is
12   Generic (
13     nom               : integer := 1; -- number of modes
14     clock_width       : integer := 1;
15     reset_width       : integer := 1;
16     modus_width       : integer := lenofint(nom);
17     control_width     : integer := 2;
18     status_width      : integer := 1;
19     data_in_width     : integer := 1;
20     hs_din_in_width   : integer := 1;
21     hs_din_out_width  : integer := 1;
22     data_out_width    : integer := 1;
23     hs_dout_in_width  : integer := 1;
24     hs_dout_out_width : integer := 1 );
25   Port (
26     clk         : in  std_logic;
27     rst         : in  std_logic;
28     modus       : in  std_logic_vector(modus_width -1 downto 0);
29     enable      : in  std_logic;
30     stop        : in  std_logic;
31     modestatus  : out std_logic;
32     clock       : in  std_logic_vector(clock_width -1 downto 0);
33     reset       : in  std_logic_vector(reset_width -1 downto 0);
34     control     : in  std_logic_vector(control_width -1 downto 0);
35     status      : out std_logic_vector(status_width -1 downto 0);
36     data_in     : in  std_logic_vector(data_in_width -1 downto 0);
37     hs_din_in   : in  std_logic_vector(hs_din_in_width -1 downto 0);
38     hs_din_out  : out std_logic_vector(hs_din_out_width -1 downto 0);
39     data_out    : out std_logic_vector(data_out_width -1 downto 0);
40     hs_dout_in  : in  std_logic_vector(hs_dout_in_width -1 downto 0);
41     hs_dout_out : out std_logic_vector(hs_dout_out_width -1 downto 0);
42     -- ports for connection to modes
43     mclock      : out std_logic_vector(((clock_width * nom)-1) downto 0);
44     mreset      : out std_logic_vector(((reset_width * nom)-1) downto 0);
45     mdata_out   : out std_logic_vector(((data_in_width * nom)-1) downto 0);
46     mhs_dout_in : in  std_logic_vector(((hs_din_out_width * nom)-1) downto 0);
47     mhs_dout_out: out std_logic_vector(((hs_din_in_width * nom)-1) downto 0);
```

```
48      mdata_in    : in  std_logic_vector(((data_out_width * nom)-1) downto 0);
49      mhs_din_in  : in  std_logic_vector(((hs_dout_out_width * nom)-1) downto 0);
50      mhs_din_out : out std_logic_vector(((hs_dout_in_width * nom)-1) downto 0);
51      mcontrol    : out std_logic_vector(((control_width * nom)-1) downto 0);
52      mstatus     : in  std_logic_vector(((status_width * nom)-1) downto 0) );
53  end modeswitch;
54
55  architecture Behavioral of modeswitch is
56    signal modereg : std_logic_vector(modus_width -1 downto 0);
57    signal modeset : std_logic;
58    signal address : integer :=0;
59
60    component g_or
61      Generic (
62        width : integer := 1 );
63      Port (
64        x : in  std_logic_vector(width -1 downto 0);
65        y : out std_logic );
66    end component;
67
68  begin
69    -- assignment of global to local signals
70    address <= vec2int(modereg);
71    modestatus <= modeset;
72
73    -- register to save the modus
74    process (clk, rst, modus, enable)
75    begin
76      IF (rst = '1') then
77        for i in modereg'LOW to modereg'HIGH loop
78          modereg(i) <= '0';
79        end loop;
80      elsif clk'EVENT and clk = '1' then
81        if stop = '1' then
82          for i in modereg'LOW to modereg'HIGH loop
83            modereg(i) <= not stop;
84          end loop;
85        elsif enable = '1' then
86          for i in modereg'LOW to modereg'HIGH loop
87            modereg(i) <= modus(i);
88          end loop;
89        end if;
90      end if;
91    end process;
92
93    -- generation of modeset
94    genor: g_or
95      GENERIC MAP (width => modus_width)
96      PORT MAP (
```

```
97          x => modereg,
98          y => modeset );
99
100     -- multiplexer for outgoing data
101     process (modeset, reset, modereg, mhs_dout_in,
102              mdata_in, mhs_din_in, mstatus, address)
103     begin
104       -- mhs_dout_in --> hs_din_out
105       -- mdata_in --> data_out
106       -- mhs_din_in --> hs_dout_out
107       -- mstatus --> status
108
109       if address > 0 and address <= nom then
110         for i in hs_din_out'RANGE loop
111           hs_din_out(i) <= mhs_dout_in((address-1)* hs_din_out'LENGTH +i);
112         end loop;
113         for i in data_out'RANGE loop
114           data_out(i) <= mdata_in((address-1)* data_out'LENGTH +i);
115         end loop;
116         for i in hs_dout_out'RANGE loop
117           hs_dout_out(i) <= mhs_din_in((address-1)* hs_dout_out'LENGTH +i);
118         end loop;
119         for i in status'RANGE loop
120           status(i) <= mstatus((address-1)* status'LENGTH +i);
121         end loop;
122       else
123         for i in hs_din_out'RANGE loop
124           hs_din_out(i) <= '0';
125         end loop;
126         for i in data_out'RANGE loop
127           data_out(i) <= '0';
128         end loop;
129         for i in hs_dout_out'RANGE loop
130           hs_dout_out(i) <= '0';
131         end loop;
132         for i in status'RANGE loop
133           status(i) <= '0';
134         end loop;
135       end if;
136     end process; -- of MUX
137
138     -- routing for clock and reset
139     process (reset, clock)
140     begin
141       for i in mreset'LOW to mreset'HIGH loop
142         mreset(i) <= reset((i mod reset'LENGTH));
143       end loop;
144       for i in mclock'LOW to mclock'HIGH loop
145         mclock(i) <= clock ((i mod clock'LENGTH));
```

```
146       end loop;
147     end process;
148
149     -- demultiplexer for incoming data
150     process (modeset, rst, modereg, control, data_in,
151              hs_din_in, hs_dout_in, address)
152     begin
153       -- control --> mcontrol
154       -- data_in --> mdata_out
155       -- hs_din_in --> mhs_dout_out
156       -- hs_dout_in --> mhs_din_out
157
158       if address > 0 and address <= nom then
159         for i in mcontrol'RANGE loop
160           if ((i >= (address-1) * control_width)
161               and (i <= address * control_width -1 )) then
162             mcontrol( i) <= control(i mod control_width);
163           else
164             mcontrol(i) <= '0';
165           end if;
166         end loop;
167         for i in mdata_out'RANGE loop
168           if ((i >= (address-1)*data_in_width)
169               and (i <= address*data_in_width -1 )) then
170             mdata_out( i) <= data_in(i mod data_in_width);
171           else
172             mdata_out(i) <= '0';
173           end if;
174         end loop;
175         for i in mhs_dout_out'RANGE loop
176           if ((i >= (address-1)*hs_din_in_width)
177               and (i <= address*hs_din_in_width -1 )) then
178             mhs_dout_out( i) <= hs_din_in(i mod hs_din_in_width);
179           else
180             mhs_dout_out(i) <= '0';
181           end if;
182         end loop;
183         for i in mhs_din_out'RANGE loop
184           if ((i >= (address-1)*hs_dout_in_width)
185               and (i <= address*hs_dout_in_width -1 )) then
186             mhs_din_out(i) <= hs_dout_in(i mod hs_dout_in_width);
187           else
188             mhs_din_out(i) <= '0';
189           end if;
190         end loop;
191
192       else
193         for i in mcontrol'RANGE loop
194           mcontrol(i) <= '0';
```

```
195         end loop;
196         for i in mdata_out'RANGE loop
197           mdata_out(i) <= '0';
198         end loop;
199         for i in mhs_dout_out'RANGE loop
200           mhs_dout_out(i) <= '0';
201         end loop;
202         for i in mhs_din_out'RANGE loop
203           mhs_din_out(i) <= '0';
204         end loop;
205       end if;
206     end process;
207
208   end Behavioral;
```

A.6.1 Generisches OR (g_or.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   g_or.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity g_or is
11    Generic (width : integer := 1 );
12    Port ( x : in  std_logic_vector(width -1 downto 0);
13           y : out std_logic );
14  end g_or;
15
16  architecture Behavioral of g_or is
17  begin
18    process (x)
19      variable tmp : std_logic;
20    begin
21      tmp := '0';
22      for i in x'LOW to x'HIGH loop
23        tmp := tmp or x(i);
24      end loop;
25      y <= tmp;
26    end process;
27
28  end Behavioral;
```

A.7 Modus (modus.vhd)

```
1  -- Author: Marcel Flade
2  -- File:   modus.vhd
3  -- Date:   11.11.2003
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 entity modus is
11   Generic (
12     clock_width       : integer := 1;
13     reset_width       : integer := 1;
14     control_width     : integer := 2;
15     status_width      : integer := 1;
16     data_in_width     : integer := 1;
17     hs_din_in_width   : integer := 1;
18     hs_din_out_width  : integer := 1;
19     data_out_width    : integer := 1;
20     hs_dout_in_width  : integer := 1;
21     hs_dout_out_width : integer := 1 );
22   Port (
23     clock       : in  std_logic_vector(clock_width -1 downto 0);
24     reset       : in  std_logic_vector(reset_width -1 downto 0);
25     control     : in  std_logic_vector(control_width -1 downto 0);
26     status      : out std_logic_vector(status_width -1 downto 0);
27     data_in     : in  std_logic_vector(data_in_width -1 downto 0);
28     hs_din_in   : in  std_logic_vector(hs_din_in_width -1 downto 0);
29     hs_din_out  : out std_logic_vector(hs_din_out_width -1 downto 0);
30     data_out    : out std_logic_vector(data_out_width -1 downto 0);
31     hs_dout_in  : in  std_logic_vector(hs_dout_in_width -1 downto 0);
32     hs_dout_out : out std_logic_vector(hs_dout_out_width -1 downto 0)
33    );
34 end modus;
35
36 architecture Behavioral of modus is
37   signal reset_from_handler : std_logic; -- control(1)
38   signal start_from_handler : std_logic; -- control(0)
39   signal running            : std_logic; -- status(0)
40 begin
41
42   reset_from_handler <= control(1); -- signal stop from handlercontrol
43   start_from_handler <= control(0); -- signal start from handlercontrol
44   status(0) <= running;
45 end Behavioral;
```

A.8 Zusatzfunktionen (ifb_functions.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   ifb_functions.vhd
3   -- Date:   11.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.all;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  package ifb_functions is
11    function vec2int(signal vec : in std_logic_vector) return integer;
12    function lenofint (a : in integer) return integer;
13  end ifb_functions;
14
15  package body ifb_functions is
16    function vec2int(signal vec : in std_logic_vector) return integer is
17      -- return the integer value of an vector
18      variable res : integer := 0;
19    begin
20      res := 0;
21        for i in vec'LOW to vec'HIGH loop
22        if vec(i) = '1' then
23          res := res + 2**i;
24        end if;
25      end loop;
26      return res;
27    end vec2int;
28
29    function lenofint (a : in integer) return integer is
30      -- returns the length of a binary encoding of a
31      variable len, tmp : integer := 0;
32    begin
33      len := 0;
34      tmp := a;
35      loop
36        tmp := tmp / 2;
37        len := len + 1;
38        exit when tmp = 0;
39      end loop;
40      return len;
41    end lenofint;
42
43  end ifb_functions;
```

B Quelltexte zum Demonstrator

Dieses Kapitel enthält die Quelltexte zur Implementierung des Demonstrators. Quelltexte die von den Templates aus Anhang A nicht abweichen, sind nicht aufgeführt. Für sie werden nur Verweise auf den entsprechenden Abschnitt in Anhang A gegeben. Die Quelltexte dieses Abschnittes sind, soweit nicht anders gekennzeichnet, in VHDL geschrieben.

B.1 Fail-safe-Interfaceblock (fs_ifb.vhd)

```
1  -- Author: Marcel Flade
2  -- File:   fs-ifb.vhd
3  -- Date:   26.11.2003
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 entity fs_ifb is
11  GENERIC (
12    clock_width : integer := 2; -- for system clock and 9600 Hz for RS232
13    reset_width : integer := 2; -- system reset and button
14    incom_data_width       : integer := 1;
15    incom_hshake_in_width  : integer := 1;
16    incom_hshake_out_width : integer := 1;
17    incom_irq_width        : integer := 1;
18    outgo_data_width       : integer := 8;
19    outgo_hshake_in_width  : integer := 4;
20    outgo_hshake_out_width : integer := 1;
21    outgo_irq_width        : integer := 1 );
22  Port (
23    -- Ports for generell control signals
24    clock : in  std_logic_vector(clock_width -1 downto 0);
25    reset : in  std_logic_vector(reset_width -1 downto 0);
26    -- Ports for incoming data
27    incom_data      : in  std_logic_vector(incom_data_width-1 downto 0);
28    incom_hshake_in : in  std_logic_vector(incom_hshake_in_width-1 downto 0);
29    incom_hshake_out: out std_logic_vector(incom_hshake_out_width-1 downto 0);
30    incom_irq       : in  std_logic_vector(incom_irq_width-1 downto 0);
31    -- Ports for outgoing data
32    outgo_data      : out std_logic_vector(outgo_data_width-1 downto 0);
33    outgo_hshake_in : in  std_logic_vector(outgo_hshake_in_width-1 downto 0);
34    outgo_hshake_out: out std_logic_vector(outgo_hshake_out_width-1 downto 0);
35    outgo_irq       : out std_logic_vector(outgo_irq_width-1 downto 0));
```

```
36   end fs_ifb;
37
38  architecture Behavioral of fs_ifb is
39    constant nom_phi : integer := 1; -- number of modes of phandler_in
40    constant nom_sh  : integer := 2; -- number of modes of shandler
41    constant nom_pho : integer := 1; -- number of modes of phandler_out
42
43    constant cuphi_clock_width   : integer := 2;
44    constant cuphi_reset_width   : integer := 1;
45    constant cuphi_control_width : integer := 3;
46    constant cuphi_status_width  : integer := 5;
47    constant cuphi_irq_width     : integer := 1;
48
49    constant cush_clock_width   : integer := 1;
50    constant cush_reset_width   : integer := 1;
51    constant cush_control_width : integer := 5;
52    constant cush_status_width  : integer := 4;
53
54    constant cupho_clock_width   : integer := 1;
55    constant cupho_reset_width   : integer := 1;
56    constant cupho_control_width : integer := 3;
57    constant cupho_status_width  : integer := 2;
58    constant cupho_irq_width     : integer := 1;
59
60    constant phish_data_width       : integer := 1;
61    constant phish_hshake_in_width  : integer := 1;
62    constant phish_hshake_out_width : integer := 1;
63
64    constant shpho_data_width       : integer := 8;
65    constant shpho_hshake_in_width  : integer := 1;
66    constant shpho_hshake_out_width : integer := 1;
67
68     -- Definition of internal signals
69    signal cuphi_clock  : std_logic_vector(cuphi_clock_width-1 downto 0);
70    signal cuphi_reset  : std_logic_vector(cuphi_reset_width-1 downto 0);
71    signal cuphi_control: std_logic_vector(cuphi_control_width-1 downto 0);
72    signal cuphi_status : std_logic_vector(cuphi_status_width-1 downto 0);
73    signal cuphi_irq    : std_logic_vector(cuphi_irq_width-1 downto 0);
74
75    signal cush_clock  : std_logic_vector(cush_clock_width-1 downto 0);
76    signal cush_reset  : std_logic_vector(cush_reset_width-1 downto 0);
77    signal cush_control: std_logic_vector(cush_control_width-1 downto 0);
78    signal cush_status : std_logic_vector(cush_status_width-1 downto 0);
79
80    signal cupho_clock  : std_logic_vector(cupho_clock_width-1 downto 0);
81    signal cupho_reset  : std_logic_vector(cupho_reset_width-1 downto 0);
82    signal cupho_control: std_logic_vector(cupho_control_width-1 downto 0);
83    signal cupho_status : std_logic_vector(cupho_status_width-1 downto 0);
84    signal cupho_irq    : std_logic_vector(cupho_irq_width-1 downto 0);
```

```
85
86      signal phish_data      : std_logic_vector(phish_data_width-1 downto 0);
87      signal phish_hshake_in : std_logic_vector(phish_hshake_in_width-1 downto 0);
88      signal phish_hshake_out: std_logic_vector(phish_hshake_out_width-1 downto 0);
89
90      signal shpho_data      : std_logic_vector(shpho_data_width-1 downto 0);
91      signal shpho_hshake_in : std_logic_vector(shpho_hshake_in_width-1 downto 0);
92      signal shpho_hshake_out: std_logic_vector(shpho_hshake_out_width-1 downto 0);
93
94      -- local signals
95      signal lclock : std_logic_vector(clock_width-1 downto 0);
96      signal lreset : std_logic_vector(reset_width-1 downto 0);
97
98      COMPONENT IBUFG
99        Port (I : in std_logic;
100           O : out std_logic
101           );
102     END COMPONENT;
103
104     COMPONENT cu
105      GENERIC(
106       clock_width : integer;
107       reset_width : integer;
108       phi_clock_width   : integer;
109       phi_reset_width   : integer;
110       phi_control_width : integer;
111       phi_status_width  : integer;
112       phi_irq_width     : integer;
113       sh_clock_width   : integer;
114       sh_reset_width   : integer;
115       sh_control_width : integer;
116       sh_status_width  : integer;
117       pho_clock_width   : integer;
118       pho_reset_width   : integer;
119       pho_control_width : integer;
120       pho_status_width  : integer;
121       pho_irq_width     : integer);
122      PORT(
123       clock : in std_logic_vector(clock_width-1 downto 0);
124       reset : in std_logic_vector(reset_width-1 downto 0);
125       phi_clock  : out std_logic_vector(phi_clock_width-1 downto 0);
126       phi_reset  : out std_logic_vector(phi_reset_width-1 downto 0);
127       phi_control: out std_logic_vector(phi_control_width-1 downto 0);
128       phi_status : in  std_logic_vector(phi_status_width-1 downto 0);
129       phi_irq    : in  std_logic_vector(phi_irq_width-1 downto 0);
130       sh_clock   : out std_logic_vector(sh_clock_width-1 downto 0);
131       sh_reset   : out std_logic_vector(sh_reset_width-1 downto 0);
132       sh_control : out std_logic_vector(sh_control_width-1 downto 0);
133       sh_status  : in  std_logic_vector(sh_status_width-1 downto 0);
```

```
134       pho_clock  : out std_logic_vector(pho_clock_width-1 downto 0);
135       pho_reset  : out std_logic_vector(pho_reset_width-1 downto 0);
136       pho_control: out std_logic_vector(pho_control_width-1 downto 0);
137       pho_status : in  std_logic_vector(pho_status_width-1 downto 0);
138       pho_irq    : out std_logic_vector(pho_irq_width-1 downto 0));
139     END COMPONENT;
140
141     COMPONENT phandler_in
142      GENERIC(
143       nom : integer;
144       cu_clock_width   : integer;
145       cu_reset_width   : integer;
146       cu_control_width : integer;
147       cu_status_width  : integer;
148       cu_irq_width     : integer;
149       ifb_data_width       : integer;
150       ifb_hshake_in_width  : integer;
151       ifb_hshake_out_width : integer;
152       ifb_irq_width        : integer;
153       sh_data_width       : integer;
154       sh_hshake_in_width  : integer;
155       sh_hshake_out_width : integer );
156      PORT(
157       cu_clock   : in  std_logic_vector(cu_clock_width-1 downto 0);
158       cu_reset   : in  std_logic_vector(cu_reset_width-1 downto 0);
159       cu_control : in  std_logic_vector(cu_control_width-1 downto 0);
160       cu_status  : out std_logic_vector(cu_status_width-1 downto 0);
161       cu_irq     : out std_logic_vector(cu_irq_width-1 downto 0);
162       ifb_data      : in  std_logic_vector(ifb_data_width-1 downto 0);
163       ifb_hshake_in : in  std_logic_vector(ifb_hshake_in_width-1 downto 0);
164       ifb_hshake_out: out std_logic_vector(ifb_hshake_out_width-1 downto 0);
165       ifb_irq       : in  std_logic_vector(ifb_irq_width-1 downto 0);
166       sh_data       : out std_logic_vector(sh_data_width-1 downto 0);
167       sh_hshake_in  : in  std_logic_vector(sh_hshake_in_width-1 downto 0);
168       sh_hshake_out : out std_logic_vector(sh_hshake_out_width-1 downto 0) );
169     END COMPONENT;
170
171     COMPONENT shandler
172      GENERIC(
173         nom : integer;
174       cu_clock_width   : integer;
175       cu_reset_width   : integer;
176       cu_control_width : integer;
177       cu_status_width  : integer;
178       phi_data_width       : integer;
179       phi_hshake_in_width  : integer;
180       phi_hshake_out_width : integer;
181       pho_data_width       : integer;
182       pho_hshake_in_width  : integer;
```

```
183       pho_hshake_out_width : integer );
184      PORT(
185       cu_clock   : in  std_logic_vector(cu_clock_width-1 downto 0);
186       cu_reset   : in  std_logic_vector(cu_reset_width-1 downto 0);
187       cu_control : in  std_logic_vector(cu_control_width-1 downto 0);
188       cu_status  : out std_logic_vector(cu_status_width-1 downto 0);
189       phi_data      : in  std_logic_vector(phi_data_width-1 downto 0);
190       phi_hshake_in : in  std_logic_vector(phi_hshake_in_width-1 downto 0);
191       phi_hshake_out: out std_logic_vector(phi_hshake_out_width-1 downto 0);
192       pho_data      : out std_logic_vector(pho_data_width-1 downto 0);
193       pho_hshake_in : in  std_logic_vector(pho_hshake_in_width-1 downto 0);
194       pho_hshake_out: out std_logic_vector(pho_hshake_out_width-1 downto 0) );
195     END COMPONENT;
196
197     COMPONENT phandler_out
198      GENERIC(
199       nom : integer;
200       cu_clock_width   : integer;
201       cu_reset_width   : integer;
202       cu_control_width : integer;
203       cu_status_width  : integer;
204       cu_irq_width     : integer;
205       ifb_data_width       : integer;
206       ifb_hshake_in_width  : integer;
207       ifb_hshake_out_width : integer;
208       ifb_irq_width        : integer;
209       sh_data_width       : integer;
210       sh_hshake_in_width  : integer;
211       sh_hshake_out_width : integer );
212      PORT(
213       cu_clock   : in  std_logic_vector(cu_clock_width-1 downto 0);
214       cu_reset   : in  std_logic_vector(cu_reset_width-1 downto 0);
215       cu_control : in  std_logic_vector(cu_control_width-1 downto 0);
216       cu_status  : out std_logic_vector(cu_status_width-1 downto 0);
217       cu_irq     : in std_logic_vector(cu_irq_width-1 downto 0);
218       ifb_data      : out  std_logic_vector(ifb_data_width-1 downto 0);
219       ifb_hshake_in : in  std_logic_vector(ifb_hshake_in_width-1 downto 0);
220       ifb_hshake_out: out std_logic_vector(ifb_hshake_out_width-1 downto 0);
221       ifb_irq       : out  std_logic_vector(ifb_irq_width-1 downto 0);
222       sh_data       : in  std_logic_vector(sh_data_width-1 downto 0);
223       sh_hshake_in  : in  std_logic_vector(sh_hshake_in_width-1 downto 0);
224       sh_hshake_out : out std_logic_vector(sh_hshake_out_width-1 downto 0) );
225     END COMPONENT;
226
227   begin
228     lreset(0) <= reset(0);
229     CLOCKBUF: IBUFG Port Map ( I => clock(0), O => lclock(0) );
230     BUTTON_BUF: IBUFG Port Map ( I => reset(1), O => lreset(1) );
231     local_reset <= lreset(0) or lreset(1);
```

```
232
233  CONTROLUNIT: cu
234      GENERIC MAP(
235        clock_width => clock_width,
236        reset_width => reset_width,
237        phi_clock_width => cuphi_clock_width,
238        phi_reset_width => cuphi_reset_width,
239        phi_control_width => cuphi_control_width,
240        phi_status_width => cuphi_status_width,
241        phi_irq_width => cuphi_irq_width,
242        sh_clock_width => cush_clock_width,
243        sh_reset_width => cush_reset_width,
244        sh_control_width => cush_control_width,
245        sh_status_width => cush_status_width,
246        pho_clock_width => cupho_clock_width,
247        pho_reset_width => cupho_reset_width,
248        pho_control_width => cupho_control_width,
249        pho_status_width => cupho_status_width,
250        pho_irq_width => cupho_irq_width )
251      PORT MAP(
252        clock => lclock,
253        reset => lreset,
254        phi_clock => cuphi_clock,
255        phi_reset => cuphi_reset,
256        phi_control => cuphi_control,
257        phi_status => cuphi_status,
258        phi_irq => cuphi_irq,
259        sh_clock => cush_clock,
260        sh_reset => cush_reset,
261        sh_control => cush_control,
262        sh_status => cush_status,
263        pho_clock => cupho_clock,
264        pho_reset => cupho_reset,
265        pho_control => cupho_control,
266        pho_status => cupho_status,
267        pho_irq => cupho_irq );
268
269  PHin: phandler_in
270      GENERIC MAP(
271        nom => nom_phi,
272        cu_clock_width => cuphi_clock_width,
273        cu_reset_width => cuphi_reset_width,
274        cu_control_width => cuphi_control_width,
275        cu_status_width => cuphi_status_width,
276        cu_irq_width => cuphi_irq_width,
277        ifb_data_width => incom_data_width,
278        ifb_hshake_in_width => incom_hshake_in_width,
279        ifb_hshake_out_width => incom_hshake_out_width,
280        ifb_irq_width => incom_irq_width,
```

```
281         sh_data_width => phish_data_width,
282         sh_hshake_in_width => phish_hshake_in_width,
283         sh_hshake_out_width => phish_hshake_out_width )
284       PORT MAP(
285         cu_clock => cuphi_clock,
286         cu_reset => cuphi_reset,
287         cu_control => cuphi_control,
288         cu_status => cuphi_status,
289         cu_irq => cuphi_irq,
290         ifb_data => incom_data,
291         ifb_hshake_in => incom_hshake_in,
292         ifb_hshake_out => incom_hshake_out,
293         ifb_irq => incom_irq,
294         sh_data => phish_data,
295         sh_hshake_in => phish_hshake_in,
296         sh_hshake_out => phish_hshake_out );
297
298   SH: shandler
299       GENERIC MAP(
300         nom => nom_sh,
301         cu_clock_width => cush_clock_width,
302         cu_reset_width => cush_reset_width,
303         cu_control_width => cush_control_width,
304         cu_status_width => cush_status_width,
305         phi_data_width => phish_data_width,
306         phi_hshake_in_width => phish_hshake_out_width,
307         phi_hshake_out_width => phish_hshake_in_width,
308         pho_data_width => shpho_data_width,
309         pho_hshake_in_width => shpho_hshake_in_width,
310         pho_hshake_out_width => shpho_hshake_out_width )
311       PORT MAP(
312         cu_clock => cush_clock,
313         cu_reset => cush_reset,
314         cu_control => cush_control,
315         cu_status => cush_status,
316         phi_data => phish_data,
317         phi_hshake_in => phish_hshake_out,
318         phi_hshake_out => phish_hshake_in,
319         pho_data => shpho_data,
320         pho_hshake_in => shpho_hshake_in,
321         pho_hshake_out => shpho_hshake_out );
322
323   PHout: phandler_out
324       GENERIC MAP(
325         nom => nom_pho,
326         cu_clock_width => cupho_clock_width,
327         cu_reset_width => cupho_reset_width,
328         cu_control_width => cupho_control_width,
329         cu_status_width => cupho_status_width,
```

```
330         cu_irq_width => cupho_irq_width,
331         ifb_data_width => outgo_data_width,
332         ifb_hshake_in_width => outgo_hshake_in_width,
333         ifb_hshake_out_width => outgo_hshake_out_width,
334         ifb_irq_width => outgo_irq_width,
335         sh_data_width => shpho_data_width,
336         sh_hshake_in_width => shpho_hshake_out_width,
337         sh_hshake_out_width => shpho_hshake_in_width )
338       PORT MAP(
339         cu_clock => cupho_clock,
340         cu_reset => cupho_reset,
341         cu_control => cupho_control,
342         cu_status => cupho_status,
343         cu_irq => cupho_irq,
344         ifb_data => outgo_data,
345         ifb_hshake_in => outgo_hshake_in,
346         ifb_hshake_out => outgo_hshake_out,
347         ifb_irq => outgo_irq,
348         sh_data => shpho_data,
349         sh_hshake_in => shpho_hshake_out,
350         sh_hshake_out => shpho_hshake_in );
351
352   end Behavioral;
```

B.2 Controlunit (cu.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   cu.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity cu is
11    GENERIC (
12      clock_width : integer := 2;
13      reset_width : integer := 2;
14      phi_clock_width   : integer := 2;
15      phi_reset_width   : integer := 1;
16      phi_control_width : integer := 3;
17      phi_status_width  : integer := 5;
18      phi_irq_width     : integer := 1;
19      sh_clock_width   : integer := 1;
20      sh_reset_width   : integer := 1;
21      sh_control_width : integer := 5;
```

```
22      sh_status_width  : integer := 4;
23      pho_clock_width   : integer := 1;
24      pho_reset_width   : integer := 1;
25      pho_control_width : integer := 3;
26      pho_status_width  : integer := 2;
27      pho_irq_width     : integer := 1 );
28    PORT (
29      clock : in std_logic_vector(clock_width -1 downto 0);
30      reset : in std_logic_vector(reset_width -1 downto 0);
31      phi_clock  : out std_logic_vector(phi_clock_width -1 downto 0);
32      phi_reset  : out std_logic_vector(phi_reset_width -1 downto 0);
33      phi_control: out std_logic_vector(phi_control_width -1 downto 0);
34      phi_status : in  std_logic_vector(phi_status_width -1 downto 0);
35      phi_irq    : in  std_logic_vector(phi_irq_width -1 downto 0);
36      sh_clock   : out std_logic_vector(sh_clock_width -1 downto 0);
37      sh_reset   : out std_logic_vector(sh_reset_width -1 downto 0);
38      sh_control : out std_logic_vector(sh_control_width -1 downto 0);
39      sh_status  : in  std_logic_vector(sh_status_width -1 downto 0);
40      pho_clock  : out std_logic_vector(pho_clock_width -1 downto 0);
41      pho_reset  : out std_logic_vector(pho_reset_width -1 downto 0);
42      pho_control: out std_logic_vector(pho_control_width -1 downto 0);
43      pho_status : in  std_logic_vector(pho_status_width -1 downto 0);
44      pho_irq    : out std_logic_vector(pho_irq_width -1 downto 0) );
45  END cu;
46
47  architecture Behavioral of cu is
48    constant system_f : integer := 20000000;
49    -- definition of local signals
50    signal lclock         : std_logic_vector(clock_width -1 downto 0);
51    signal clk9600        : std_logic;
52    signal lreset         : std_logic;
53    signal timer_overflow : std_logic;
54    signal timer_nearend  : std_logic;
55
56    signal lphi_clock  : std_logic_vector(phi_clock_width -1 downto 0);
57    signal lphi_reset  : std_logic_vector(phi_reset_width -1 downto 0);
58    signal lphi_control: std_logic_vector(phi_control_width -1 downto 0);
59    signal lphi_status : std_logic_vector(phi_status_width -1 downto 0);
60    signal lphi_irq    : std_logic_vector(phi_irq_width -1 downto 0);
61
62    signal lsh_clock  : std_logic_vector(sh_clock_width -1 downto 0);
63    signal lsh_reset  : std_logic_vector(sh_reset_width -1 downto 0);
64    signal lsh_control: std_logic_vector(sh_control_width -1 downto 0);
65    signal lsh_status : std_logic_vector(sh_status_width -1 downto 0);
66
67    signal lpho_clock  : std_logic_vector(pho_clock_width -1 downto 0);
68    signal lpho_reset  : std_logic_vector(pho_reset_width -1 downto 0);
69    signal lpho_control: std_logic_vector(pho_control_width -1 downto 0);
70    signal lpho_status : std_logic_vector(pho_status_width -1 downto 0);
```

```
71    signal lpho_irq    : std_logic_vector(pho_irq_width -1 downto 0);
72
73    COMPONENT taktgen_9600Hz is
74      Generic (Eingangs_f : integer);
75      Port    (clock : in std_logic;
76               reset : in std_logic;
77               clock9600 : out std_logic);
78    END COMPONENT;
79
80    COMPONENT timer_1sec is
81      Generic (Eingangs_f : integer;
82               near_end_def : integer );
83      Port    (reset : in std_logic;
84               clock : in std_logic;
85               newstart : in std_logic;
86               overflow : out std_logic;
87               nearend : out std_logic);
88    END COMPONENT;
89
90    COMPONENT phin_ctrl
91      Port (clock : in std_logic;
92            reset : in std_logic;
93            phin_running : in std_logic;
94            phin_modeset : in std_logic;
95            phin_stop : out std_logic;
96            phin_start : out std_logic;
97            phin_modus : out std_logic );
98    END COMPONENT;
99
100   COMPONENT phout_ctrl
101     Port (clock : in std_logic;
102           reset : in std_logic;
103           phout_running : in std_logic;
104           phout_modeset : in std_logic;
105           phout_stop : out std_logic;
106           phout_start : out std_logic;
107           phout_modus : out std_logic );
108   END COMPONENT;
109
110   COMPONENT sh_ctrl
111     Port (clock : in std_logic;
112           reset : in std_logic;
113           sh_running : in std_logic;
114           sh_modeset : in std_logic;
115           sh_stop : out std_logic;
116           sh_start : out std_logic;
117           sh_modus : out std_logic_vector(1 downto 0);
118           failure : in std_logic;
119           failure_valid : in std_logic;
```

```
120             timer_nearend : in std_logic;
121             timer_overflow : in std_logic;
122             fs_write : out std_logic;
123             fs_written : in std_logic );
124     END COMPONENT;
125
126   begin
127     lreset <= reset(0) or reset(1);
128
129     phi_clock   <= lphi_clock;
130     phi_reset(0)<= lreset;
131     phi_control <= lphi_control;
132     lphi_status <= phi_status;
133     lphi_irq    <= phi_irq;
134
135     sh_clock    <= lsh_clock;
136     sh_reset(0) <= lreset;
137     sh_control  <= lsh_control;
138     lsh_status  <= sh_status;
139
140     pho_clock   <=  lpho_clock;
141     pho_reset(0)<=  lreset;
142     pho_control <= lpho_control;
143     lpho_status <=  pho_status;
144     pho_irq     <= lpho_irq;
145
146     lphi_clock(0) <= clock(0);
147     lphi_clock(1) <= clk9600;
148     lsh_clock(0)  <= clock(0);
149     lpho_clock(0) <= clock(0);
150
151   TAKT_GEN_9600Hz: taktgen_9600Hz
152     GENERIC MAP (Eingangs_f => system_f)
153     PORT MAP    (clock => clock(0),
154                  reset => lreset,
155                  clock9600 => clk9600 );
156
157   TIMER_1SECOND: timer_1sec
158      GENERIC MAP (Eingangs_f => system_f,
159                   near_end_def => 15 )
160      PORT MAP (reset => lreset,
161                clock => clock(0),
162                newstart => lphi_status(4),
163                overflow => timer_overflow,
164                nearend => timer_nearend);
165
166   PHIN_CONTROL: phin_ctrl
167      PORT MAP (clock => clock(0),
168                reset => lreset,
```

```
169               phin_running => lphi_status(0),
170               phin_modeset => lphi_status(1),
171               phin_stop  => lphi_control(1),
172               phin_start => lphi_control(0),
173               phin_modus => lphi_control(2) );
174
175   PHOUT_CONTROL: phout_ctrl
176      PORT MAP (clock => clock(0),
177               reset => lreset,
178               phout_running => lpho_status(0),
179               phout_modeset => lpho_status(1),
180               phout_stop  => lpho_control(1),
181               phout_start => lpho_control(0),
182               phout_modus => lpho_control(2) );
183
184   SEQHANDLER_CONTROL: sh_ctrl
185      PORT MAP (clock => clock(0),
186               reset => lreset,
187               sh_running => lsh_status(0),
188               sh_modeset => lsh_status(1),
189               sh_stop  => lsh_control(1),
190               sh_start => lsh_control(0),
191               sh_modus => lsh_control(3 downto 2),
192               failure => phi_status(2),
193               failure_valid => phi_status(3),
194               timer_nearend => timer_nearend,
195               timer_overflow => timer_overflow,
196               fs_write => lsh_control(4),
197               fs_written => lsh_status(3) );
198
199   end Behavioral;
```

B.2.1 Kontrollautomat Protocolhandler_in (phin_ctrl.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   phin_ctrl.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity phin_ctrl is
11      PORT ( clock        : in std_logic;
12             reset        : in std_logic;
13             phin_running : in std_logic;
```

```
14              phin_modeset : in std_logic;
15              phin_stop    : out std_logic;
16              phin_start   : out std_logic;
17              phin_modus   : out std_logic );
18  end phin_ctrl;
19
20  architecture Behavioral of phin_ctrl is
21    type states_phinctrl is
22         (phin_Init, phin_Starting, phin_Setmode01, phin_Go);
23    signal cs, ns : states_phinctrl;
24  begin
25    G:  process (phin_running, phin_modeset, cs)
26        begin
27        if cs = phin_Init then
28          ns <= phin_Starting;
29
30        elsif cs = phin_Starting then
31          if phin_running = '1' then
32            ns <= phin_Setmode01;
33          else
34            ns <= phin_Starting;
35          end if;
36
37        elsif cs = phin_Setmode01 then
38          if phin_modeset = '1' then
39            ns <= phin_Go;
40          else
41            ns <= phin_Setmode01;
42          end if;
43
44        elsif cs = phin_Go then
45          ns <= phin_Go;
46        end if;
47      end process;
48
49    SYNCH: process (clock, reset, ns)
50        begin
51          if reset = '1' then
52            cs <= phin_Init;
53          elsif clock'EVENT and clock = '1' then
54            cs <= ns;
55          end if;
56        end process;
57
58    F:  process (cs)
59        begin
60        if cs = phin_Init then
61          phin_stop  <= '1';
62          phin_start <= '0';
```

```
63          phin_modus <= '0';
64        elsif cs = phin_Starting then
65          phin_stop  <= '0';
66          phin_start <= '1';
67          phin_modus <= '0';
68        elsif cs = phin_Setmode01 then
69          phin_stop  <= '0';
70          phin_start <= '0';
71          phin_modus <= '1';
72        elsif cs = phin_Go then
73          phin_stop  <= '0';
74          phin_start <= '0';
75          phin_modus <= '0';
76        else
77          phin_stop  <= '1';
78          phin_start <= '0';
79          phin_modus <= '0';
80        end if;
81      end process;
82
83  end Behavioral;
```

B.2.2 Kontrollautomat Sequencehandler (sh_ctrl.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   sh_ctrl.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity sh_ctrl is
11      PORT ( clock         : in std_logic;
12             reset         : in std_logic;
13             sh_running    : in std_logic;
14             sh_modeset    : in std_logic;
15             sh_stop       : out std_logic;
16             sh_start      : out std_logic;
17             sh_modus      : out std_logic_vector(1 downto 0);
18             failure       : in std_logic;
19             failure_valid : in std_logic;
20             timer_nearend : in std_logic;
21             timer_overflow: in std_logic;
22             fs_write      : out std_logic;
23             fs_written    : in std_logic );
```

```
24  end sh_ctrl;
25
26  architecture Behavioral of sh_ctrl is
27    type states_shctrl is
28         (sh_Init, sh_Starting, sh_Setmode01, sh_SetmodeFs, sh_Go, sh_Fs01);
29    signal cs, ns : states_shctrl;
30    signal failure_changed : std_logic;
31    signal failure_old : std_logic;
32    signal failure_new : std_logic;
33    signal modus : std_logic_vector(1 downto 0);
34  begin
35    sh_modus <= modus;
36    failure_changed <= (failure_new xor failure_old) or timer_nearend;
37
38    F_CHANGE: process (clock, reset, failure_valid)
39            begin
40            if reset = '1' then
41              failure_new <= '0';
42              failure_old <= '0';
43            elsif clock'EVENT and clock = '1' then
44              if failure_valid = '1' then
45                failure_old <= failure_new;
46                failure_new <= failure;
47              end if;
48            end if;
49          end process;
50
51    G:  process (sh_running, sh_modeset, cs, failure, failure_changed)
52        begin
53        if cs = sh_Init then
54          if not sh_running = '1' then
55            ns <= sh_Starting;
56          else
57            ns <= sh_Init;
58          end if;
59
60        elsif cs = sh_Starting then
61          if sh_running = '1' then
62            if failure = '0' then
63              ns <= sh_Setmode01;
64            else
65              ns <= sh_SetmodeFs;
66            end if;
67          elsif failure_changed = '1' then
68            ns <= sh_Init;
69          else
70            ns <= sh_Starting;
71          end if;
72
```

```
73          elsif cs = sh_Setmode01 then
74            if sh_modeset = '1' then
75              ns <= sh_Go;
76            elsif failure_changed = '1' then
77              ns <= sh_Init;
78            else
79              ns <= sh_Setmode01;
80            end if;
81
82          elsif cs = sh_SetmodeFs then
83            if sh_modeset = '1' then
84              ns <= sh_Fs01;
85            elsif failure_changed = '1' then
86              ns <= sh_Init;
87            else
88              ns <= sh_SetmodeFs;
89            end if;
90
91          elsif cs = sh_Go then
92            if failure_changed = '1' then
93              ns <= sh_Init;
94            else
95              ns <= sh_Go;
96            end if;
97
98          elsif cs = sh_Fs01 then
99            if failure_changed = '1' then
100             ns <= sh_Init;
101           elsif fs_written = '1' then
102             ns <= sh_Go;
103           else
104             ns <= sh_Fs01;
105           end if;
106
107         else
108           ns <= sh_Init;
109         end if;
110       end process;
111
112   SYNCH: process (clock, reset, ns)
113       begin
114         if reset = '1' then
115           cs <= sh_Init;
116         elsif clock'EVENT and clock = '1' then
117           cs <= ns;
118         end if;
119       end process;
120
121   F:  process (cs)
```

```
122         begin
123         if cs = sh_Init then
124           sh_stop  <= '1';
125           sh_start <= '0';
126           modus(0) <= '0';
127           modus(1) <= '0';
128           fs_write <= '0';
129         elsif cs = sh_Starting then
130           sh_stop  <= '0';
131           sh_start <= '1';
132           modus(0) <= '0';
133           modus(1) <= '0';
134           fs_write <= '0';
135         elsif cs = sh_Setmode01 then
136           sh_stop  <= '0';
137           sh_start <= '0';
138           modus(0) <= '1';
139           modus(1) <= '0';
140           fs_write <= '0';
141         elsif cs = sh_SetmodeFs then
142           sh_stop  <= '0';
143           sh_start <= '0';
144           modus(0) <= '0';
145           modus(1) <= '1';
146           fs_write <= '0';
147         elsif cs = sh_Go then
148           sh_stop  <= '0';
149           sh_start <= '0';
150           modus(0) <= '0';
151           modus(1) <= '0';
152           fs_write <= '0';
153         elsif cs = sh_Fs01 then
154           sh_stop  <= '0';
155           sh_start <= '0';
156           modus(0) <= '0';
157           modus(1) <= '0';
158           fs_write <= '1';
159         else
160           sh_stop  <= '1';
161           sh_start <= '0';
162           modus(0) <= '0';
163           modus(1) <= '0';
164           fs_write <= '0';
165         end if;
166       end process;
167   end Behavioral;
```

B.2.3 Kontrollautomat Protocolhandler_out (phout_ctrl.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   phout_ctrl.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity phout_ctrl is
11      PORT (clock         : in std_logic;
12            reset         : in std_logic;
13            phout_running : in std_logic;
14            phout_modeset : in std_logic;
15            phout_stop    : out std_logic;
16            phout_start   : out std_logic;
17            phout_modus   : out std_logic );
18  end phout_ctrl;
19
20  architecture Behavioral of phout_ctrl is
21    type states_phoutctrl is
22         (phout_Init, phout_Starting, phout_Setmode01, phout_Go);
23    signal cs, ns : states_phoutctrl;
24  begin
25
26    G:  process (phout_running, phout_modeset, cs)
27        begin
28        if cs = phout_Init then
29          ns <= phout_Starting;
30        elsif cs = phout_Starting then
31          if phout_running = '1' then
32            ns <= phout_Setmode01;
33          else
34            ns <= phout_Starting;
35          end if;
36        elsif cs = phout_Setmode01 then
37          if phout_modeset = '1' then
38            ns <= phout_Go;
39          else
40            ns <= phout_Setmode01;
41          end if;
42        elsif cs = phout_Go then
43          ns <= phout_Go;
44        end if;
45      end process;
46
47    SYNCH: process (clock, reset, ns)
```

```
48        begin
49          if reset = '1' then
50            cs <= phout_Init;
51          elsif clock'EVENT and clock = '1' then
52            cs <= ns;
53          end if;
54        end process;
55
56    F:  process (cs)
57        begin
58        if cs = phout_Init then
59          phout_stop  <= '1';
60          phout_start <= '0';
61          phout_modus <= '0';
62        elsif cs = phout_Starting then
63          phout_stop  <= '0';
64          phout_start <= '1';
65          phout_modus <= '0';
66        elsif cs = phout_Setmode01 then
67          phout_stop  <= '0';
68          phout_start <= '0';
69          phout_modus <= '1';
70        elsif cs = phout_Go then
71          phout_stop  <= '0';
72          phout_start <= '0';
73          phout_modus <= '0';
74        else
75          phout_stop  <= '1';
76          phout_start <= '0';
77          phout_modus <= '0';
78        end if;
79       end process;
80  end Behavioral;
```

B.2.4 Timer (timer_1sec.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   timer_1sec.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity timer_1sec is
11     GENERIC (Eingangs_f : integer := 20000000;
```

```
12              near_end_def : integer := 15 );
13      Port (reset    : in std_logic;
14            clock    : in std_logic;
15            newstart : in std_logic;
16            overflow : out std_logic;
17            nearend  : out std_logic );
18  end timer_1sec;
19
20  architecture Behavioral of timer_1sec is
21    constant Ausgangs_f : integer := 1;
22    signal count : integer;
23  begin
24
25  TIME_GEN: process(clock, reset)
26    begin
27      if reset = '1' then
28        count <= (Eingangs_f  / Ausgangs_f);
29      elsif clock'EVENT and clock = '1' then
30        if (count = 0 or newstart = '1') then
31          count <= (Eingangs_f  / Ausgangs_f);
32        else
33          count <= count - 1;
34        end if;
35      end if;
36    end process;
37
38  TIME_OUT: process(count)
39    begin
40      if count = 0 then
41        overflow <= '1';
42      else
43        overflow <= '0';
44      end if;
45
46      if count = near_end_def then
47        nearend <= '1';
48      else
49        nearend <= '0';
50      end if;
51    end process;
52  end Behavioral;
```

B.2.5 Taktgenerator (taktgen_9600Hz.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   taktgen_9600Hz.vhd
3   -- Date:   26.11.2003
```

```
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity taktgen_9600Hz is
11     GENERIC (Eingangs_f : integer := 20000000 );
12     PORT    (clock : in std_logic;
13              reset : in std_logic;
14              clock9600 : out std_logic );
15  end taktgen_9600Hz;
16
17  architecture Behavioral of taktgen_9600Hz is
18    constant Ausgangs_f : integer := 9600;
19    signal count : integer;
20    signal clk9600 : std_logic;
21    attribute clock_signal : string;
22    attribute clock_signal of clk9600: signal is "yes";
23  begin
24
25  TAKTGEN: process(clock, reset)
26    begin
27      if reset = '1' then
28        count <= (Eingangs_f  / Ausgangs_f);
29      elsif clock'EVENT and clock = '1' then
30        if count = 0 then
31          count <= (Eingangs_f  / Ausgangs_f);
32        else
33          count <= count - 1;
34        end if;
35      end if;
36    end process;
37
38  TAKTOUT: process(count)
39    begin
40      if (count < (Eingangs_f / (Ausgangs_f * 2))) then
41        clk9600 <= '1';
42      else
43        clk9600 <= '0';
44      end if;
45    end process;
46  end Behavioral;
```

B.3 Handler

Die Handler (Abschnitt A.3) und die darin enthaltenen Komponenten *Handlercontrol* und *Modeswitch* wurden nicht verändert. Der Quelltext zu den Komponenten ist in Abschnitt A.5 bzw. A.6 zu finden. Die Komponente *Interruptswitch* wurde aus dem Quelltext der Protocolhandler entfernt, da sie für die Implementierung nicht benötigt wurde.

B.4 Modus für Protocolhandler_in (mod_phi_rs232.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   mod_phi_rs232.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity mod_phi_rs232 is
11    GENERIC (
12      clock_width       : integer := 2;
13      reset_width       : integer := 1;
14      control_width     : integer := 2;
15      status_width      : integer := 4;
16      data_in_width     : integer := 1;
17      hs_din_in_width   : integer := 1;
18      hs_din_out_width  : integer := 1;
19      data_out_width    : integer := 1;
20      hs_dout_in_width  : integer := 1;
21      hs_dout_out_width : integer := 1 );
22    PORT (
23      clock      : in  std_logic_vector(clock_width-1 downto 0);
24      reset      : in  std_logic_vector(reset_width-1 downto 0);
25      control    : in  std_logic_vector(control_width-1 downto 0);
26      status     : out std_logic_vector(status_width-1 downto 0);
27      data_in    : in  std_logic_vector(data_in_width-1 downto 0);
28      hs_din_in  : in  std_logic_vector(hs_din_in_width-1 downto 0);
29      hs_din_out : out std_logic_vector(hs_din_out_width-1 downto 0);
30      data_out   : out std_logic_vector(data_out_width-1 downto 0);
31      hs_dout_in : in  std_logic_vector(hs_dout_in_width-1 downto 0);
32      hs_dout_out: out std_logic_vector(hs_dout_out_width-1 downto 0) );
33  END mod_phi_rs232;
34
35  architecture Behavioral of mod_phi_rs232 is
```

```
36    type states is
37         (waiting,start,d0,d1,d2,d3,d4,d5,d6,d7,parity_odd,stop);
38    signal cs, ns : states;
39
40    signal reset_from_handler : std_logic; -- control(1)
41    signal start_from_handler : std_logic; -- control(0)
42    signal running : std_logic;            -- status(0)
43    signal lreset : std_logic;
44    signal lclock : std_logic;
45    signal rxd : std_logic;
46    signal txd : std_logic;
47    signal d_out : std_logic;
48    signal shift : std_logic;
49    signal failure_arise : std_logic;
50    signal failure_valid : std_logic;
51    signal pg_start : std_logic;
52    signal data_arrived : std_logic;
53
54  component parity_check_odd is
55    PORT (clock : in std_logic;
56          reset : in std_logic;
57          data : in std_logic;
58          start : in std_logic;
59          parity_ok : out std_logic;
60          valid : out std_logic );
61  end component;
62
63  begin
64    reset_from_handler <= control(1); --signal stop from handlercontrol
65    start_from_handler <= control(0); --signal start from handlercontrol
66    status(0) <= running;
67    lclock <= clock(1);
68    lreset <= reset(0);
69    rxd <= hs_din_in(0);
70    hs_din_out(0) <= txd;
71    data_out(0) <= d_out;
72    hs_dout_out(0) <= shift;
73    status(1) <= not failure_arise;
74    status(2) <= failure_valid;
75    status(3) <= data_arrived and not failure_arise;
76
77  PROTOCOL_GUARD: parity_check_odd
78    PORT MAP (
79      clock => lclock,
80      reset => lreset,
81      data => rxd,
82      start => pg_start,
83      parity_ok => failure_arise,
84      valid => failure_valid );
```

```
85
86  CREATE_RUNNING: process
87      (lclock,lreset,start_from_handler,reset_from_handler)
88    begin
89      if lreset = '1' then
90        running <= '0';
91      elsif lclock'EVENT and lclock = '1' then
92        if reset_from_handler = '1' then
93          running <= '0';
94        elsif start_from_handler = '1' then
95          running <= '1';
96        end if;
97      end if;
98    end process;
99
100 SYNCH: process (lclock, lreset, rxd, running)
101   begin
102     if lreset = '1' then
103       cs <= waiting;
104     elsif lclock'EVENT and lclock = '1' then
105       if running = '1' then
106         cs <= ns;
107       end if;
108     end if;
109   end process;
110
111 F: process (rxd)
112   begin
113     if cs = waiting then
114       if rxd = '1' then
115         ns <= start;
116       else
117         ns <= waiting;
118       end if;
119     elsif cs = start then
120       ns <= d0;
121     elsif cs = d0 then
122       ns <= d1;
123     elsif cs = d1 then
124       ns <= d2;
125     elsif cs = d2 then
126       ns <= d3;
127     elsif cs = d3 then
128       ns <= d4;
129     elsif cs = d4 then
130       ns <= d5;
131     elsif cs = d5 then
132       ns <= d6;
133     elsif cs = d6 then
```

```
134         ns <= d7;
135       elsif cs = d7 then
136         ns <= parity_odd;
137       elsif cs = parity_odd then
138         ns <= stop;
139       elsif cs = stop then
140         if rxd = '0' then
141           ns <= waiting;
142         else
143           ns <= stop;
144         end if;
145       else
146         ns <= waiting;
147       end if;
148     end process;
149
150   G: process (lclock, lreset, cs)
151     begin
152       case cs is
153         when waiting => d_out <= '0';
154                         shift <= '0';
155                         pg_start <= '0';
156                         data_arrived <= '0';
157         when start => d_out <= '0';
158                       shift <= '0';
159                       pg_start <= '1';
160                       data_arrived <= '0';
161         when d0 => d_out <= rxd;
162                    shift <= '1';
163                    pg_start <= '1';
164                    data_arrived <= '0';
165         when d1 => d_out <= rxd;
166                    shift <= '0';
167                    pg_start <= '1';
168                    data_arrived <= '0';
169         when d2 => d_out <= rxd;
170                    shift <= '1';
171                    pg_start <= '1';
172                    data_arrived <= '0';
173         when d3 => d_out <= rxd;
174                    shift <= '0';
175                    pg_start <= '1';
176                    data_arrived <= '0';
177         when d4 => d_out <= rxd;
178                    shift <= '1';
179                    pg_start <= '1';
180                    data_arrived <= '0';
181         when d5 => d_out <= rxd;
182                    shift <= '0';
```

```
183                   pg_start <= '1';
184                   data_arrived <= '0';
185         when d6 => d_out <= rxd;
186                   shift <= '1';
187                   pg_start <= '1';
188                   data_arrived <= '0';
189         when d7 => d_out <= rxd;
190                   shift <= '0';
191                   pg_start <= '0';
192                   data_arrived <= '0';
193         when parity_odd => d_out <= '0';
194                            shift <= '0';
195                            pg_start <= '0';
196                            data_arrived <= '0';
197         when stop => d_out <= '0';
198                      shift <= '0';
199                      pg_start <= '0';
200                      data_arrived <= '1';
201         when others => d_out <= '0';
202                        shift <= '0';
203                        pg_start <= '0';
204                        data_arrived <= '0';
205       end case;
206     end process;
207   end Behavioral;
```

B.4.1 Paritätsprüfer (parity_check_odd.vhd)

```
1   -- Author: Marcel Flade
2   -- File:   parity_check_odd.vhd
3   -- Date:   26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity parity_check_odd is
11      PORT (clock     : in std_logic;
12            reset     : in std_logic;
13            data      : in std_logic;
14            start     : in std_logic;
15            parity_ok : out std_logic;
16            valid     : out std_logic);
17  END parity_check_odd;
18
19  architecture Behavioral of parity_check_odd is
```

```
20    type states is (waiting, s0, s1, final_ok, final_false);
21    signal cs, ns : states;
22  begin
23
24  SYNCH: process(clock, reset)
25    begin
26      if reset = '1' then
27        cs <= waiting;
28      elsif clock'EVENT and clock = '1' then
29        cs <= ns;
30      end if;
31    end process;
32
33  F: process (data, start)
34    begin
35      if cs = waiting then
36        if start = '1' then
37          if data = '0' then ns <= s0;
38          elsif data = '1' then ns <= s1;
39          end if;
40        else ns <= waiting;
41        end if;
42      elsif cs = s0 then
43        if start = '1' then
44          if data = '0' then ns <= s0;
45          elsif data = '1' then ns <= s1;
46          end if;
47        elsif start = '0' then
48          if data = '0' then ns <= final_ok;
49          elsif data = '1' then ns <= final_false;
50          end if;
51        end if;
52      elsif cs = s1 then
53        if start = '1' then
54          if data = '0' then ns <= s1;
55          elsif data = '1' then ns <= s0;
56          end if;
57        elsif start = '0' then
58          if data = '0' then ns <= final_false;
59          elsif data = '1' then ns <= final_ok;
60          end if;
61        end if;
62      elsif cs = final_ok then
63        ns <= waiting;
64      elsif cs = final_false then
65        ns <= waiting;
66      else ns <= waiting;
67      end if;
68    end process;
```

```
69
70  G: process (cs)
71    begin
72      case cs is
73        when waiting => parity_ok <= '0';
74                        valid <= '0';
75        when s0 => parity_ok <= '0';
76                   valid <= '0';
77        when s1 => parity_ok <= '0';
78                   valid <= '0';
79        when final_ok => parity_ok <= '1';
80                         valid <= '1';
81        when final_false => parity_ok <= '0';
82                            valid <= '1';
83        when others => parity_ok <= '0';
84                       valid <= '0';
85      end case;
86    end process;
87  end Behavioral;
```

B.5 Modus für Protocolhandler_out (mod_pho_epp.vhd)

```
1   -- Author:  Marcel Flade
2   -- File:    mod_pho_epp.vhd
3   -- Date:    26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10  entity mod_pho_epp is
11    GENERIC (
12      clock_width       : integer := 1;
13      reset_width       : integer := 1;
14      control_width     : integer := 2;
15      status_width      : integer := 1;
16      data_in_width     : integer := 8;
17      hs_din_in_width   : integer := 1;
18      hs_din_out_width  : integer := 1;
19      data_out_width    : integer := 8;
20      hs_dout_in_width  : integer := 4;
21      hs_dout_out_width : integer := 1 );
22    PORT (
23      clock      : in  std_logic_vector(clock_width-1 downto 0);
24      reset      : in  std_logic_vector(reset_width-1 downto 0);
25      control    : in  std_logic_vector(control_width-1 downto 0);
```

```
26      status     : out std_logic_vector(status_width-1 downto 0);
27      data_in    : in  std_logic_vector(data_in_width-1 downto 0);
28      hs_din_in  : in  std_logic_vector(hs_din_in_width-1 downto 0);
29      hs_din_out : out std_logic_vector(hs_din_out_width-1 downto 0);
30      data_out   : out std_logic_vector(data_out_width-1 downto 0);
31      hs_dout_in : in  std_logic_vector(hs_dout_in_width-1 downto 0);
32      hs_dout_out: out std_logic_vector(hs_dout_out_width-1 downto 0) );
33  end mod_pho_epp;
34
35  architecture Behavioral of mod_pho_epp is
36    TYPE protocol_states IS
37         (init,wCycle,rCycle,dwCycle,awCycle,drCycle,arCycle,cycleEnd);
38    signal cs, ns : protocol_states;
39    signal reset_from_handler : std_logic; -- control(1)
40    signal start_from_handler : std_logic; -- control(0)
41    signal running : std_logic;            -- status(0)
42    signal nWrite     : std_logic;  -- DB25: Pin 1 ; FPGA: Pin 206; IN
43    signal nDataStrobe: std_logic;  -- DB25: Pin 14; FPGA: Pin 205; IN
44    signal nAddrStrobe: std_logic;  -- DB25: Pin 17; FPGA: Pin 204; IN
45    signal nWait      : std_logic;  -- DB25: Pin 11; FPGA: Pin   3; OUT
46    signal nReset     : std_logic;  -- DB25: Pin 16; FPGA: Pin 203; IN
47    signal data       : std_logic_vector(7 downto 0);
48    signal lclock : std_logic;
49    signal lreset : std_logic;
50  begin
51
52    reset_from_handler <= control(1); -- signal stop from handlercontrol
53    start_from_handler <= control(0); -- signal start from handlercontrol
54    status(0) <= running;
55    lclock <= clock(0);
56    lreset <= reset(0);
57    nReset <= not hs_dout_in(0);
58    nDataStrobe <= not hs_dout_in(1);
59    nAddrStrobe <= not hs_dout_in(2);
60    nWrite <= not hs_dout_in(3);
61    data <= data_in;
62    data_out <= data;
63    hs_dout_out(0) <= not nWait;
64
65  CREATE_RUNNING: process
66      (lclock,lreset,start_from_handler,reset_from_handler)
67    begin
68      if lreset = '1' then
69        running <= '0';
70      elsif lclock'EVENT and lclock = '1' then
71        if reset_from_handler = '1' then
72          running <= '0';
73        elsif start_from_handler = '1' then
74          running <= '1';
```

```
75        end if;
76      end if;
77    end process;
78
79  SYNCH: process (lclock, lreset, running, nReset)
80    begin
81      if lreset = '1' then
82        cs <= init;
83      elsif lclock'EVENT and lclock = '1' then
84        if nReset = '1' then
85          cs <= init;
86        elsif running = '1' then
87          cs <= ns;
88        end if;
89      end if;
90    end process;
91
92  F: process (nWrite, nDataStrobe, nAddrStrobe, nReset, cs)
93      begin
94        if cs = init then
95          if nWrite = '0' then ns <= wCycle;
96          elsif nWrite = '1' then ns <= rCycle;
97          else ns <= init;
98          end if;
99        elsif cs = wCycle then
100         if nWrite = '1' then ns <= rCycle;
101         elsif nDataStrobe = '0' then ns <= dwCycle;
102         elsif nAddrStrobe = '0' then ns <= awCycle;
103         else ns <= wCycle;
104         end if;
105       elsif cs = rCycle then
106         if nWrite = '0' then ns <= wCycle;
107         elsif nDataStrobe = '0' then ns <= drCycle;
108         elsif nAddrStrobe = '0' then ns <= arCycle;
109         else ns <= rCycle;
110         end if;
111       elsif cs = dwCycle then
112             ns <= cycleEnd;
113       elsif cs = awCycle then
114           ns <= cycleEnd;
115       elsif cs = drCycle then
116         ns <= cycleEnd;
117       elsif cs = arCycle then
118           ns <= cycleEnd;
119       elsif cs = cycleEnd then
120         if (nDataStrobe = '1') or (nAddrStrobe = '1') then
121            ns <= init;
122         else ns <= cycleEnd;
123         end if;
```

```
124         end if;
125       end process;
126
127   G: process (cs)
128     begin
129       case cs is
130         when init     => nWait <= '0';
131         when wCycle   => nWait <= '0';
132         when rCycle   => nWait <= '0';
133         when dwCycle  => nWait <= '0';
134         when drCycle  => nWait <= '0';
135         when awCycle  => nWait <= '0';
136         when arCycle  => nWait <= '0';
137         when cycleEnd => nWait <= '1';
138         when others   => nWait <= '0';
139       end case;
140     end process;
141   end Behavioral;
```

B.6 Normalmodus für Sequencehandler (mod_sh_normal.vhd)

```
1   -- Author: Marcel Flade
2   -- File: mod_sh_normal.vhd
3   -- Date: 26.11.2003
4
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.ALL;
7   use IEEE.STD_LOGIC_ARITH.ALL;
8   use IEEE.STD_LOGIC_UNSIGNED.ALL;
9   use WORK.ifb_functions.all;
10
11  entity mod_sh_normal is
12    GENERIC (
13      clock_width       : integer := 1;
14      reset_width       : integer := 1;
15      control_width     : integer := 2;
16      status_width      : integer := 1;
17      data_in_width     : integer := 1;
18      hs_din_in_width   : integer := 1;
19      hs_din_out_width  : integer := 1;
20      data_out_width    : integer := 8;
21      hs_dout_in_width  : integer := 1;
22      hs_dout_out_width : integer := 1 );
23    PORT (
24      clock : in std_logic_vector(clock_width -1 downto 0);
25      reset : in std_logic_vector(reset_width -1 downto 0);
26      control : in std_logic_vector(control_width -1 downto 0);
```

```
27      status : out std_logic_vector(status_width -1 downto 0);
28      data_in : in std_logic_vector(data_in_width -1 downto 0);
29      hs_din_in : in std_logic_vector(hs_din_in_width -1 downto 0);
30      hs_din_out : out std_logic_vector(hs_din_out_width -1 downto 0);
31      data_out : out std_logic_vector(data_out_width -1 downto 0);
32      hs_dout_in : in std_logic_vector(hs_dout_in_width -1 downto 0);
33      hs_dout_out : out std_logic_vector(hs_dout_out_width -1 downto 0) );
34  end mod_sh_normal;
35
36  architecture Behavioral of mod_sh_normal is
37    type states is (waiting, d0, d1, d2, d3, d4, d5, d6, d7, ready);
38    signal cs, ns : states;
39    signal lclock : std_logic;
40    signal lreset : std_logic;
41    signal start_from_handler : std_logic;
42    signal reset_from_handler : std_logic;
43    signal running : std_logic;
44    signal shift : std_logic;
45    signal reg : std_logic_vector(7 downto 0);
46    signal reg_en : std_logic_vector(7 downto 0);
47    signal d_in : std_logic;
48    signal data_complete : std_logic; -- enable for sh_register
49
50  begin
51    reset_from_handler <= control(1);
52    start_from_handler <= control(0);
53    status(0) <= running;
54    lclock <= clock(0);
55    lreset <= reset(0);
56    shift <= hs_din_in(0);
57    d_in <= data_in(0);
58    data_out <= reg;
59    hs_dout_out(0) <= data_complete;
60
61  CREATE_RUNNING: process
62      (lclock,lreset,start_from_handler,reset_from_handler)
63    begin
64      if lreset = '1' then
65        running <= '0';
66      elsif lclock'EVENT and lclock = '1' then
67        if reset_from_handler = '1' then
68          running <= '0';
69        elsif start_from_handler = '1' then
70          running <= '1';
71        end if;
72      end if;
73    end process;
74
75  SYNCH: process (lclock, lreset, running)
```

```
76    begin
77      if lreset = '1' then
78        cs <= waiting;
79      elsif lclock'EVENT and lclock = '1' then
80        if running = '1' then
81          cs <= ns;
82        end if;
83      end if;
84    end process;
85
86  F: process (shift)
87    begin
88      if cs = waiting then
89        if shift = '1' then ns <= d0;
90        else ns <= waiting;
91        end if;
92      elsif cs = d0 then
93        if shift = '0' then ns <= d1;
94        else ns <= d0;
95        end if;
96      elsif cs = d1 then
97        if shift = '1' then ns <= d2;
98        else ns <= d1;
99        end if;
100     elsif cs = d2 then
101       if shift = '0' then ns <= d3;
102       else ns <= d2;
103       end if;
104     elsif cs = d3 then
105       if shift = '1' then ns <= d4;
106       else ns <= d3;
107       end if;
108     elsif cs = d4 then
109       if shift = '0' then ns <= d5;
110       else ns <= d4;
111       end if;
112     elsif cs = d5 then
113       if shift = '1' then ns <= d6;
114       else ns <= d5;
115       end if;
116     elsif cs = d6 then
117       if shift = '0' then ns <= d7;
118       else ns <= d6;
119       end if;
120     elsif cs = d7 then
121       if shift = '0' then ns <= ready;
122       else ns <= d7;
123       end if;
124     elsif cs = ready then
```

```
125         ns <= waiting;
126       else
127         ns <= waiting;
128       end if;
129     end process;
130
131   G: process (cs)
132     begin
133       case cs is
134         when waiting => reg_en <= "00000000";
135                         data_complete <= '0';
136         when d0 => reg_en <= "00000001";
137                    data_complete <= '0';
138         when d1 => reg_en <= "00000010";
139                    data_complete <= '0';
140         when d2 => reg_en <= "00000100";
141                    data_complete <= '0';
142         when d3 => reg_en <= "00001000";
143                    data_complete <= '0';
144         when d4 => reg_en <= "00010000";
145                    data_complete <= '0';
146         when d5 => reg_en <= "00100000";
147                    data_complete <= '0';
148         when d6 => reg_en <= "01000000";
149                    data_complete <= '0';
150         when d7 => reg_en <= "10000000";
151                    data_complete <= '0';
152         when ready => reg_en <= "00000000";
153                       data_complete <= '1';
154         when others => reg_en <= "00000000";
155                        data_complete <= '0';
156       end case;
157     end process;
158
159   REG_PROC: process (lclock, lreset, reg_en)
160     begin
161       if lreset = '1' then
162         reg <= "00000000";
163       elsif lclock'EVENT and lclock = '1' then
164         for i in reg'RANGE loop
165           if reg_en(i) = '1' then
166             reg(i) <= d_in;
167           end if;
168         end loop;
169       end if;
170     end process;
171   end Behavioral;
```

B.7 Fail-safe-Modus für Sequencehandler (mod_sh_failsafe.vhd)

```
1  -- Author: Marcel Flade
2  -- File:   mod_sh_failsafe.vhd
3  -- Date:   26.11.2003
4
5  library IEEE;
6  use IEEE.STD_LOGIC_1164.ALL;
7  use IEEE.STD_LOGIC_ARITH.ALL;
8  use IEEE.STD_LOGIC_UNSIGNED.ALL;
9
10 entity mod_sh_failsafe is
11   GENERIC (
12     clock_width       : integer := 1;
13     reset_width       : integer := 1;
14     control_width     : integer := 3;
15     status_width      : integer := 2;
16     data_in_width     : integer := 1;
17     hs_din_in_width   : integer := 1;
18     hs_din_out_width  : integer := 1;
19     data_out_width    : integer := 8;
20     hs_dout_in_width  : integer := 1;
21     hs_dout_out_width : integer := 1 );
22   PORT (
23     clock      : in  std_logic_vector(clock_width-1 downto 0);
24     reset      : in  std_logic_vector(reset_width-1 downto 0);
25     control    : in  std_logic_vector(control_width-1 downto 0);
26     status     : out std_logic_vector(status_width-1 downto 0);
27     data_in    : in  std_logic_vector(data_in_width-1 downto 0);
28     hs_din_in  : in  std_logic_vector(hs_din_in_width-1 downto 0);
29     hs_din_out : out std_logic_vector(hs_din_out_width-1 downto 0);
30     data_out   : out std_logic_vector(data_out_width-1 downto 0);
31     hs_dout_in : in  std_logic_vector(hs_dout_in_width-1 downto 0);
32     hs_dout_out: out std_logic_vector(hs_dout_out_width-1 downto 0) );
33 END mod_sh_failsafe;
34
35 architecture Behavioral of mod_sh_failsafe is
36   type states is (init, writeReg, wroteReg);
37   signal cs, ns : states;
38   signal lclock : std_logic;
39   signal lreset : std_logic;
40   signal start_from_handler : std_logic;
41   signal reset_from_handler : std_logic;
42   signal running : std_logic;
43   signal data : std_logic_vector(7 downto 0);
44   signal reg_enable : std_logic;
45   signal write : std_logic;
46   signal written : std_logic;
47 begin
```

```
48
49    reset_from_handler <= control(1);
50    start_from_handler <= control(0);
51    status(0) <= running;
52    lclock <= clock(0);
53    lreset <= reset(0);
54    write <= control(2);
55    status(1) <= written;
56    data_out <= data;
57    hs_dout_out(0) <= reg_enable;
58
59  CREATE_RUNNING: process
60      (lclock,lreset,start_from_handler,reset_from_handler)
61    begin
62      if lreset = '1' then
63        running <= '0';
64      elsif lclock'EVENT and lclock = '1' then
65        if reset_from_handler = '1' then
66          running <= '0';
67        elsif start_from_handler = '1' then
68          running <= '1';
69        end if;
70      end if;
71    end process;
72
73  SYNCH: process (lclock, lreset, running)
74    begin
75      if lreset = '1' then
76        cs <= init;
77      elsif lclock'EVENT and lclock = '1' then
78        if running = '1' then
79          cs <= ns;
80        end if;
81      end if;
82    end process;
83
84  F: process (write)
85    begin
86      if cs = init then
87        if write = '1' then ns <= writeReg;
88        else ns <= init;
89        end if;
90      elsif cs = writeReg then
91        ns <= wroteReg;
92      elsif cs = wroteReg then
93        if write = '0' then ns <= init;
94        else ns <= wroteReg;
95        end if;
96      else
```

```
97          ns <= init;
98        end if;
99      end process;
100
101   G: process (cs)
102     begin
103       case cs is
104         when init => reg_enable <= '0';
105                      data <= "00000000";
106                      written <= '0';
107         when writeReg => reg_enable <= '1';
108                          data <= "00000000";
109                          written <= '0';
110         when wroteReg => reg_enable <= '0';
111                          data <= "00000000";
112                          written <= '1';
113         when others => reg_enable <= '0';
114                        data <= "00000000";
115                        written <= '0';
116       end case;
117     end process;
118   end Behavioral;
```

B.8 Roboterkontrolle und Steuerung

Die nachfolgenden Quelltexte wurden in C++ geschrieben und dienen zur Steuerung und Kontrolle der Roboters. Damit die Programme funktionieren muss die Datei *inpout32.dll* im Windowssystemverzeichnis vorhanden sein. Ausführliche Informationen zum Gebrauch der Bibliothek sind auf der Webseite [web03] zu finden.

B.8.1 Robotersteuerung

```
1   // Author: Marcel Flade
2   // File:   robotsteuerung_rs232.cpp
3   // Date:   25.11.2003
4
5   #include "stdafx.h"
6   #include "stdio.h"
7   #include "stdlib.h"
8
9   short _stdcall Inp32(short PortAddress);
10  void _stdcall Out32(short PortAddress, short data);
11
```

```
12  #define BASE_COM1 0x3F8
13
14  #define MASK_DREHUNG 0xC0
15  #define MASK_GELENK1 0x30
16  #define MASK_GELENK2 0xC0
17  #define MASK_GREIFER 0x03
18
19  #define SHL_DREHUNG 6
20  #define SHL_GELENK1 4
21  #define SHL_GELENK2 2
22  #define SHL_GREIFER 0
23
24  #define MV_NULL 0x00
25  #define MV_PLUS 0x01
26  #define MV_MINU 0x02
27  #define MV_HOME 0x03
28
29  void init_com(int addr, int odd_par)  {
30    // ungerade Parität bei odd_par != 0 - kein Fehler
31    unsigned char tmp;
32
33    printf("Initialisierung von 0x%x ...", addr);
34    // Linecontrol Bit 7 setzen
35    tmp = Inp32( addr+3 );
36    tmp = tmp | 0x80;
37    Out32( addr+3, tmp);
38
39    // Geschwindigkeit einstellen
40    Out32( addr, 0x0C );
41    Out32( addr+1, 0x00 );
42
43    // Datenlänge 8 Bit, ungerade Parität, 1 Stopbit einstellen
44    if (odd_par != 0) {
45      Out32( addr+3, 0x8B);
46      printf("ungerade Parität ...");
47    }
48    else {
49      Out32( addr+3, 0xAB);
50      printf("gerade Parität ...");
51    }
52
53    // Linecontrol Bit 7 rücksetzen
54    tmp = Inp32( addr+3 );
55    tmp = tmp & 0x7F;
56    Out32( addr+3, tmp);
57
58    printf("abgeschlossen\n", addr);
59  }
60
```

```
61  void send_com(int addr, short data) {
62    printf("Sende 0x%x ...", data);
63    Out32( addr, data);
64    printf("gesendet\n");
65  }
66
67  int main(int argc, char* argv[]) {
68    char *peingabe, eingabe[100];
69    unsigned char drehung;
70    unsigned char gelenk1;
71    unsigned char gelenk2;
72    unsigned char greifer;
73    int ende = 0;
74    int fehler = '0';
75    short data = 0;
76    int i;
77
78    peingabe = eingabe;
79    init_com(BASE_COM1, fehler);
80
81    printf("Programm zur Steuerung des Kranes\n");
82    printf("-----------------------------------------------------------------\n");
83    printf("Steuerung durch Eingabe der Zeichen '0', '+', '-', 'h', 'f', 'e'\n");
84    printf("0 = keine Bewegung;  +,- = Freiheitsgrad +,- 1;  h = Homeposition;
85            f = Fehlerhafte Parität wird gesendet; e = Ende\n");
86    printf("Format: Je ein Zeichen für Drehung, Gelenk1, Gelenk2, Greifer. Bei f
87      und e egal wieviele Zeichen. Fehlerhafte Zeichen werden als 0 gewertet.\n");
88    printf("-----------------------------------------------------------------\n\n");
89
90    while (ende == 0) {
91      printf("Bitte Steuerdaten eingeben: ");
92      scanf("%s", eingabe);
93
94      switch (eingabe[0]) {
95        case 'e':
96        case 'E': ende = 1;
97              break;
98        case 'f':
99        case 'F': fehler = '1';
100             break;
101       case '0': drehung = 0x00; break;
102       case '+': drehung = 0x01; break;
103       case '-': drehung = 0x02; break;
104       case 'h':
105       case 'H': drehung = 0x03; break;
106       default : fehler = '1';
107     }
108     drehung = drehung << SHL_DREHUNG;
109
```

```
110       switch (eingabe[1]) {
111         case 'e':
112         case 'E': ende = 1;
113               break;
114         case 'f':
115         case 'F': fehler = '1';
116               break;
117         case '0': gelenk1 = 0x00; break;
118         case '+': gelenk1 = 0x01; break;
119         case '-': gelenk1 = 0x02; break;
120         case 'h':
121         case 'H': gelenk1 = 0x03; break;
122         default : fehler = '1';
123       }
124       gelenk1 = gelenk1 << SHL_GELENK1;
125
126       switch (eingabe[2]) {
127         case 'e':
128         case 'E': ende = 1;
129               break;
130         case 'f':
131         case 'F': fehler = '1';
132               break;
133         case '0': gelenk2 = 0x00; break;
134         case '+': gelenk2 = 0x01; break;
135         case '-': gelenk2 = 0x02; break;
136         case 'h':
137         case 'H': gelenk2 = 0x03; break;
138         default : fehler = '1';
139       }
140       gelenk2 = gelenk2 << SHL_GELENK2;
141
142       switch (eingabe[3]) {
143         case 'e':
144         case 'E': ende = 1;
145               break;
146         case 'f':
147         case 'F': fehler = '1';
148               break;
149         case '0': greifer = 0x00; break;
150         case '+': greifer = 0x01; break;
151         case '-': greifer = 0x02; break;
152         case 'h':
153         case 'H': greifer = 0x03; break;
154         default : fehler = '1';
155       }
156       greifer = greifer << SHL_GREIFER;
157
158       data = 0x00;
```

```
159       data = data | drehung;
160       data = data | gelenk1;
161       data = data | gelenk2;
162       data = data | greifer;
163
164       if (ende == 0) {
165         init_com(BASE_COM1, !fehler);
166         send_com(BASE_COM1, data);
167       }
168       printf("\n");
169
170       for (i=0;i++;i<100) {
171         eingabe[i] = 0x00;}
172       fehler = 0;
173     }
174     return 0;
175   }
```

B.8.2 Roboterkontrolle

```
1   // Author: Marcel Flade
2   // File:   robotctrl_epp.cpp
3   // Date:   25.11.2003
4
5   #include "stdafx.h"
6   #include "stdio.h"
7   #include "stdlib.h"
8   #include <time.h>
9   #include <sys/timeb.h>
10  #include <sys/types.h>
11
12  short _stdcall Inp32(short PortAddress);
13  void _stdcall Out32(short PortAddress, short data);
14
15  #define BASE_LPT1 0x378
16
17  #define MIN_DREHUNG 0
18  #define MAX_DREHUNG 360
19  #define MIN_GELENK1 0
20  #define MAX_GELENK1 180
21  #define MIN_GELENK2 0
22  #define MAX_GELENK2 180
23  #define MIN_GREIFER 0
24  #define MAX_GREIFER 180
25
26  #define MASK_DREHUNG 0xC0
27  #define MASK_GELENK1 0x30
```

```
28  #define MASK_GELENK2 0xC0
29  #define MASK_GREIFER 0x03
30
31  #define SHR_DREHUNG 6
32  #define SHR_GELENK1 4
33  #define SHR_GELENK2 2
34  #define SHR_GREIFER 0
35
36  #define MV_NULL 0x00
37  #define MV_PLUS 0x01
38  #define MV_MINU 0x02
39  #define MV_HOME 0x03
40
41  #define STEP 1
42  #define INTERVALL 1 // Intervall von 1 Sekunde
43
44  unsigned char get_epp() {
45    short ldata = 0xFF;
46    ldata = Inp32(BASE_LPT1+4);
47    return ldata;
48  }
49
50  int main(int argc, char* argv[]) {
51    unsigned char data, tmp;
52    int drehung = 0;  // 0..360 Grad
53    int gelenk1 = 0;  // 0..180
54    int gelenk2 = 0;  // 0..180
55    int greifer = 0;  // 0..180 Grad Oeffnungswinkel
56    unsigned char ende = 0;
57    unsigned char a;
58    int secdiff;
59    int oldsec;
60    struct tm *tnow;
61    time_t ltime;
62
63    while (ende == 0) {
64      time( &ltime);
65      tnow = localtime( &ltime);
66      printf("%2d:%2d:%2d  ",tnow->tm_hour, tnow->tm_min, tnow->tm_sec);
67      oldsec = tnow->tm_sec;
68
69      // Daten empfangen
70      data = get_epp();
71      // Daten auswerten
72      tmp = data & MASK_DREHUNG;
73      tmp = tmp >> SHR_DREHUNG;
74      switch (tmp) {
75        case MV_NULL: break;
76        case MV_PLUS: drehung = (drehung + STEP) % MAX_DREHUNG;
```

```
77                 break;
78         case MV_MINU: drehung = (drehung - STEP) % MAX_DREHUNG;
79                 break;
80         case MV_HOME: drehung = MIN_DREHUNG;
81                 break;
82       }
83
84       tmp = data & MASK_GELENK1;
85       tmp = tmp >> SHR_GELENK1;
86       switch (tmp) {
87         case MV_NULL: break;
88         case MV_PLUS: if (gelenk1 < MAX_GELENK1) gelenk1 = (gelenk1 + STEP);
89                 break;
90         case MV_MINU: if (gelenk1 < MIN_GELENK1) gelenk1 = (gelenk1 - STEP);
91                 break;
92         case MV_HOME: gelenk1 = MIN_GELENK1;
93                 break;
94       }
95
96       tmp = data & MASK_GELENK2;
97       tmp = tmp >> SHR_GELENK2;
98       switch (tmp) {
99         case MV_NULL: break;
100        case MV_PLUS: if (gelenk2 < MAX_GELENK2) gelenk2 = (gelenk2 + STEP);
101                break;
102        case MV_MINU: if (gelenk2 < MIN_GELENK2) gelenk2 = (gelenk2 - STEP);
103                break;
104        case MV_HOME: gelenk2 = MIN_GELENK2;
105                break;
106      }
107
108      tmp = data & MASK_GREIFER;
109      tmp = tmp >> SHR_GREIFER;
110      switch (tmp) {
111        case MV_NULL: break;
112        case MV_PLUS: if (greifer < MAX_GREIFER) greifer = (greifer + STEP);
113                break;
114        case MV_MINU: if (greifer < MIN_GREIFER) greifer = (greifer - STEP);
115                break;
116        case MV_HOME: greifer = MIN_GREIFER;
117                break;
118      }
119
120      printf("Aktuelle Position (D-T-S-G): %3d - %3d - %3d - %3d\n",
121              drehung, gelenk1, gelenk2, greifer);
122
123      do {
124        time(&ltime);
125        tnow = localtime (&ltime);
```

```
126         secdiff = ((tnow->tm_sec) - oldsec );
127         if (secdiff < 0) secdiff += 60;
128       } while (secdiff < INTERVALL);
129     }
130     return 0;
131   }
```

Literatur

[AG01] TTTech Computertechnik AG. *Specification of the TTP/A Protocol*, Mai 2001.

[AG02] TTTech Computertechnik AG. *Time-Triggered Protocol TTP/C High-Level Specification Document*, Juli 2002.

[Axe97] Jan Axelson. *Parallel Port Complete*. Lakeview Research, Madison, USA, 1997.

[CANa] CAN Bus Grundlagen. http://www.me-systeme.de/canbus.html. Webseite.

[CANb] CAN Protokoll. http://www.antal.de/htm/canprotokoll.htm. Webseite.

[CAN91] *CAN Specification*, September 1991. Version 2.0, http://www.microcontrol.net/download/can2spec.pdf.

[Dem93] Klaus Dembowski. *Computerschnittstellen und Bussysteme*. Markt-und-Technik-Verlag, Haar bei München, 1993.

[Els94] Jürgen Elsing. *Schnittstellen-Handbuch: verständliche Erläuterung und Benutzung von Centronics, V24, IEC-Bus*. IWT Verlag GmbH, Vaterstetten bei München, 1994. 4. Auflage.

[EPP] Warp Nine Engineering- The IEEE 1284 Experts. http://www.fapo.com/ieee1284.htm. Webseite.

[Fic03] Oliver Fick. Verschlüsselung von Parametern komplexer Schnittstellen für eingebettete Systeme auf Basis von XML. Studienarbeit, Universität Paderborn, 2003.

[FW:03] Apple. http://www.apple.com, 2003. Webseite.

[Gör89] Prof. Dr. Winfried Görke. *Fehlertolerante Rechensysteme*. R. Oldenbourg Verlag GmbH, München, 1989.

[Har03a] Harald Bögeholz, Johannes Schuster. FireWire prescht vor. *c't*, Oktober 2003. http://www.heise.de/ct/03/10/166/default.shtml .

[Har03b] Prof. Dr. Wolfram Hardt. Hardware-Software Codesign. Vorlesung, Technische Universität Chemnitz, 2003.

[Hüb01] Prof. Uwe Hübner. Rechnernetze. Vorlesung, Technische Universität Chemnitz, 2001.

[Hei95] Mathias Hein. *Ethernet: Standards, Protokolle, Komponenten*. International Thomson Publishing, Bonn, 1995.

[iDuV90] DKE Deutsche Kommision Elektrotechnik Elektronik Informationstechnik im DIN und VDE. DIN 40041 - Zuverlässigkeit; Begriffe. DIN Deutsches Institut für Normung e.V., Dezember 1990.

[iDuV02] DKE Deutsche Kommision Elektrotechnik Elektronik Informationstechnik im DIN und VDE. DIN EN 61508 VDE - Funktionale Sicherheit sicherheitsbezogener elektrische/elektronischer/programmierbarer elektronischer Systeme. DIN Deutsches Institut für Normung e.V., November 2002.

[Ihm01] Stefan Ihmor. Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter. Diplomarbeit, Universität Paderborn, 2001.

[Inc02] Digilent Inc. *Digilab 2E Reference Manual*, 14. April 2002. www.digilentinc.com.

[Jr03] Nilson Bastos Jr. Communication's System and FireWire communication Case: Interacting Robots. Universität Paderborn (C-LAB, Heinz Nixdorf Institut, IPL), 2003.

[Kop97] Hermann Kopetz. *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA, 1997.

[LVD] National Semiconductor LVDS Homepage. http://www.national.com/appinfo/lvds/. Webseite.

[Mül03] Prof. Dr. Dietmar Müller. ASIC Entwurf. Vorlesung, Technische Universität Chemnitz, 2003.

[Nau02] Dr. Bernt Naumann. Zuverlässigkeit und Diagnose digitaler Systeme. Vorlesung, Technische Universität Chemnitz, 2002.

[San] Kioshu San. Deutschsprachige Ethernet Homepage. http://kioshu.technologies.de/. Webseite.

[Sem00] National Semiconductor. *LVDS Owner's Manual*, Frühling 2000. Revision 2.0.

[Tan03] Andrew S. Tanenbaum. *Computer-Netzwerke*. Prentice Hall, 2003. 4. überarbeitete Auflage.

[Thi94] Michael Thieser. *PC-Schnittstellen*. Franzis-Verlag GmbH, München, 1994.

[TTP] TTTech - Time-Triggered Technology. http://www.tttech.com. Webseite.

[USB98] *Universal Serial Bus Specification*, September 1998. Revision 1.1.

[USB00] *Universal Serial Bus Specification*, April 2000. Revision 2.0.

[Wah98] Günter Wahl. UML kompakt. *OBJEKTspektrum*, Februar 1998.

[Wan98] Markus Wannemacher. *Das FPGA-Kochbuch*. International Thomson Publishing GmbH, Bonn, 1998.

[web03] Power tools for interfacing. http://www.logix4u.cjb.net/, 2003. Webseite.

[Wie] Adrian Wiedemann. Das ISO-OSI-Modell. http://www.osi-modell.de. Webseite.

[Xil03a] Xilinx. *Spartan-IIE 1.8V FPGA Family: Complete Data Sheet*, 9. Juli 2003. Version 2.1, http://direct.xilinx.com/bvdocs/publications/ds077.pdf.

[Xil03b] Xilinx. *Virtex-II Platform FPGAs: Complete Data Sheet*, 1. August 2003. http://direct.xilinx.com/bvdocs/publications/ds031.pdf.