Vorwort zur Wissenschaftlichen Schriftenreihe ”Eingebettete, selbstorganisierende Systeme”

Diese neu ins Leben gerufene Schriftenreihe widmet sich einer sehr aktuellen Thematik der Technischen Informatik, den eingebetteten, selbstorganisierenden Systemen (ESS). Seit Jahren durchdringen eingebettete Systeme unseren Alltag in fast allen Lebensbereichen. Angefangen von automatisierten Türöffnungssystemen, über komplex gesteuerte Servicemaschinen, z.B. Waschmaschinen, bis hin zu mobilen, persönlich zugeordneten Systemen wie Mobiltelefone und Handheld-Computer, sind eingebettete Systeme zur Selbstverständlichkeit geworden. Neue Anforderungen durch den Kunden, der in immer kürzeren Zeitintervallen Neuerungen erwartet, und steigende Festkosten für die Einrichtung einer Produktlinie haben einen neuen Aspekt in den Entwurf und Betrieb eingebetteter Systeme gebracht: *Selbstorganisation*. Einzelaspekte der Selbstorganisation können Selbstdiagnose, Selbsttest, Selbstheilung oder auch statische sowie dynamische Rekonfigurierung von Systemen sein. Dabei sind die Aspekte der Funktionalität und der Kommunikation zu unterscheiden. Beide haben großen Einfluss auf die Performanz und Stabilität eines eingebetteten Systems. Im Bereich der Kommunikation sind die Schnittstellen, die die Komponenten des eingebetteten Systems verbinden, von besonderem Interesse. Der Aspekt der rekonfigurierbaren Schnittstellen zwischen einzelnen Komponenten eingebetteter Systeme wird in diesem ersten Band aufgegriffen. Der erste Beitrag von Stefan Ihmor befasst sich mit der systematischen Modellierung von Schnittstellen. Zunächst werden aktive Tasks, Kommunikationskanäle und Kommunikationsmedien unterschieden sowie die Parameter der Zielarchitektur in den Entwurfsprozess für Schnittstellen eingebracht. Darauf aufbauend wird ein Template zur systematischen Konzeption von Schnittstellen vorgestellt und bewertet. Der zweite und dritte Beitrag werden von Marcel Flade vorgestellt. Beide Beiträge bauen auf den grundlegenden Arbeiten von Stefan Ihmor auf. Dabei betrachtet der zweite Beitrag das Problem der Rekonfigurierung von Schnittstellen zur Sicherstellung von sicherer Funktionalität, auch im Fehlerfall. Die Fehlererkennung und -behandlung werden von der Schnittstelle erkannt und ausgeführt. Im dritten Beitrag befasst sich Marcel Flade mit Schnittstellen in Hardware/Software-Systemen. Dabei stellt sich das Problem, das die Kontrolle über die Funktionalität der Schnittstelle auf den Software- und Hardwareteil aufgeteilt werden musste. Die Arbeiten zeigen eine vollständige Implementierung eines Beispiels und eines Werkzeugs zur automatisierten Generierung der VHDL-Implementierung.

Die beiden Autoren der einzelnen Beiträge, Stefan Ihmor und Marcel Flade, wurden für ihre hervorragenden Arbeiten mit Preisen der Universität Paderborn und der Technischen Universität Chemnitz ausgezeichnet.

Mit diesen ausgewählten Arbeiten, die in den ersten Band dieser wissenschaftlichen Schriftenreihe aufgenommen sind, werden dem Leser grundlegende Aspekt rekonfigurierbarer Schnittstellen für eingebettete selbstorganisierende Systeme vorgestellt und - so hoffe ich - das Interesse einer breiten Leserschaft für weitere Arbeiten und Entwicklungen im Bereich der eingebetteten selbstorganisierenden Systeme geweckt.

Den Autoren der einzelnen Beiträge, Stefan Ihmor und Marcel Flade, und allen Mitwirkenden bei der Erstellung und Umsetzung, insbesondere Ariane Schmidt für die technische Redaktion, sowie den Mitarbeiten des TUDpress Verlag danke ich für den engagierten Einsatz und die konstruktive Unterstützung.

Prof. Dr. Wolfram Hardt

November 2005

Inhaltsverzeichnis

1 **Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter**

Dipl.-Inf. Stefan Ihmor 3

2 **FPGA-basierte Fail-safe-Schnittstellen für eingebettete Systeme**

Dipl.-Inf. Marcel Flade 145

3 **Automatische Adaption von Hardware-Acceleratoren für Verhaltenssimulation**

Dipl.-Inf. Marcel Flade 196

Entwurf von Echtzeitschnittstellen am Beispiel interagierender Roboter

Dipl.-Inf. Stefan Ihmor

In diesem Beitrag wird die systematische Modellierung von Schnittstellen betrachtet. Zu Beginn erfolgt die Unterscheidung von aktiven Tasks, Kommunikationskanälen und Kommunikationsmedien. Anschließend werden die Parameter der Zielarchitektur in den Entwurfsprozess für Schnittstellen eingebracht. Anhand dieser Betrachtungen wird eine Vorlage zur Konzeption von Schnittstellen erarbeitet und bewertet.

Inhaltsverzeichnis

1 Einführung **9**
1.1 Motivation . . . 10
1.1.1 Geschwindigkeit von Schnittstellen . . . 10
1.1.2 Komplexität von Schnittstellen . . . 11
1.1.3 Schnittstellenklassen . . . 11
1.1.4 Adaption von Schnittstellen . . . 13
1.1.5 Die These . . . 14
1.2 Problemstellung . . . 15
1.2.1 Schnittstellenparameter . . . 15
1.2.2 Modellierungsansatz und Entwurfskonzept . . . 15
1.2.3 Adaption und automatisierter Entwurf . . . 15
1.2.4 Demonstrator . . . 16
1.3 Gliederung der Arbeit . . . 16

2 Stand der Technik **17**
2.1 Übertragungsmedien . . . 17
2.1.1 Verdrilltes Kupferkabel (Twisted-Pair) . . . 18
2.1.2 Koaxialkabel . . . 19
2.1.3 Glasfaserkabel . . . 19
2.1.4 Richtfunk und Satellitenfunk . . . 20
2.2 Protokolle . . . 20
2.2.1 Aufbau eines Protokolls . . . 22
2.2.2 Die Steckverbindung – eine HW-Schnittstelle . . . 27
2.2.3 Das serielle Protokoll . . . 30
2.2.4 Das parallele Protokoll . . . 34
2.2.5 Universal Serial Bus (USB) . . . 37
2.2.6 FireWire . . . 42
2.2.7 Time Triggered Protocols (TTP) . . . 47
2.3 Determinismus . . . 57
2.4 Echtzeit . . . 57
2.4.1 Definition . . . 58
2.4.2 Anwendungsbezug . . . 60
2.5 Systematisierung . . . 60
2.5.1 ISO/OSI . . . 60

2.5.2 TCP/IP 61
2.5.3 ATM 62
2.6 Bewertungskriterien 62
2.6.1 Kriterien für den Entwurf 64
2.6.2 Kriterien für den ReUse 64
2.7 Zusammenfassung 65

3 Modellierung von Echtzeitschnittstellen 67
3.1 Modellierungsansatz 68
3.2 Formale Methoden 68
3.3 Modellierung von Schnittstellen mit UML 69
3.3.1 UseCase Diagramm 70
3.3.2 Klassendiagramm (Class Diagramms) 71
3.3.3 Zustandsdiagramme (Statecharts) 72
3.3.4 Sequenzdiagramme (Sequencecharts) 74
3.3.5 Aktivitätendiagramme (Activity Diagramms) 75
3.4 Modellierungskonzept 77
3.4.1 Anforderungsanalyse 78
3.4.2 Taskanforderungen 80
3.4.3 Kanalanforderungen 83
3.4.4 Timing dependency Graph (TDG) 84
3.5 Ergebnisse 85

4 Entwurf von Echtzeitschnittstellen 87
4.1 Die Idee: Das Entwurfskonzept 88
4.1.1 Der Interface-Block (IFB) 90
4.1.2 Die Kontrolleinheit (CU) 92
4.1.3 Der Sequenzgenerator (SG) 92
4.1.4 Der Protokollgenerator (PG) 93
4.2 Implementierung der Komponenten 94
4.2.1 Bezug zur Adaptierbarkeit 95
4.2.2 Automatischer Entwurf 95
4.2.3 Ansatz einer Codegenerierung 95
4.3 Restriktionen im HW-Entwurf 96
4.4 Entwurfsbeispiel: Serielle Schnittstelle (V24) 98
4.5 Ergebnisse 106

5 Demonstrator 109
5.1 Bedeutung einer Fallstudie 109
5.2 Gesamtszenario 109
5.3 Kollisionsvermeidung 111
5.4 HW-Aufbau des Demonstrators 112
5.5 Echtzeitschnittstellen im Demonstrator 113
5.5.1 FPGA1 – Joystick (Joystickschnittstelle) 114

5.5.2 FPGA1/2 – Roboter1/2 (Serielle Schnittstelle) 116
5.5.3 FPGA2 – Hostrechner (PCI-Schnittstelle) 117
5.5.4 FPGA1 – FPGA2 (TTP) 118
5.6 Implementierung 122
5.6.1 Joystick 122
5.6.2 Serielle Schnittstelle 123
5.6.3 TTP 123
5.7 Ergebnisse 131

6 Zusammenfassung und Ausblick **133**
6.1 Zusammenfassung der Arbeit 133
6.2 Ausblick auf Modellierung und Entwurf 134
6.3 Ausblick für den Demonstrator 134

Literaturverzeichnis **137**

Abkürzungsverzeichnis **143**

1 Einführung

Diese Arbeit betrachtet den Entwurf von Echtzeitschnittstellen, der einen kritischen Teil des Gesamtentwurfs von Echtzeitsystemen ausmacht. Im Fokus der Aufmerksamkeit liegt die Untersuchung der physikalischen Ebene bis hin zum *A*pplication *P*rogrammer *I*nterface (API). Das ebenfalls wichtige Thema *G*raphical *U*ser *I*nterface (GUI), das oberhalb des API aufsetzen würde, wird in dieser Arbeit nicht mehr behandelt.

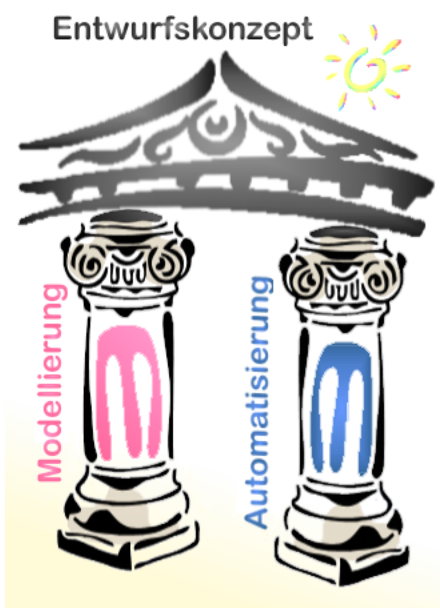


Abbildung 1.1: Schnittstellenentwurf als Säulenmodell illustriert

Um das Thema *Schnittstellenentwurf* zu illustrieren wird es durch ein *Säulenmodell* (vgl. Abbildung 1.1) veranschaulicht. Man muss sich vor Augen führen, dass sich der hier betrachtete Entwurfsprozess [Har00] in zwei Schwerpunkte gliedern lässt: *Modellierung* – und *Automatisierung*, aus denen sich das Grundgerüst des Säulenmodells, die so genannten „Säulen", zusammensetzt. Davon getragen wird das „Dach" des Modells, welches das *Entwurfskonzept* symbolisiert. Das Entwurfskonzept ist als eine Ergänzung des Entwurfsprozesses zu verstehen und befasst sich eingehend mit dem Modellierungsaspekt. Weiterhin liefert es Ansätze für einen automatisierten Entwurf.

Eine systematisierte Behandlung von *Echtzeitschnittstellen* impliziert in diesem Zusammenhang die Betrachtung von *Übertragungsmedien*, *Protokollen*, *Zeitverhalten* und *Determinismus (Vorhersagbarkeit)*.

Die *Integration* verschiedener Komponenten ist ein wesentlicher Schritt in dem *Entwurfsprozess* eines eingebetteten Systems. Häufig werden verschiedene Protokolle auf der Hardware (HW) bzw. Software (SW) -Seite verwendet, so dass eine Generierung und / oder eine Transformation der Protokolle notwendig ist. Hierfür ist eine *Automatisierung* wünschenswert, deren Grundlagen und Lösungsansätze in der Arbeit evaluiert und anhand einer Fallstudie illustriert werden.

1.1 Motivation

Um die Aktualität und Relevanz des Themas zu unterstreichen, wird das Umfeld des Schnittstellenentwurfs in den folgenden Abschnitten von mehreren Standpunkten aus analysiert.

1.1.1 Geschwindigkeit von Schnittstellen

Moor's Law stellt folgenden These auf: „Die *CPU-Leistung* steigt pro Jahr um 50%". Der *technologische Fortschritt* in der Halbleiter- und Schaltungstechnik ermöglicht es, immer mehr Gatteräquivalente auf der gleichen Fläche unterzubringen. Das ist die Grundlage, um immer aufwendigere und schnellere SW- und HW-Komponenten zu realisieren. Doch hier stößt man bereits an physikalische Grenzen. So ist für die Geschwindigkeit laut Einstein die Lichtgeschwindigkeit die absolute Obergrenze und Molekül- oder Atomgrößen stellen das untere Limit im Bereich der Granularität dar. Ebenso wie bei den CPUs findet eine kontinuierliche *Verbesserung der Übertragungsmedien* statt. Verglichen mit der Steigerung der Rechenleistung hat die Geschwindigkeit der Datenkommunikation in den letzten Jahrzehnten um Faktor 10 zugenommen [Tan96]. Die Obergrenze für *Übertragungsgeschwindigkeiten* ist aber noch nicht erreicht. So zeigen theoretische Werte, dass in Glasfaserkabel Übertragungsraten von 50.000 GByte/s (Gigabyte pro Sekunde) möglich sind. Technisch realisiert sind heute aber nur Raten von 1 GByte/s. Das liegt hauptsächlich an den Schwierigkeiten, zwischen optischen und elektrischen Signalen umzuschalten [Mik94]. Im Verlauf der Entwicklungsgeschichte ist der Trend von teueren Großrechnern zu verteilten Systemen übergegangen. Die gerade erwähnten Potentiale der Übertragungsmedien lassen darauf schließen, dass aller Voraussicht nach dieser Trend anhalten wird.

Bezogen auf diese Ausgangssituation wäre es kurzsichtig ausschließlich die Leistungssteigerungen von CPUs und Komponenten, sowie Übertragungsmedien, voranzutreiben, die Schnittstellen aber zu vernachlässigen. Leistungsstarke und flexible Schnittstellen werden benötigt, da hier sonst zwangsläufig Engpässe enstehen werden. Abbildung 1.2 demonstriert exemplarisch einige solcher Abhängigkeiten zwischen HW- und SW-Komponenten in Zusammenhang mit deren verbindenden Schnittstellen.

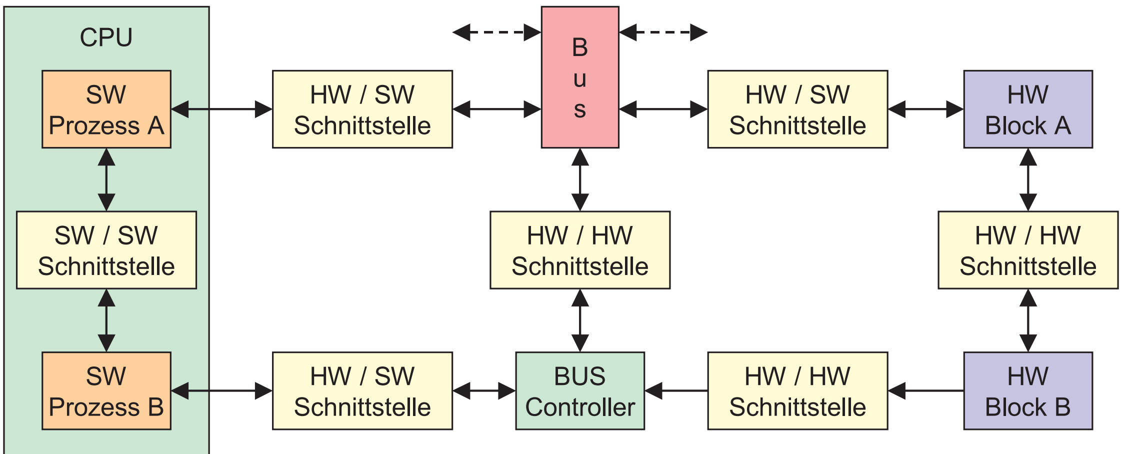


Abbildung 1.2: Informationskette und Schnittstellenklassen

1.1.2 Komplexität von Schnittstellen

Um eine konkrete Vorstellung von realistischen Größenordnungen zu bekommen, kann man sich die Annahme für I/O-Pins bei *F*ield *P*rogrammable *G*ate *A*rrays (FPGA) durch *Rent's Rule* ansehen, die besagt: $Pins \propto Gatter^f$ [1]. Der Parameter f ist $f \approx 0,5$ bei strukturierten Designs, $f > 0,5$ für Random-Logic. Also ist die Anzahl der möglichen Pins etwa proportional zur Quadrat-Wurzel der Gatteranzahl. Moderne FPGAs bestehen aus weit über 1.000.000 Gattern, was somit etwa 1.000 Pins entspricht. Der Einsatz von komplexen Komponenten bringt meistens ebenso aufwendige Schnittstellen mit sich. Diese bestehen nur selten aus einer einfachen Verdrahtung von Leitungen, sondern haben oft ein großes Maß an Eigen- und Zusatzfunktionalität. Das kostet selbstverständlich *Laufzeit* und kann in schlechten Entwürfen maßgeblich zum *kritischen Pfad* beitragen.

1.1.3 Schnittstellenklassen

Schnittstellen begegnen uns in der Informatik an vielen Stellen. Sie sind sowohl in der Hardware- als auch in der Software vertreten und sind unumgänglich, wenn Objekte miteinander verbunden werden sollen. Durch die Aufteilung eines eingebetteten Systems in HW- und SW-Komponenten enstehen drei Kombinationsmöglichkeiten für *Schnittstellenklassen*:

[1] $\propto$: proportional zu, wird abgeschätzt durch

- Software $\leftrightarrow$ Software
- Hardware $\leftrightarrow$ Hardware
- Hardware $\leftrightarrow$ Software

Beispiele für die drei Schnittstellenklassen liefert die Abbildung 1.2. SW / SW Schnittstellen können nur innerhalb einer ausführenden Einheit, wie z. B. einer CPU, zwischen zwei SW-Tasks auftreten. Die Kontaktierung zweier HW-Komponenten führt zu einer reinen HW / HW-Schnittstelle. Überall, wo SW und HW in Kontakt treten, z. B. bei der Übertragung oder Verarbeitung dynamischer Daten durch eine HW-Komponente, bezeichnet man die Berührungspunkte als HW / SW-Schnittstelle.

Schnittstellen ermöglichen die *Zerlegung* komplexer Gebilde in überschaubare Einheiten und die *Integration* von Komponenten in bestehende Entwürfe. Durch genormte Schnittstellen kann man eine Vielfalt verschiedener Objekte kapseln, die intern unterschiedlich realisiert sind, aber nach außen ein fest definiertes Verhalten vorweisen. In der Software werden Schnittstellen i. A. durch Funktionsaufrufe mit Parameterlisten oder Public-Methoden vorgegeben. Als Beispiel dafür kann man sich ein einfaches Programm vorstellen, dass Binomialkoeffizienten berechnet und dazu die Fakultät bestimmen muss. Die mathematische Gleichung wird folgendermaßen beschrieben:

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

Der Algorithmus dazu könnte dann wie in Abbildung 1.3 aussehen:

```
BINOMIAL(INT m,n)
INT bk := 0;
BEGIN
  IF (m >= n) THEN
    BEGIN
      bk := FAK(m) / (FAK(n) * FAK(m-n));
    END;
  RETURN(bk);
END;
```

Abbildung 1.3: Berechnung von Binomialkoeffizienten

An dieser Stelle nutzt man die Möglichkeit Programmcode durch eine Funktionsaufruf zu integrieren, der die Fakultätsberechnung übernimmt. Eingangswert ist dann die Zahl, zu der die Fakultät zu berechnen ist, Ausgangswert ist die Fakultät selbst. Wie der Algorithmus aber intern arbeitet, ob er z. B. iterativ (vgl. Abbildung 1.4) oder rekursiv (vgl. Abbildung 1.5) programmiert ist, spielt dabei für den Anwender keine Rolle, die Schnittstelle ist die gleiche. Der Datenaustausch erfolgt über Speicherzellen mit einem

```
FAK (INT num)
INT fakultaet := 1;
BEGIN
  WHILE (num > 1) DO
    BEGIN
      fakultaet := fakultaet * num;
      num := num -1;
    END;
  RETURN (fakultaet);
END;
```

Abbildung 1.4: Berechnung der Fakultät, iterativer Algorithmus

```
FAK (INT num)
INT fakultaet;
BEGIN
  IF (num = 1) THEN
    fakultaet := 1;
  ELSE
    fakultaet := num * FAK(num-1);
  RETURN (fakultaet);
END;
```

Abbildung 1.5: Berechnung der Fakultät, rekursiver Algorithmus

Bussystem als Kommunikationsweg. Als Protokoll kann man die Parametersyntax und -Semantik ansehen.

In der Hardware können Analogien zur Software festgestellt werden. Hier spricht man z. B. von einer seriellen-, einer PS2- oder einer Funk-Maus, unabhängig ob die Maus ihre Positionserfassung mechanisch oder optisch vornimmt. Bereits durch diese einfachen Beispiele wird ersichtlich wie wichtig Schnittstellen sind und wie häufig man ihnen bewusst oder unbewusst begegnet.

1.1.4 Adaption von Schnittstellen

Im Anwendungsbereich der *eingebetteten Systeme (ES)* werden besonders die *HW/SW-* und *HW/HW-Schnittstellen* betrachtet, da typischerweise HW-Komponenten im Design integriert sind. Systeme die aus verschiedenen Komponenten bestehen werden als *heterogene Systeme* bezeichnet. Dies bringt eine Vielfalt an Schnittstellen mit sich und führt zum Problem der *Adaption* der einzelnen Komponenten. Wie schon allgemein bei Schnittstellen, so sind es auch bei der Adaption folgende Gesichtspunkte, die zu beachten sind:

- Übertragungsmedien,
- Protokolle,
- Zeitverhalten,
- Determinusmus (Vorhersagbarkeit) und
- zur Hilfestellung: Einordnung in bekannte Modelle

Häufig bestimmt nicht nur die Funktion, sondern auch die Adaptierbarkeit der zu integrierenden Komponenten deren Auswahl und ist somit ein entscheidendes Kriterium. Oft ist die Adaption mit der Erfahrung und der Vorkenntnis des Entwicklers verbunden und kann ein kritischer Punkt im Entwurfszyklus sein. In enger Verwandschaft damit steht der Gesichtspunkt der *Wiederverwendbarkeit* (engl.: ReUse) von Komponenten. Genormte und einfach zu erzeugende Schnittstellen fördern die Wiederverwendung von Komponenten und erhöhen die *Effektivität* und *Qualität* der Entwurfsarbeit.

1.1.5 Die These

Obige Kapitel verdeutlichen, dass eine *Systematisierung* und *Automatisierung* im Entwurf und der Adaption von Schnittstellen erforderlich ist. Das legt die Entwicklung eines *Entwurfskonzepts* (vgl. Abbildung 1.1) nahe. Zur Begründung seien noch einmal die bisher aufgeführten Entwurfsprobleme zusammengefasst:

- Die hohen Rechenleistungen der Prozessoren und die Transferraten der Übertragungsmedien benötigen leistungsstarke Schnittstellen. Das bedingt die Anforderungen an den Durchsatz sowie das Zeitverhalten einer Schnittstelle.
- Die wachsende Pinanzahl der HW-Komponenten erhöht die Komplexität einer Schnittstelle. Die Anzahl der Pins spiegelt sich in der Kanalanzahl und Kanalbandbreite (ggf. Multiplexverfahren) wieder.
- Wohldefinierte Schnittstellen erleichtern die Einordnung und Handhabbarkeit der rasch zunehmenden Anzahl von Komponenten. Sie spielen eine wichtige Rolle für die Adaption. Weiterhin verringert eine systematische und automatisierbare Entwurfsmethodik die Entwurfszeit, sowie entstehende Entwurfskosten.
- Die Wiederverwendung und Integration von Komponenten erhöht die Effektivität und Qualität der Entwurfsarbeit. Daher sollte eine Adaption möglichst einfach sein. Ein einheitlicher Entwurfsprozess erleichtert dies und ist Grundlage der angestrebten Automatisierung.

Ziel der Arbeit ist es, eine tragfähige Basis für ein Entwurfskonzept für Schnittstellen zu entwickeln und an Beispielen zu erproben. Die eben erwähnten Aspekte sollen dabei berücksichtigt werden.

1.2 Problemstellung

Diese Arbeit liefert einen Beitrag zur *Methodik* des *Entwurfs* und der *Automatisierung* von Schnittstellen. Aus dem bisher analysierten Problemfeld ergeben sich die in Abschnitt 1.2.1 bis 1.2.4 näher erläuterten Kernpunkte. Es werden dabei existierende Technologien, sowie Werkzeuge zu Beschreibung von Modellen, z. B. UML (Unified Modelling Language), eingesetzt und ein *Entwurfskonzept* entwickelt.

1.2.1 Schnittstellenparameter

Zu den Schnittstellenparametern zählen die *Übertragungsmedien, Protokolle, Zeitverhalten* und *Determinismus (Vorhersagbarkeit)*. Dabei ist zu klären welche Medien geeignet sind und mit welchen Protokollen sie betrieben werden können. Ebenso soll festgestellt werden, wie das Zeitverhalten und die Vorhersagbarkeit in den Schnittstellenentwurf eingehen und modelliert werden können. Diese Arbeit beschränkt sich auf die Behandlung von *Echtzeitkommunikation*, d. h. nur echtzeitfähige Schnittstellen werden analysiert. Das stellt aber keine wirkliche Einschränkung dar, weil Echtzeitschnittstellen eine Untermenge aller Schnittstellen sind, jedoch muss dieser Schnittstellentyp exakten Zeitvorgaben genügen. Die Entwurfsmethodik ist also auch auf allgemeine Schnittstellen übertragbar, sofern die Restriktion der Vorhersagbarkeit aufgehoben wird.

1.2.2 Modellierungsansatz und Entwurfskonzept

Unter diesem Gesichtspunkt sind die Ansätze dafür darzustellen, wie Echtzeitschnittstellen modelliert werden können. Das Ergebnis wird in Form eines *Modellierungskonzepts* formuliert. Ausgehend von den sehr abstrakten Anforderungen wird so eine systematische Verfeinerung des Modells ermöglicht. Ziel der Verfeinerung ist es, eine Basis für den automatisierten Entwurfsprozess zu bieten. Dieses Vorgehen bezeichnet man auch als Top-Down Strukturierung und gliedert die jeweils betrachtete Ebene der Schnittstellen in lösbare Teilprobleme. Unter Zuhilfenahme dieses Modellierungskonzepts ist dann ein *Entwurfskonzept* zu entwickeln, das eine Behandlung der betrachteten Schnittstellenklasse ermöglicht. Das Entwurfskonzept dient als Grundlage für eine Automatisierung und wird an konkreten Beispielen im Demonstrator validiert.

1.2.3 Adaption und automatisierter Entwurf

Ein weiterer Punkt der Betrachtung stellt die *Adaption* von aktiven HW / SW-Komponenten an gegebene Übertragungsmedien (-kanäle) dar. Im Entwurfsprozess erfolgt zunächst die Auswahl geeigneter Medien und Komponenten. Dann müssen notwendige Adaptionen durch Schnittstellen vorgenommen werden, wobei das in der Modellierung erarbeitete Entwurfskonzept Anwendung findet. Auch soll das Konzept, in Hinblick auf einen späteren automatisierten Entwurf, entsprechend ausgelegt sein.

1.2.4 Demonstrator

An einem *Demonstrator* soll das Entwurfskonzept an verschiedenen HW / SW-Schnittstellen illustriert werden. Hierfür ist ein entsprechendes *Szenario* zu konstruieren und in Hardware aufzubauen. Der Einsatz von Rapid-Prototyping Boards macht eine *Synthese* erforderlich. Als Programmiersprache wird VHDL verwendet.

1.3 Gliederung der Arbeit

Die Arbeit befasst sich mit dem Thema: *Entwurf von Echtzeitschnittstellen.* Kapitel 1 bietet eine Einführung in aktuelle Entwurfsprobleme. Kapitel 2 geht näher auf die Betrachteten Echtzeitschnittstellen ein und umreißt den aktuellen Stand der Technik in Bezug auf wichtige Schnittstellenparameter. Vorgestellt werden Übertragungsmedien, und deren Protokolle. Gleichzeitig werden Restriktionen angegeben, um diese echtzeitfähig zu betreiben. Dabei spielen das Zeitverhalten und der Determinismus eine entscheidende Rolle. Im weiteren wird eine Systematisierung bestehender Modelle gegeben und eine Angabe von Entwurfskriterien gemacht. In Kapitel 3 werden Modellierungansätze von Echtzeitschnittstellen beschrieben. Dabei wird ausführlich auf das Modellierungskonzept und dessen Bedeutung eingegangen. Das nachfolgenden Kapitel 4 befasst sich mit dem Entwurfsaspekt. Hier wird das Entwurfskonzept vorgestellt und mit Hilfe der Modellierungswerkzeuge beschrieben. An dieser Stelle wird auch der Bezug zu der Adaption von Komponenten und einer Automatisierung hergestellt. In Kapitel 5 wird dann das Entwurfskonzept am Demonstrator, der ausführlich vorgestellt wird, validiert. Kapitel 6 fasst die Ergebnisse zusammen, bevor ein Ausblick auf Zukunftsperspektiven und Weiterentwicklungsmöglichkeiten in Kapitel 7 gegeben wird.

2 Stand der Technik

Dieses Kapitel vermittelt einen Einblick in den aktuellen Stand der Technik. Es werden die aus Kapitel 1 bekannten Schnittstellenparameter näher erläutert. Gleichzeitig dient es als Grundlage zum Verständnis der späteren Kapitel. Die folgenden Punkte sind Teil der Vorüberlegungen beim Entwurf des Demonstrators (siehe Kapitel 5) und finden sich auch dort wieder.

2.1 Übertragungsmedien

Das Versenden von digitalen Daten, sog. Bitströmen, erfolgt mittels *Übertragungsmedien*, auch *Kommunikationskanäle* genannt. Für die eigentliche Übertragung können verschiedene Medien eingesetzt werden. Jedes Medium wird durch spezifische *Kanaleigenschaften*, wie z. B. *Bandbreite, Latenz (Verzögerung), Kosten, Wartungs-* und *Installationsaufwand* charakterisiert [Tan96] (vgl. Abbildung 2.1).

MEDIUM		Distanz	Übertragungsgeschwindigkeit
Verdrillte Kupferkabel		5 km	150 Mbit/s
Richtfunk		10 km	140 Mbit/s
Satellitenfunk		10.000 km	2 Gbit/s
Koaxialkabel		3 km	800 Mbit/s
Glasfaserkabel		30 km	2 Gbit/s

Abbildung 2.1: Übertragungsmedien im Überblick

2.1.1 Verdrilltes Kupferkabel (Twisted-Pair)

Die in der Technik verwendeten verdrillten Kupferkabel (engl.: Twisted-Pair Cable) sind das weitaus verbreitetste Übertragungsmedium für die Individualkommunikation. Über zwei Kupferleiter (0,6 mm Durchmesser) erfolgt die Datenübertragung auf elektrischem Wege. Die Drähte sind miteinander verdrillt, um soweit wie möglich gegenseitige Störungen benachbarter Adern innerhalb eines Kabels (sogenanntes Nebensprechen) auszuschließen.

Solche Kabel sind billig und einfach zu verlegen. Auf der anderen Seite bieten nur sie eine geringe (schmale) Bandbreite und eine relativ hohe Störanfälligkeit, z. B. durch elektromagnetische Wellen. Auch eine geringe Abhörsicherheit ist ein negativer Aspekte dieser Kabelart.

Mit der Entwicklung von lokalen Netzen auf Basis verdrillter Kupferkabel wurden auch die Qualitätsanforderungen höher. 1991 publizierte die Electronic Industry Association (EIA) einen Standard, der die Grundlage für den Einsatz von nicht-abgeschirmten verdrillten Kupferkabeln in der Datenübertragung bildete. Die nicht-abgeschirmten verdrillten Kupferkabel (engl.: Unshielded Twisted-Pair; abgekürzt UTP) wurden in dem Standard in fünf Kategorien eingeteilt, welche die maximale Übertragungsgeschwindigkeit festlegen (vgl. Tabelle 2.1).

Tabelle 2.1: EIA-Standards für UTP-Verkabelung [Fla98]

Verkabelung	Einsatzbereich
Kategorie 1	Telefonie und schmalbandige Datenübertragung
Kategorie 2	ISDN und T1-Leitungen. ISDN ermöglicht Datenübertagung bei 64 kbit/s. T1 ist ein Datenübertragungsdienst in den USA mit einer Geschwindigkeit von 1,5 MBit/s
Kategorie 3	Daten bis zu 16 MHz. Einsatz in den LAN-Verfahren 10BaseT bei 10 MBit/s und 100BaseT4 bei 100 MBit/s.
Kategorie 4	Daten bis zu 20 MHz. Einsatz bei den LAN-Standards Tokenring mit 16 MBit/s und 100BaseT4 bei 100 MBit/s
Kategorie 5	Daten bis zu 100MHz. Grundlage der LAN-Standards 100BaseTX und 100BaseT4, beide mit einer Übertragungsgeschwindigkeit von 100 MBit/s.

Anforderungen an ein Twisted-Pair Kabel: Das Kabel muss eine Impedanz von 100 Ω haben und mindestens 6 mal pro Meter verdrillt sein. Der Gleichstromwiderstand darf auf 330 m 28.6 Ω nicht überschreiten. Die maximale Abschwächung des Signals darf auf 330 m nicht größer als 16 dB bei einer Frequenz von 5MHz sein ($16dB = \frac{1}{40}$ der Ausgangsleistung) [Fla98]. Diese Werte wurden praxisnah ermittelt um eine möglichst stabile Übertragungsgüte zu gewährleisten und sind speziell auf die Kanaleigenschaften eines Twisted-Pair Kabel abgestimmt.

2.1.2 Koaxialkabel

In einem Koaxialkabel (engl.: Coaxial Cable) von meist fünf bis zehn Millimeter Durchmesser sind zwei Kupferleiter ineinanderliegend (koaxial) angeordnet. In der Mitte eines hohlen Außenleiters (Masse) befindet sich, getrennt durch eine Isolierschicht, der Innenleiter (Signal). Die Datenübertragung erfolgt auf elektrischem Wege und ist besser abgeschirmt als vergleichsweise beim Twisted-Pair Kabel [Ric97]. Trotzdem hat das Koaxialkabel hat in letzter Zeit an Wichtigkeit verloren.

Anforderungen an ein Koaxial Kabel: Koaxialkabel haben 50 Ω Impedanz und können bei Längen von 1 km noch bis zu 1 bis 2 GByte/s betrieben werden. Bei zunehmender Länge sinkt dann die Bandbreite.

2.1.3 Glasfaserkabel

In Glasfaserkabeln (engl.: Optical Fiber Cable) erfolgt die Informationsübertragung durch Glasfasern mittels extrem kurzer Laserlichtimpulse (im Nanosekundenbereich) mit hoher Impulsrate (Bandbreite bis zu mehreren GHz). Ein Glasfaserkabel (vgl. Abbildung 2.2) oder Lichtwellenleiter ist eine sehr feine zylindrische Faser aus hochreinem Silikatglas [Ric97]. Sie besteht aus einem Kern mit einer Umhüllung aus Glas, wobei der Mantel eine geringere Dichte als der Kern aufweist, so dass es an der Grenzfläche zur Totalreflexion kommt, die für die Lichtausbreitung erforderlich ist. Um das Glas befindet sich eine Ummantelung aus Kunststoff.

Es existieren zwei Arten von Lichtwellenleitern (LWL): Multimodefasern, bei denen viele diskrete Wellen zur Signalübertragung dienen und Monomodefasern, bei denen nur eine einzige Welle zur Signalübertragung verwendet wird. Durch den Einsatz von LWL können beträchtliche Entfernungen (vgl. Abbildung 2.1) überbrückt werden.

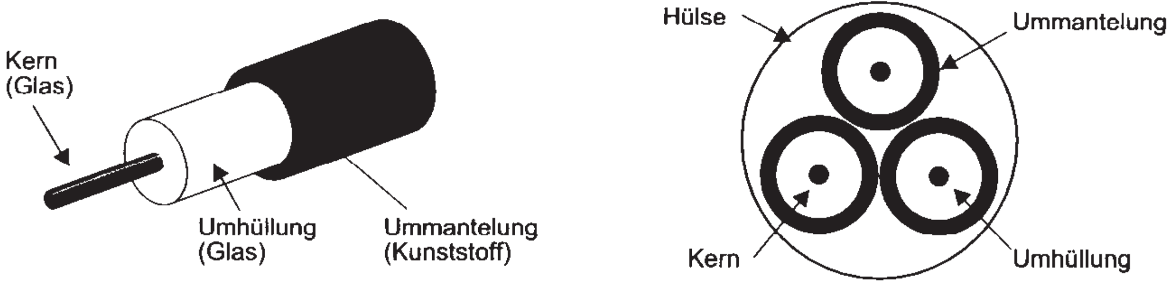


Abbildung 2.2: Aufbau eines Glasfaserkabels

Anforderungen an ein Lichtwellenleiter: Der Kern hat bei Multimodefasern einen Durchmesser von 50 μm. Das ist etwa die Stärke eines menschlichen Haares. Einzelmodefasern dagegen haben nur einen Durchmesser von etwa 8 bis 10 μm [Tan96].

2.1.4 Richtfunk und Satellitenfunk

Funkverkehr wird i. A. zu den aerischen Medien gezählt, im Gegensatz zu den bisher erwähnten terrestrischen Kanälen. In ES kommen häufig Infrarot- sowie Ultraschallsysteme zum Einsatz. Navigationssysteme, die auf GPS (Global Positioning System) basieren, sind dagegen ein Beispiel für ein ES, das Satellitenkommunikation benutzt. Der Vorteil des Funkverkehrs ist die große Reichweite und dass er kein Medium benutzt. Jedoch sind diese Verfahren ziemlich störanfällig und können leicht abgehört werden. Ein weiterer Nachteil ist der hohe Energiebedarf solcher Systeme.

2.2 Protokolle

Zur Übermittlung von Daten über ein Medium wird ein *Protokoll* benötigt. Bevor die ausgewählten Protokolle vorgestellt werden, wird zunächst eine Definition des Begriffs und eine Einordnung in das Umfeld der Datenübertragung gegeben. Alle Modelle, die Kommunikation in hierarchisch aufeinander aufbauende Schichten einteilen (vgl. Abbildung 2.3), haben eins gemeinsam: Einer entsprechenden Schicht wird immer ein *Dienst* und ein *Protokoll* zugeordnet. Diese beiden Begriffe werden manchmal fälschlicherweise gleichgesetzt, haben aber voneinander getrennte Bedeutungen. Ein Dienst läuft immer über die Schnittstelle zwischen den Schichten ab und gibt wieder, welche Operationen der *Dienstanbieter* dem *Dienstnutzer* zur Verfügung stellt. Die Implementierung bleibt dabei hinter der Schnittstelle des Dienstes verborgen.

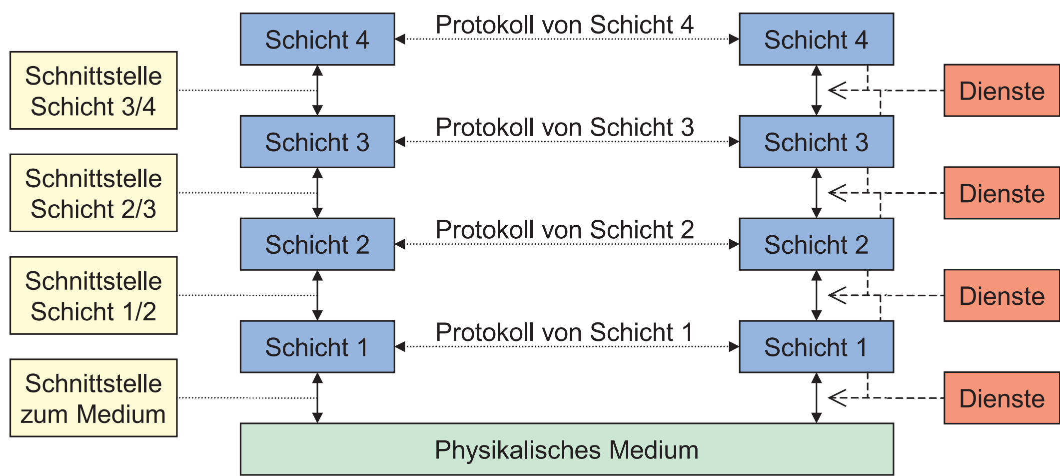


Abbildung 2.3: Allgemeines Schichtenmodell

Ein Protokoll dagegen dient der, auf gleicher Schicht stattfindenden, Kommunikation zwischen zwei Instanzen. Alle Rahmen, Pakete oder Nachrichten die zwischen Sender und Empfänger ausgetauscht werden erhalten durch das Protokoll eine Syntax und Semantik, d. h. ein Format und eine Bedeutung. Eine Instanz nutzt in Abhängigkeit von

ihrem Protokoll die daraus resultierenden Dienste des Dienstanbieters. Das stellt die Unabhängigkeit zwischen Dienst und Protokoll sicher.

Die eigentliche Kommunikation vom Sender zu Empfänger verläuft also über die Dienste des Senders bis hinab zur physikalischen Schicht (vgl. Abbildung 2.4). Hier erfolgt die Datenübertragung als Bitstrom zum Empfänger. Dort werden die Daten auf der Empfängerseite wieder bis zur äquivalenten Senderschicht durch die Dienste hochgereicht.

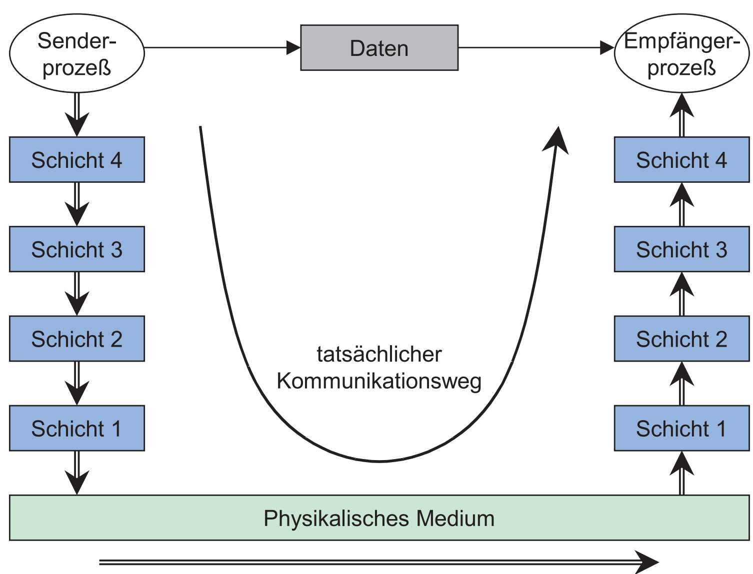


Abbildung 2.4: Kommunikationsweg im Schichtenmodell

Das in Abbildung 2.3 vorgestellte Modell dient als Grundlage der meisten Kommunikationsverfahren. Ein Datentransfer kann auf viele Arten stattfinden, z. B. über ein Netzwerk, wie es bei LANs (Local Area Network) oder dem Internet realisiert ist, oder durch eine Direktverkabelung, wie es bei Nullmodemkabel oder einem Drucker vorliegt. Je nach Anwendungsgebiet und Realisierung sind die Modelle unterschiedlich komplex und haben eine verschiedene Anzahl von Schichten. Alle Modelle haben dabei gemeinsam, dass mindestens die *physikalische Schicht* implementiert werden muss, was meistens in HW erfolgt, und dass jede Schicht ihr eigenes Protokoll besitzt.

Diese Arbeit befasst sich aber weniger mit *Netzwerkprotokollen*, sondern geht vielmehr auf die in ES verwendeten *Datenübertragungsprotokolle* ein. Die bisher erklärten Begriffe haben auch hier Gültigkeit. Technisch relevante Echtzeitprotokolle werden im Folgenden genauer analysiert. Den Protokollen wird jeweils eine Anzahl von Medien zugeordnet, auf denen sie implementiert sind. Die Übertragungsmedien sind bereits in Abschnitt 2.1 erklärt. Häufig ermöglichte ein technischer Fortschritt der Übertragungsmedien die Adaption oder Neuentwicklung medienabhängiger Protokolle. Ein protokollspezifischer Teil der Übertragungsmedien sind die Steckverbindungen. Deshalb werden diese bei der Einführung der Protokolle erklärt.

2.2.1 Aufbau eines Protokolls

Eine einheitliche Beschreibung aller Protokolle ist schwierig, besonders da in komplexen Modellen mehrere Protokolle zum Einsatz kommen, für die je nach Auftreten unterschiedliche Anforderungen und Parameter bestimmend sind. Wie bereits erwähnt, ist in Schichtenmodellen zu jeder Schicht ein Protokoll erforderlich. Die Kommunikation innerhalb eines ES wird ebenso wie bei Netzwerken durch Protokolle gesteuert. Einige Merkmale sind dabei identisch, andere finden hier keine Verwendung. Da ES meist heterogen sind, werden Komponenten miteinander verbunden, die nicht baugleich sind. Dies stellt besondere Herausforderungen an die verwendeten Protokolle und Schnittstellen.

Um dennoch den Entwurf handhaben zu können, ist es erforderlich charakteristische Merkmale (vgl. Abbildung 2.5) zu extrahieren, auf die der Entwurf aufbaut. Die so ermittelten *Parameter* lassen sich auch in den Netzwerkprotokollen der unteren Schichten wiederfinden. Man kann eine Einteilung in HW-Parameter und SW-Parameter treffen, deren Bedeutung im folgenden Abschnitt erläutert wird. Nachfolgend werden dann die aufgeführten Protokolle, bezogen auf diese Parameter, betrachtet.

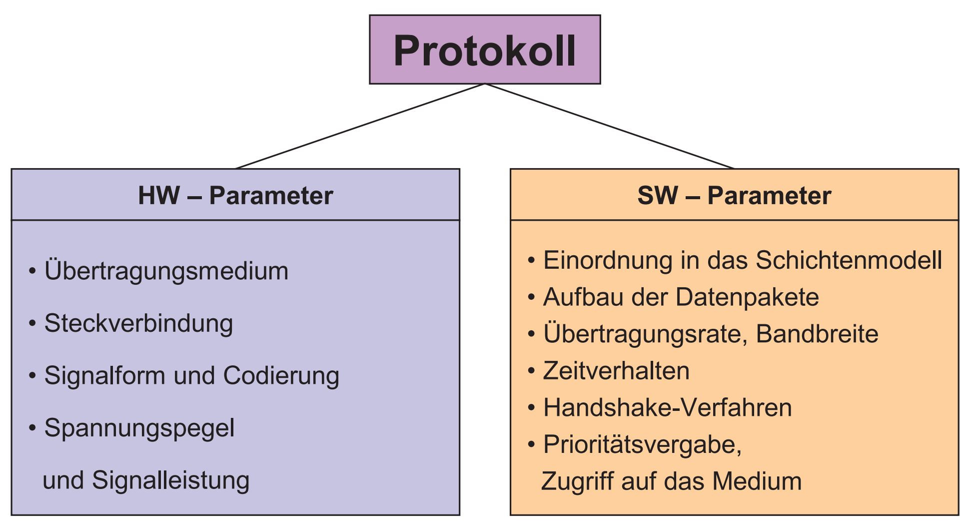


Abbildung 2.5: Wichtige Parameter eines Protokolls

Übertragungsmedium

Die für diese Arbeit wichtigen *Übertragungsmedien* wurden in Kapitel 2.1 bereits vorgestellt. Es erfolgt eine Zuordnung der Protokolle zu den in der Praxis verwendeten Medien. Dazu gehört auch die Auswahl der entsprechenden Stecker.

Steckverbindungen

Eine *Steckverbindung* besteht aus einem Stecker und der dazu gehörigen Buchse. Mit dem Stecker, bzw. der Buchse, verbindet man eine vordefinierte Belegung der Leitungen, ab-

hängig vom jeweiligen Protokoll. Die Abbildung der einzelnen Stecker veranschaulicht so die Signale eines Protokolls. Das bedeutet, ein Stecker repräsentiert nichts anderes, als die HW-Schnittstelle eines Protokolls der physikalischen Schicht. Weiterhin können Steckverbindungen hilfreich sein, die vorgeschriebene Topologie eines Netzwerkes einzuhalten.

Signalform und Codierung

Jedes Signal, das auf der Leitung übertragen wird, kann durch seinen *Spannungs-* und *Stromverlauf* charakterisiert werden. Die beiden Merkmale lassen sich ableiten zu der *Signalform* und der jeweiligen *Codierung* der Daten. Als Ausgangsdaten werden ausschließlich binäre Signale betrachtet. In den hier betrachteten Systemen wird keine Modulierung der Daten vorgenommen, wie z. B. durch ein Modem (Modulator / Demodulator), so dass sie auch als binäre Signale übertragen und empfangen werden können. Die Codierung der Daten steht oft in unmittelbarem Zusammenhang mit den physikalischen Eigenschaften des Mediums. Eine große Auswahl verschiedener Codierungsarten ist in [Tan96] vorgestellt. Ein Effekt, der durch Codierung erreicht werden kann, ist die implizite Übertragung des Sendertakts in den Daten selbst. Das erspart eine separate Takt-Leitung.

Spannungspegel und Signalleistung

Der *Spannungspegel* ist eine wichtige Eigenschaft der Signalform und wird deshalb getrennt aufgeführt. Wenn verschiedene Technologien aufeinander treffen, ist der Spannungspegel häufig ein Aspekt, der bei der Adaption berücksichtigt werden muss. Ein Beispiel für eine solche Transformation der Spannungspegel ist in Abbildung 2.6 illustriert. Der Fall tritt auf, wenn ein digitales System, das logisch Null mit 0V und logisch Eins mit 3,3V codiert, auf ein anderes System abgebildet werden muss, das Null mit +12V und logisch Eins mit -12V beschreibt.

Einordnung in das Schichtenmodell

Um den Aufbau komplexer Übertragungssysteme zu beschreiben, werden *Schichtenmodelle* angegeben und erläutert. Dadurch entsteht ein besserer Gesamteindruck und die folgenden SW-Parameter werden leichter verständlich gemacht.

Aufbau der Datenpakete

Eine charakteristische SW-Eigenschaft des Protokolls ist der *Aufbau der Datenpakete*. Dadurch lässt sich festlegen, in welcher Form die Daten verschickt werden. Dazu werden sowohl Nutz- als auch Steuerdaten gezählt. Auf der obersten Schicht erzeugt man die Nutzdaten, die zur Zielinstanz transportiert werden müssen. Auf dem Weg dahin fügen einzelne Dienste Steuerinformationen am Anfang (engl.: Header) und / oder am Ende (engl.: Trailer) an. Gelegentlich werden auch Zusatzinformationen zur Fehlerabsicherung

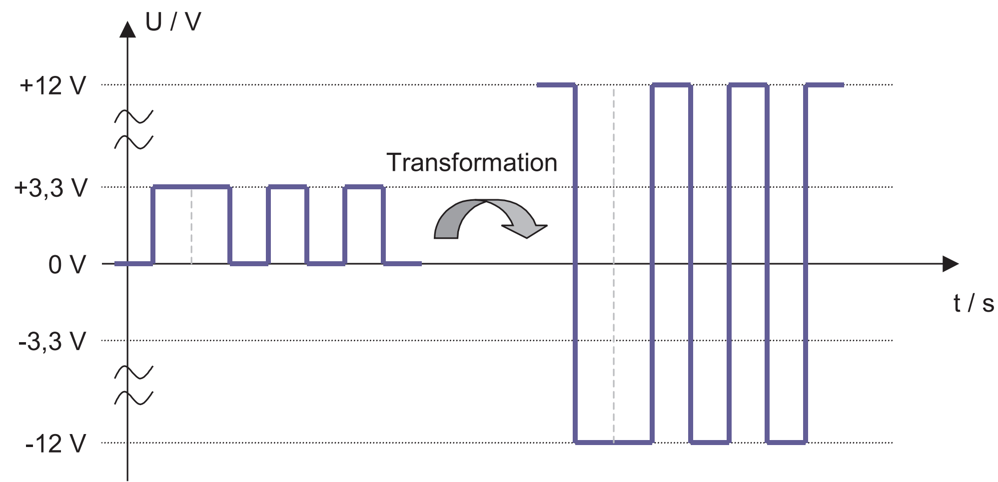


Abbildung 2.6: Spannungstransformation von (0V, +3.3V) auf (+12V, -12V)

in der Mitte eingefügt. Das Verhältnis zwischen Nutz- und Steuerdaten beeinflusst die Datenrate (s. u.). Der Aufbau solcher Datenpakete kann abstrakt gesehen durch einen Protokollgenerator geschehen. Ein solcher Protokollgenerator ist Teil der automatischen Generierung und wird in Kapitel 4 beschrieben. Den Aufbau eines solchen Datenpakets veranschaulicht Abbildung 2.7. Das Beispiel stammt aus dem Bereich der Netzwerkprotokolle, die sehr komplexe Datenpakete behandeln.

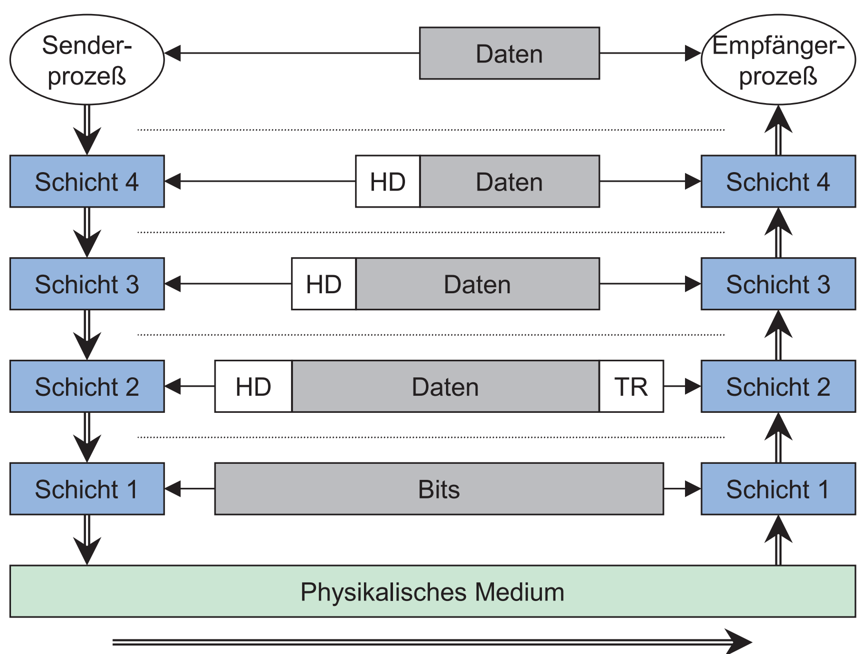


Abbildung 2.7: Aufbau von Datenpaketen

Übertragungsrate, Bandbreite

Mit *Übertragungsrate* ist die Anzahl der übertragenen Bits pro Sekunde gemeint. Es gibt auch eine Nutzdatenrate, die bezieht sich dann auf die Übertragung von Nutzdaten pro Sekunde. Nutzdaten sind der Anteil einer Nachricht, der die eigentliche Information trägt. Den Rest bezeichnet man als Redundanz. Das Verhältnis von Nutzdaten zu den Gesamtdaten gibt die *Effektivität* der Übertragung an. Die *Baudrate*, in der Literatur auch als auch als Schrittgeschwindigkeit bekannt, ist ein Maß für die übertragenen Symbole pro Sekunde. Im Fall von binären Daten ist die Baudrate identisch zur Übertragungsrate, was hier der Fall ist. Es gilt: $f_{gr} = \frac{1}{T}$, wobei f_{gr} die maximale Grenzfrequenz ist und T die Übertragungszeit. Wie man aus der Gleichung erkennen kann, ist die Frequenz antiproportional zur Übertragungszeit. Das bedeutet, falls man ein kleine Übertragungszeit erreichen möchte, muss man hohe Frequenzen bewältigen können.

Die *Bandbreite* eines Systems, im nachrichtentechnischen Sinn gesehen, legt die Obergrenze der maximal zu übertragenden Frequenz f_{gr} fest. Eine nähere Erläuterung der Bandbreite ist vielmehr Inhalt der Nachrichtentechnik [Kam96, Lük99, Pro95]. Es sei aber noch angemerkt, dass gilt: Das Frequenz-Bandbreite-Produkt ist immer konstant. Ein Nachteil, der aus hohen Übertragungsraten resultiert ist, dass hohe Bandbreiten auch entsprechend teure Medien erfordern (z. B. Glasfaser).

Die *Effektivität* eines Protokolls spiegelt sich im Verhältnis der Nutzdatenleitungen zu denen der Steuer- und Signalleitungen wieder. Parallele Schnittstellen ermöglichen dem Protokoll mehrere Datenbits gleichzeitig zu versenden, dafür handelt man sich aber zusätzliche Leitungen ein, die wiederum Kosten verursachen.

Zeitverhalten

Das *Zeitverhalten* beschreibt die zeitlichen Abläufe eine Protokolls. Auf oberster Ebene gibt es zwei Typen von Verhalten. Man unterscheidet *synchrone* und *asynchrone* Protokolle. Ein synchrones Protokoll basiert auf Zeitscheiben (engl.: Time Slot), die häufig durch einen Takt dargestellt werden. Alle Ereignisse (engl.: Event), die behandelt werden, sind mit diesem Takt synchronisiert, d. h. zeitlich davon abhängig. Asynchrone Protokolle sind ereignisgesteuert (engl.: Event-Triggered). Darunter versteht man, dass bestimmte Schlüsselereignisse, unabhängig von jeder Zeitbasis, spezielle Vorgänge auslösen.

Zu jeder Regel gibt es Ausnahmen: Es können auch Mischformen existieren. Beispielsweise kann der Start einer Übertragung durchaus ereignisgesteuert sein, die eigentliche Übertragung erfolgt dann aber nach einem festen Zeitschema. Häufig sind digitale Systeme nicht in der Lage, asynchrone Vorgänge zu verarbeiten. Falls der Takt eines Systems nur hoch genug ist, kann man nach außen hin ein scheinbar asynchrones Verhalten vorgeben, indem eine Reaktion auf ein Ereignis erst mit der darauf folgenden Taktflanke ausgewertet wird. Die zeitliche Differenz zum asynchronen Ereignis beträgt dann maximal einen Takt.

Im weiteren soll unter dem Begriff *Zeitverhalten* auch der entsprechende zeitliche Verlauf eines Protokolls verstanden werden. Das ist notwendig für Echtzeitprotokolle, die Bedingung für eine Echtzeitschnittstellen sind. Ein Protokoll schickt z. B. zuerst ein Startbit, dann Datenbits, und am Ende ein Stopbit. Jedes Bit muss zu seinem vordefinierten Zeitpunkt übertragen werden. Dann ist das Protokoll deterministisch.

Handshake-Verfahren

Um korrekt miteinander kommunizieren zu können, benötigen zwei Partner ein *Handshake*, zu Deutsch: ein „Händereichen“. Das Handshake schreibt dem Protokoll genau vor, welche Aktionen zwischen zwei Instanzen einer Schicht aufeinander folgen. Dabei ist zu unterscheiden, ob eine Paket-Kommunikation erfolgt oder Steuerleitungen und Datenleitungen parallel vorliegen. Der erste Fall ist bei Netzwerkkommunikation üblich. Die Kommandosequenzen und Daten werden in gleichen Paketen sequentiell verschickt. Der zweite Fall liegt bei einer festen Verdrahtung der Komponenten vor, wie es bei einem Nullmodemkabel oder einem Druckeranschluss über den Parallel-Port der Fall ist.

Es gibt viele Beispiele für beide Verfahren. Eine häufig zitierte Anwendung für den ersten Fall ist der Verbindungsaufbau zwischen Sender und Empfänger in Netzwerken. Weitere Informationen zu dem Thema findet man in [Tan96]. Um an Geschwindigkeit zu gewinnen und die Kommunikation zu vereinfachen, verfügen bestimmte eingebettete Systeme über *Steuer*- und *Datenleitungen*. So können beide Datenarten parallel arbeiten. Das kann man anhand einer einfachen Datenkommunikation verdeutlichen. Der Sender gibt das Signal „bereit zu senden“, welches der Empfänger mit „fertig zum Empfang“ quittiert. Darauf legt der Sender die Daten auf die Datenleitungen und gibt das Signal „Daten zum Lesen gültig“, wenn sich die Werte der Datenleitungen stabilisiert haben. Im Anschluss daran antwortet der Empfänger mit „Lesen abgeschlossen“, wenn er die Werte aufgenommen hat. Dann kann das Protokoll von vorne beginnen. Eine Veränderung durch Vertauschen der Abfolge im Handshake würde nicht zum gewünschten Ziel führen.

Prioritätsvergabe, Zugriff auf das Medium

Um überhaupt auf ein Medium, das über mehr als einen Sender pro Kanal verfügt, zugreifen zu können, muss ein *Zugriffsverfahren (Arbitrierung)* eingesetzt werden. Es gibt verschiedenen Methoden, die sich in die unterschiedlichen Kategorien der Echtzeit einordnen lassen. In Abschnitt 2.4 wird ausführlich auf die Bedeutung des Begriffs Echtzeit eingegangen. Diese Arbeit beschränkt sich auf Probleme der „harten“ Echtzeit. Daraus folgt, dass nur Zugriffsverfahren eingesetzt werden können, die diesen Anforderungen genügen oder darauf angepasst werden können. Die Bedingungen für harte Echtzeit erreicht jedenfalls durch *statische Kanalzuordnung*. Dabei wird die Zeitbasis in Zeitscheiben (engl.: Time Slots) eingeteilt, die je nach Bedarf bei der Berechnung der Ablaufplanung den Sendern zugewiesen werden. Die Zeitbasis wird dann Slot für Slot zyklisch abgearbeitet. Selbst wenn eine Sendereinheit zu dieser Zeit keine Nachricht zu versenden hat, steht ihr die Zeit zur Verfügung. Gegebenenfalls wird dann einfach ein

Leerrahmen verschickt.

Alle auf Prioritätsbasis basierenden Systeme können höchstens in „weichen“ Echtzeitsystemen zum Einsatz kommen. Es kann dort nicht gewährleistet werden, dass eine Instanz, die nicht die höchste Priorität hat, in einer vordefinierten Zeit das Senderecht erhält. Für harte Echtzeitsysteme ist das nicht tragbar. Einige Verfahren, die harte Echtzeitbedingungen erfüllen, werden nun vorgestellt.

2.2.2 Die Steckverbindung – eine HW-Schnittstelle

Die in dieser Arbeit verwendeten *Steckverbindungen* werden hier vorgestellt. Sie sind jeweils mit den genormten Bezeichnungen der zugehörigen Protokolle beschriftet [Ber00]. Eine Steckverbindung besteht aus einem *Stecker* und der dazu gehörigen *Buchse*. Die Stecker, auch als männlich bezeichnete Steckverbindung, haben Pins, wohingegen die Buchsen, die auch als weiblich bezeichnet werden, Federn haben. Einige Ausgänge am PC sind bei gleicher Leitungsanzahl männlich, andere weiblich. Durch diesen mechanischen Unterschied ist es möglich zwei verschiedene Schnittstellentypen voneinander zu trennen, die die gleiche Steckverbindung verwenden. Ein Beispiel dafür ist die parallele (25 SUBD, weiblich; vgl. Abbildung 2.9) und die früher benutzte serielle Schnittstelle (25 SUBD, männlich; vgl. Abbildung 2.11) des PCs. Dadurch konnte ein man ein fehlerhaftes Anschießen eines parallelen Geräts an die serielle Schnittstelle ausschließen. Deshalb erfolgt auch die Definition eines Anschlusses an Steckern oder Buchsen.

SUBD, 9 oder 25 polig, männlich

Abbildung 2.8: SUBD, 9 polig, männlich

Auf dem Stecker SUBD mit 9 Polen (vgl. Abbildung 2.8) wird üblicherweise ein serielles Protokoll definiert. Hier handelt es sich um das V24 oder auch als RS232 bezeichnete serielle Protokoll. Für diese Übertragungsart wurde früher ebenfalls der Stecker SUBD mit 25 Polen benutzt (vgl. Abbildung 2.9). Auf den Abbildungen sind Daten- und Steuerleitungen beschriftet.Als Medium werden an SUBD-Stecker i. A. ungeschirmte Kupferkabel angeschlossen.

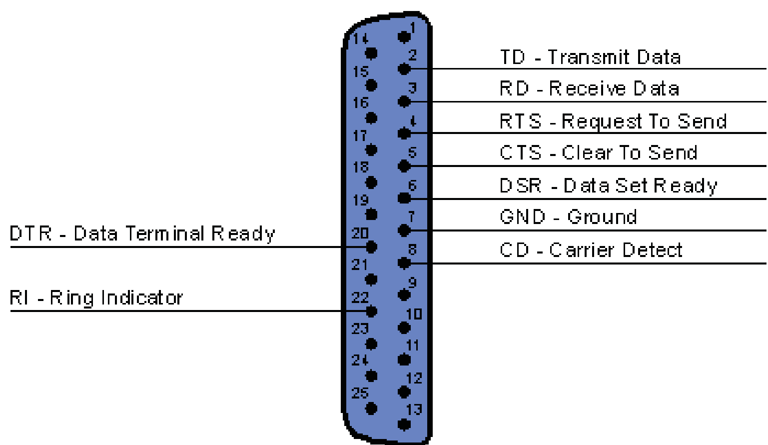


Abbildung 2.9: SUBD, 25 polig, männlich

SUBD, 15 polig, männlich

Dieser Stecker (vgl. Abbildung 2.10) ist in der PC-Welt für den Gameport reserviert. Mit dessen Hilfe wird z. B. ein analoger Joystick angeschlossen. Auch hier sind die Leitungen „standardisiert benannt“.

+ 5 Volt
Button 7
Position 3
GND - Ground
Position 2
Button 5
+ 5 Volt
+ 5 Volt
Button 3
Position 1
GND - Ground
GND - Ground
Position 0
Button 1
+ 5 Volt

Abbildung 2.10: SUBD, 15 polig, männlich

SUBD, 25 polig, weiblich

Der Stecker SUBD mit 25 Polen (vgl. Abbildung 2.11) dient auch als parallele Schnittstelle. Der Ausgang am PC ist im Gegensatz zum seriellen Port weiblich, d. h. mit Buchse versehen. In Abbildung 2.11 ist die Standardbelegung der parallelen Schnittstelle aufgeführt.

Abbildung 2.11: SUBD, 25 polig, weiblich

USB A, B

Es gibt zwei Steckertypen für den USB (Universal Serial Bus). Der sogenannte Typ „A“ (vgl. Abbildung 2.12) wird am *Host* angeschlossen, Typ „B“ (vgl. Abbildung 2.13) dagegen wird am *Client* eingesteckt. Dies gewährleistet den korrekten Anschluss und die Einhaltung der Datenflussrichtung. Als Übertragungsmedium wird ein Twisted-Pair Kabel benutzt.

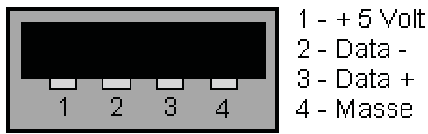


Abbildung 2.12: USB-Buchse Host

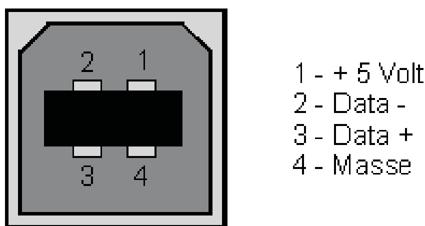


Abbildung 2.13: USB-Buchse Client

Rj45

Der Rj45 Stecker wird heutzutage in vielen Bereichen genutzt, z. B. für LANs (Local Area Networks), Modems, ISDN und Analog-Telefon Anwendungen. Entweder schließt man einfaches Kupferkabel oder Twisted-Pair Kabel (LAN und Netzwerk) an.

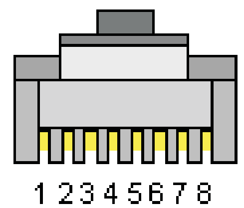


Abbildung 2.14: Rj45 Stecker

2.2.3 Das serielle Protokoll

Ein serielles Protokoll versendet die Daten sequentiell, d. h. Bit für Bit nacheinander. Es gibt mehrere serielle Protokolle und somit Schnittstellen, die sich in ihren Parametern unterscheiden. Eine Implementierung des seriellen Protokolls ist das V24 (auch RS232) -Protokoll [Dem97].

Übertragungsmedium

Als Medium wird meistens einfaches Kupferkabel verwendet. Das ist sehr günstig und lässt sich leicht verlegen. Wenn eine bessere Abschirmung erforderlich ist, kann auch verdrilltes Kupferkabel eingesetzt werden. V24-Verkehr ist aber auch über andere Medien, wie IR (Infrarot) oder Ultraschall, abwickelbar.

Steckverbindungen

Wie schon am Stecker (vgl. Abbildung 2.8 oder 2.9) ersichtlich ist, besteht die V24-Schnittstelle aus zwei Daten-, einer Masse- und ggf. mehreren Steuerleitungen. Die Datenleitungen werden mit RD (bzw. RxD) und TD (bzw. TxD) bezeichnet. Darauf werden die eigentlichen Nutzdaten gesendet. Die Spannungspegel der RxD- und TxD-Leitung werden auf GND, die Signalmasse, bezogen. Die Steuerleitungen CD, DTR, DSR, RTS, CTS und RI dienen der Implementierung eines Handshakes. Eine genaue Beschreibung der Steuersignale wird in Tabelle 2.2 gegeben. DÜE bezeichnet eine Datenübertragungseinrichtung, DEE steht für Datenendeinrichtung. Als Abkürzung für Sender und Empfänger wird i. A. S (Sender) und E (Empfänger) benutzt.

Bedeutung der Steuerleitungen: CD wird aktiv, wenn z. B. ein angeschlossenes Modem mit einem anderen Modem eine Verbindung aufgenommen hat. Somit erkennt der PC,

Tabelle 2.2: V24-Steckerbelegung [Har01]

Pin	Name	Bezeichnung	Funktion	Richtung
1	CD	carrier detect	Träger erkannt	E
2	RxD	read data	Empfangsdaten	E
3	TxD	transmit data	Sendedaten	S
4	DTR	data terminal ready	DEE empfangsbereit	E
5	GND	ground	Signalmasse	-
6	DSR	data set ready	DÜE betriebsbereit	S
7	RTS	request to send	Sendeanforderung	S
8	CTS	clear to send	Sendebereitschaft	E
9	RI	ring indicator	Ankommender Ruf	E

dass eine Verbindung besteht und Daten gesendet werden können. Mit DTR signalisiert ein Rechner, z.B. bei einer Direktverbindung, seine Betriebsbereitschaft. DSR ist die Antwort auf DTR (bei gekreuzten Leitungen). RTS wird aktiv, wenn ein Endgerät bereit ist, Daten zu senden. CTS dagegen wird aktiv, wenn ein Endgerät bereit ist, Daten zu empfangen. Das Signal RI wird von einem angeschlossenen Modem bei einem eingehenden Ruf aktiviert.

Signalform und Codierung

Die Daten werden als digital Signale übertragen. Als Codierung verwendet man ein NRZ (Non Return to Zero) -Verfahren. D. h. sowohl der logischen Eins als auch der logischen Null wird je ein Wert zugewiesen, der, solange dieser logische Wert anliegt, auch gehalten wird. Eine Folge gleicher Eingangswerte erzeugt auch eine Folge gleicher Ausgangswerte, die auf das jeweilige Spannungslevel angepasst werden. Die Separation von gleichen, aufeinander folgenden Zeichen ist nur mit Hilfe eines Taktsignals möglich.

Spannungspegel und Signalleistung

Das V24-Protokoll definiert logisch Null (Low-Pegel) mit +12V und logisch Eins (High-Pegel) mit -12V Gleichspannung. Die Werte werden bis $\leq +1$V als High-Pegel, darüber als Low-Pegel, erkannt. Es fließt ein Ausgangsstrom bis zu 10 mA bei einem Eingangswiderstand von 10 kΩ.

Einordnung in das Schichtenmodell

In dieser Arbeit wird nur die physikalische Ebene des V24-Verkehrs betrachtet. Das entspricht der *Schicht 1* in Abbildung 2.3. Eine weiter oben angeordnete Schicht könnte eine Fehlererkennung mit Hilfe des Paritätsbits durchführen oder für die Aufteilung des Datenstroms in Paketgröße sorgen.

Aufbau der Datenpakete

Die Datenübertragung erfolgt als Paket, das jeweils bitweise versendet wird. Die Nutzdaten werden mit einem Header und einem Trailer versehen. Die Paketgröße ist in Grenzen variierbar: Es gibt ein Startbit (logisch 0), 7 oder 8 Datenbits, 0 oder 1 Paritätsbit und abschließend 1 bis 2 Stopbits (logisch 1). Abbildung 2.15 zeigt die im Demonstrator verwendete Variante mit 1 Startbit, 8 Datenbits, 0 Paritätsbits und 2 Stopbits.

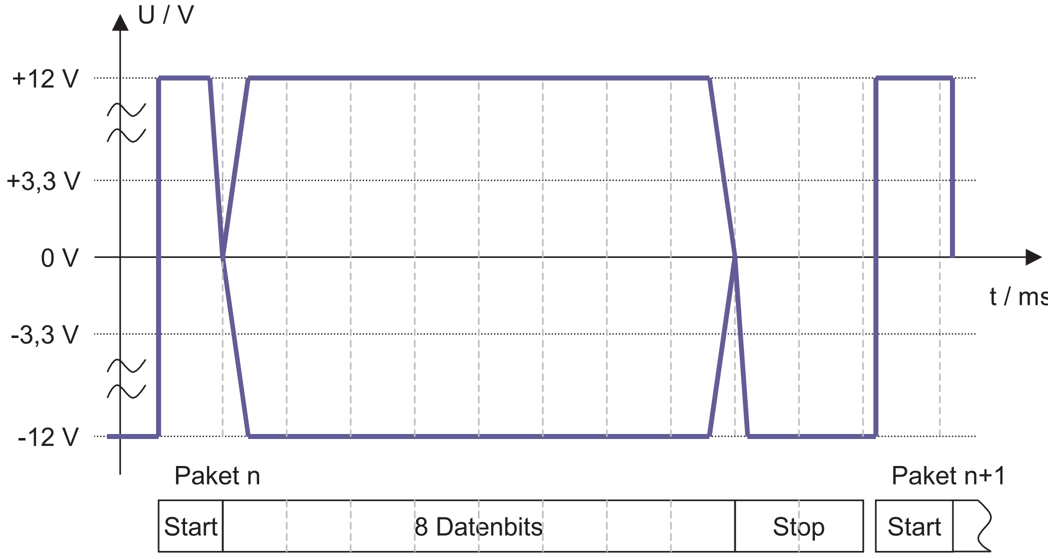


Abbildung 2.15: Aufbau des V24-Protokolls

Übertragungsrate, Bandbreite

Es sind mehrere Datenübertragungsraten für V24-Verkehr genormt. Übliche Datenraten, der V24-Schnittstelle sind:

- 4800 Baud
- 9600 Baud (Steuereinheit vom Roboter des Demonstrators)
- 38K, 56K Baud (serielles Modem)

Die anfänglichen Werte waren kleiner und stammen aus der Entwicklungszeit der Datenübertragung. Heutzutage gibt es aber auch noch größere als die hier aufgeführten Baudraten. Im Demonstrator, der in dieser Arbeit evaluiert wird, erfolgt die Kommunikation über eine synthetisierte V24-Schnittstelle mit einer Datenrate von 2 MBaud. Aus der Datenrate lassen sich die Übertragungszeiten, daraus die Frequenzen und somit auch die benötigten Bandbreitenanforderungen an das Medium berechnen.

Zeitverhalten

Das V24-Protokoll ist eigentlich eine der oben beschriebenen Mischformen aus synchronem und asynchronem Verhalten. Der Anfangszeitpunkt der Sendung ist nicht deterministisch, die eigentliche Übertragung erfolgt dann aber streng synchron. Die Anfangsphase des Sendens bezeichnet man als Synchronisationsphase. Um das Problem des asynchronen Sendestarts zu lösen, wird die Möglichkeit des Sendens nur noch zu vorgegebenen Zeitpunkten erlaubt. Das Eintreffen des ersten Signals startet im Empfänger den synchronen Empfang. Jetzt ist die Datenübertragung deterministisch. Die möglichen Sendzeitpunkte werden bestimmt durch das Zugriffsverfahren des Mediums. Die Signallaufzeit des Mediums, die auch als Latenzzeit bezeichnet wird, ist für alle zu übertragenden Signale konstant.

Handshake-Verfahren

Für das V24-Protokoll existieren eine Reihe von Handshake-Verfahren. Diese dienen der eindeutigen Verständigung zwischen Sender und Empfänger. Abbildung 2.16 zeigt am Beispiel das Hardware Handshake der V24-Schnittstelle, wenn als Datenendeinrichtung (DEE) ein Modem angeschlossen ist.

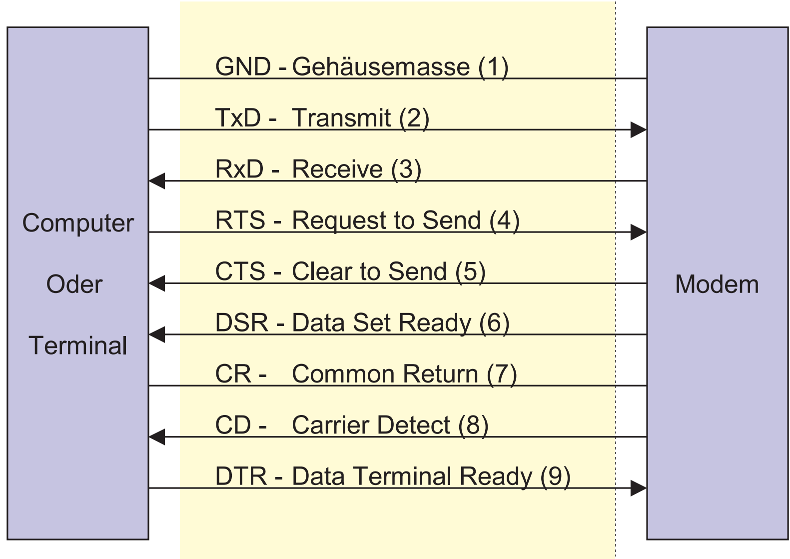


Abbildung 2.16: V24-Hardware-Handshake

Wollen zwei Computer direkt miteinander kommunizieren, wird das Signal CD nicht benötigt. Um den Rechnern beim Auftreten einer Übertragungsanfrage vorzutäuschen, dass ein Träger vorhanden ist, wird die CD- mit der DTR-Leitung kurzgeschlossen. Soll nun eine Verbindung aufgebaut werden, so signalisiert der Sender durch ein aktives DTR-Signal die Betriebsbereitschaft. Die Aktivierung dieses Signals bedeutet, dass dieser Rechner mit einem anderen in Kontakt treten möchte. Da die Verbindungsleitungen

des DTR- und des DSR-Signals überkreuzt sind, erkennt die Gegenstelle dieses Signal als DSR. Ist auch der empfangende Rechner bereit für die Datenübertragung, so quittiert er dieses mit einem aktiven DTR Signal, welches beim ersten Rechner wiederum als DSR empfangen wird. Durch Deaktivierung des DTR Signals im Sender wird die Kommunikation dann beendet.

In allen V24-Duplex-Protokollen sind immer die zwei Datenleitungen (RxD, TxD) beteiligt. Das impliziert auch die Signalmasse-Leitung als drittes Kabel. Je nach Komplexität der DEE werden unterschiedlich viele Steuerleitungen in das Handshake einbezogen. Im Demonstrator sind lediglich RxD und TxD verwendet. Die Funktion der anderen Leitungen wird einfach vorausgesetzt, so z. B. die Bereitschaft des DEE.

Prioritätsvergabe, Zugriff auf das Medium

Das Zeitverhalten des seriellen Protokolls ist deterministisch, wie sich aus der synchronen Datenübertragung und den festen Latenzzeiten ableiten lässt(s. o.). Das Zugriffsrecht auf den Kanal ist statisch festgelegt. Jeder Sender hat seinen Kanal, was bedeutet, dass keine Kollision vorkommen kann. Trotzdem muss eine feste Zeiteinteilung in Sende-Zeitschlitze (engl.: Time Slots) erfolgen, um feststellen zu können, ob der Sender noch „online“ ist. Determinismus und statische Arbitrierung machen die V24-Schnittstelle zu einer echtzeitfähigen Schnittstelle.

2.2.4 Das parallele Protokoll

Im Gegensatz zu seriellen Protokollen werden bei parallelen Protokollen die Daten parallel übertragen. Das bietet den Vorteil einer schnelleren Übertragung. Die parallele-, oder auch Centronics-Schnittstelle, wurde vor vielen Jahren vom Druckerhersteller Centronics eingeführt und von anderen Herstellern übernommen. Normalerweise ist diese Schnittstelle für den Anschluss eines Druckers vorgesehen und wird auch in den meisten Fällen dafür verwendet. Es arbeiten aber auch I / O-Geräte, wie z. B. externe Brenner, Scanner oder ZIP-Laufwerke, über diese Schnittstelle. Ein anderes großes Anwendungsfeld aus früherer Zeit sind Programme der höheren Preiskategorien, zu denen die Hersteller einen sogenannten Dongel als Hardware-Kopierschutz benutzen. Dies ist ein Modul, das einfach zwischen die Schnittstelle und den Drucker gesteckt wird. Dieses Modul wird für jedes Programm individuell programmiert, so dass es für jede Seriennummer einen entsprechenden Dongel gibt.

Übertragungsmedium

Ebenso wie beim V24-Protokoll dient als Übertragungsmedium einfaches Kupferkabel. Allerdings werden 25 statt nur 9 Leitungen benötigt, da eine größere Anzahl von Signalen vorhanden ist. Dies erfordert eine aufwendigere Steckverbindung, die über 8 Datenleitungen verfügt. Mit den restlichen 17 Pins kann ein erweitertes Handshake-Verfahren implementiert werden.

Steckverbindungen

Die Buchse der parallelen Schnittstelle ist in Abbildung 2.11 dargestellt. In der Tabelle 2.3 werden die einzelnen Leitungen noch einmal aufgeführt und deren Funktion näher erläutert.

Tabelle 2.3: Steckerbelegung der parallelen Schnittstelle [Har01]

Pin	Name	Sender	Funktion
1	STROBE	CPU	Übernahmeimpuls für die Daten (fallende Flanke)
2	Daten 0	CPU	Datenleitung (niederwertigstes Bit)
3	Daten 1	CPU	
4	Daten 2	CPU	.
5	Daten 3	CPU	.
6	Daten 4	CPU	.
7	Daten 5	CPU	.
8	Daten 6	CPU	
9	Daten 7	CPU	Datenleitung (höchstwertigstes Bit)
10	ACK	Drucker	Übernahmebestätigung für Daten (low-Pegel)
11	BUSY	Drucker	Empfangsbereitschaft, wird durch low-Pegel signalisiert
12	PE	Drucker	Papiermangelanzeige
13	SLCT	Drucker	Zeigt Online-Status des Druckers an
14	ALF	CPU	Automatisches Einfügen eines Line-Feeds bei low-Pegel
15	ERROR	Drucker	Fehleranzeige
16	INIT	CPU	Initialisierung des Druckers und leeren des Puffers durch low-Pegel
17	SHIELD	CPU	Aktivierung des Druckers
18-25	GND	-	Signalmasse

Die Leitungen werden in Datenleitungen (Pin 2 - 9), Steuerleitungen (Pin 1, 14, 16, 17) und Meldeleitungen (Pin 10 - 13, 15) eingeteilt. Das Protokoll ist für unidirektionalen Verkehr ausgelegt. Falls ein Scanner oder externes Laufwerk angeschlossen wird, werden die Leitungen entsprechend anders interpretiert, um bidirektionalen Verkehr zu ermöglichen.

Signalform und Codierung

Ebenso wie beim V24-Protokoll werden digitale Signale übertragen, und als Codierung verwendet man ein NRZ-Verfahren. Eine Folge gleicher Eingangswerte erzeugt auch eine Folge gleicher Ausgangswerte, die auf das jeweilige Spannungslevel transformiert werden.

Spannungspegel und Signalleistung

Die Parallele Schnittstelle definiert logisch Null (Low-Pegel) mit 0V und logisch Eins (High-Pegel) mit +5V Gleichspannung.

Einordnung in das Schichtenmodell

Auf der physikalischen Schicht des Parallelen Protokolls kann eine Applikation aufsetzen oder eine weitere Schicht folgen, in der eine Datenvorverarbeitung stattfindet. Die Angaben hier beziehen sich auf das Protokoll der physikalischen Schicht.

Aufbau der Datenpakete

Da die Daten parallel zu den Signal- und Steuerleitungen übertragen werden, kann man das Nutzdaten-Byte als Datenpaket ansehen. Das ermöglicht einen erhöhten Datendurchsatz. Allerdings ist zu beachten, dass nur Pakete fester Größe parallel übertragen werden können. Das ist problematisch, da die Anzahl der Datenleitungen proportional zur Paketgröße anwächst. Zum Vergleich sei hier nur das serielle Ethernetprotokoll benannt, dass eine Paketgrößen im kByte-Bereich verwendet. Die Umsetzung in ein paralleles Protokoll würde trotz gesteigerter Bandbreite am Kosten-Nutzen-Effekt scheitern.

Übertragungsrate, Bandbreite

Die Übertragungsrate errechnet sich aus den transportierten Nutzdaten pro Sekunde. Es sind alle acht Datenleitungen einzubeziehen. Das Versenden der Daten ist abhängig von den Steuer und Signalleitungen (siehe Handshake-Verfahren).

Zeitverhalten

Die parallele Schnittstelle ist im Druckerbetrieb asynchron und nicht echtzeitfähig. Will man Datenkommunikation auf Echtzeitbasis betreiben muss man alle Signal- und Steuerleitungen zyklisch in festgelegten Zeitscheiben abfragen. Nur so kann man deterministische Voraussagen über die zeitlichen Abläufe des Protokolls geben. Allgemein gilt: Jedes Signal, das erfasst werden soll, muss zyklisch abgefragt werden. Die Periodendauer wird dabei nach oben durch die maximal zu tolerierende Zeitspanne, in der dieses Signal eine Wirkung hervorrufen soll, beschränkt.

Handshake-Verfahren

Die Funktionsweise des parallelen Handshakes wird wieder am Beispiel des Druckers erklärt. Zuerst wird vom Prozessor ein Byte auf die Datenleitungen geschrieben. Mit einem Low-Pegel auf STROBE wird dem Endgerät mitgeteilt, das die Daten gültig sind. Nach dem Empfang der Daten signalisiert das Endgerät dem PC durch Aktivierung des BUSY Signals, dass es noch mit der Abarbeitung beschäftigt ist. Bei einem Drucker wird

dieses Signal z.B. beim Papiervorschub, bei einem Fehler, bei der Initialisierung und während der Dateneingabe aktiviert. Ist das Endgerät mit der Abarbeitung fertig, so wird dies dem Computer durch Aktivierung des ACK (Acknowledge) angezeigt. Danach können sofort weitere Daten gesendet werden. Diese Prozedur wird 3-Draht-Handshake der Centronics Schnittstelle genannt. Die Signale der Centronics Schnittstelle haben alle TTL-Pegel und sind in positiver Logik ausgeführt. Als nächstes folgt eine Betrachtung der Meldeleitungen: Wenn einem Drucker das Papier ausgeht, so meldet er dies über Paper Empty (PE). Ist also der Drucker am Blattende, so wird der Druckvorgang unterbrochen und der logische Pegel auf PE wechselt von Low nach High. Wenn nun ein neues Blatt eingelegt wird so wird diese wieder deaktiviert. Durch das SELECT Signal wird dem Computer mitgeteilt, ob sich der Drucker im On- oder Offlinemodus befindet (High = online, Low = offline). Ein Low-Pegel auf der ALF-Leitung führt zu einem automatischen Zeilenvorschub des Druckers. Durch das Signal INIT hat der Computer die Möglichkeit, ein angeschlossenes Gerät in seinen Anfangszustand zurückzusetzen. Bei einem Drucker bedeutet dies das Leeren des Druckerpuffers und eine Neupositionierung des Druckkopfes. Sind mehrere Drucker angeschlossen, so können die einzelnen über SELECT in angesprochen werden. Stellt der Drucker einen internen Fehler fest, so kann er dies dem PC mit dem ERROR Signal mitteilen. [Har01]

Prioritätsvergabe, Zugriff auf das Medium

Wenn mit dem parallelen Protokoll Daten deterministisch übertragen werden sollen, kann das Protokoll echtzeitfähig aufgebaut werden. Die Übertragung ist unidirektional, somit hat der Sender keine Probleme mit dem Zugriff auf das Medium. Das Zeitverhalten ist, abgeleitet aus der vordefinierten Datenrate und festen Latenzzeiten, deterministisch. Das macht die Parallele Schnittstelle zu einer echtzeitfähigen Schnittstelle, wenn eine statische Arbitrierung eingehalten wird.

2.2.5 Universal Serial Bus (USB)

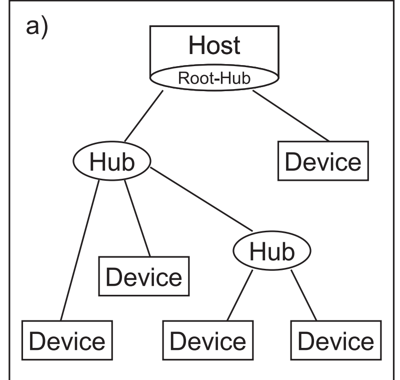
Der USB (Universal Serial Bus) wurde entwickelt, um eine Vereinheitlichung der unterschiedlichen I/O Schnittstellen zu fördern, die Anschlussmöglichkeiten zu erweitern und zu vereinfachen. Im Jahr 2000 wurde die Version USB 2.0 verabschiedet. USB 2.0 bietet zu den zwei alten eine neue High-Speed Geschwindigkeitsklasse an, um den gestiegenen Bandbreitenanforderungen Rechnung zu tragen. Jede Klasse hat ihr eigenes Protokoll. Die Informationen zu USB sind aus folgenden Quellen zusammengetragen worden: [Kel99] [For00] [Sch97] [Fis01]. In dieser Arbeit wird nur die High-Speed Klasse betrachtet.

Übertragungsmedium

Das bei USB verwendete Übertragungsmedium ist das Twisted-Pair Kabel. Es ermöglicht eine einfache Verlegung, die auch von Laien ausgeführt werden kann. Ein Ziel von USB

war es, Peripheriegeräte von PCs im laufenden Betrieb durch „Hot Plug-and-Play“ in Betrieb zu nehmen, d. h. Geräte im laufenden Betrieb anschließen und entfernen zu können. Dazu ist eine simple Verkabelung die Voraussetzung. Das Kabel besteht aus vier Adern: Masse, Power, D- und D+. Die maximale Länge für Full- / Highspeed-Kabel beträgt 5 Meter.

Die Topologie des USB ist als gemischte Stern- und Strang-Bus-Technologie entworfen, auch „Tiered-Star“ genannt (vgl. Abbildung 2.17). Es können maximal 127 Geräte angeschlossen werden. Darin sind *Hubs* und alle Arten von USB-Geräten, genannt *Functions*, enthalten.



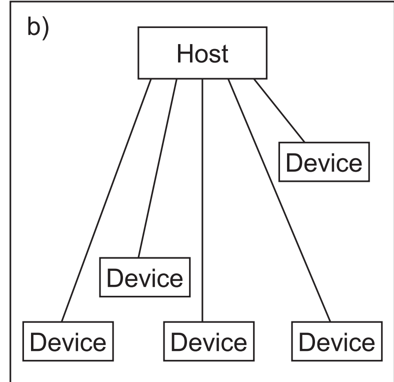


Abbildung 2.17: a) Physikalische Topologie von USB, b) Logische Struktur)

Steckverbindungen

Es gibt zwei Steckertypen für den USB. Der sogenannte Typ „A“ (vgl. Abbildung 2.12) wird am *Host*, Typ „B“ (vgl. Abbildung 2.13) dagegen am *Client* eingesteckt. Der Verkehr vom Host zum Client bezeichnet man als Upstream, die Gegenrichtung als Downstream. Ein Kabel hat immer ein Stecker vom Typ A und einen vom Typ B. Das verhindert gleichzeitig die Bildung von ungültigen Ringstrukturen.

Signalform und Codierung

Die zwei Datenleitungen dienen der differentiellen Übertragung des Signals, was die Störanfälligkeit herabsetzt. Zur Darstellung der Daten auf dem physikalischen Medium wird eine NRZI (Non Return to Zero Invert) Codierung (vgl. Abbildung 2.18) eingesetzt. Das ist ein Verfahren, bei dem eine Null einen Pegelwechsel auslöst, eine Eins aber nicht. Falls 6 Einsen in Folge auftreten wird durch *Bitstuffing* automatisch eine Null eingefügt, die auf dem Empfängerseite wieder entfernt wird. Aus NRZI kann man den Sendertakt wiederherstellen und hat so eine zusätzliche Taktleitung gespart. Das setzt eine synchrone Übertragung voraus und ist eine wichtige Eigenschaft des Zeitverhaltens.

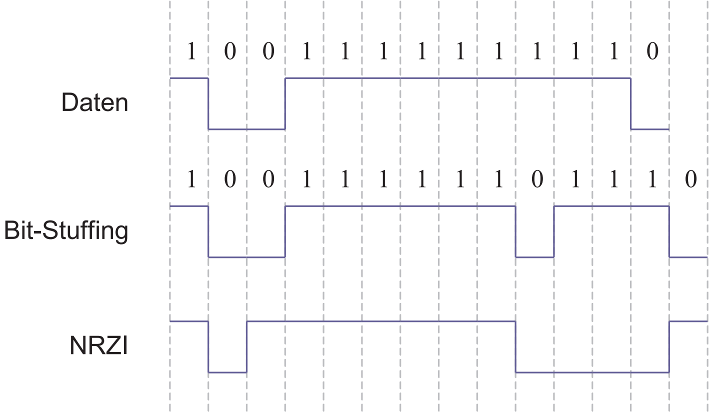


Abbildung 2.18: Signalcodierung von USB

Spannungspegel und Signalleistung

Die Power-Leitung des USB kann zur Spannungsversorgung von USB-Geräten genutzt werden. Dabei darf höchsten eine Belastung von 500mW bei 5V pro Gerät (engl.: Device) nicht überschritten werden. Die Spannung der Signalleitungen liegt ebenfalls bei 5V.

Einordnung in das Schichtenmodell

Der USB sieht drei Schichten vor (vgl. Abbildung 2.4):

- Function-Layer
- Device-Layer
- Interface-Layer

Die Daten werden gemäß des Kommunikationsweges über die Schichten transportiert. Auf der Host-Seite befindet sich der Host-Controller als Anschluss an den Bus, darüber ist die USB-System-Software und dann die Client-Software (vgl. Abbildung 2.19). Ein einfaches Gerät, auch *Function* genannt, benutzt ein Bus-Interface um mit dem Medium zu kommunizieren. Darüber ist ein Logical-Device und die eigentliche Function angeordnet. Zwischen den Schichten des Hosts sind entsprechende USB-Treiber als Protokoll implementiert.

Aufbau der Datenpakete

Das USB-Protokoll verwendet Pakete (engl.: Packets) zur Kommunikation. Jedes Paket ist in mehrere Felder aufgeteilt und beginnt mit dem SYNC-Feld. Dies besteht aus 31 Nullen gefolgt von einer Eins und dient zur Synchronisation der Abtasteinheiten. Dann

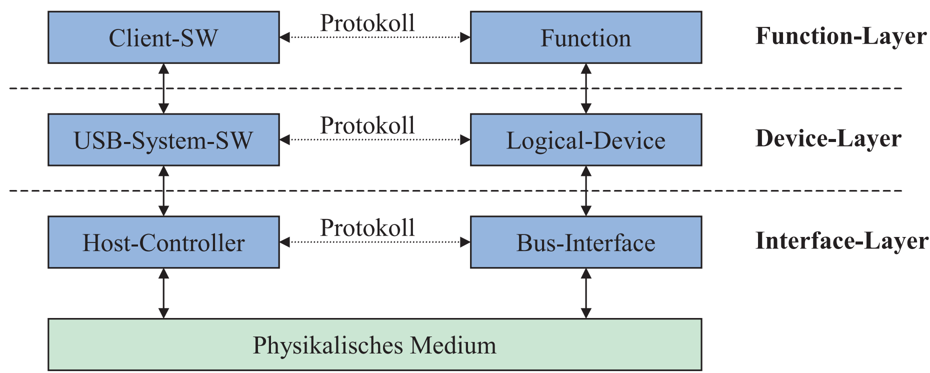


Abbildung 2.19: Einordnung der USB-Schichten

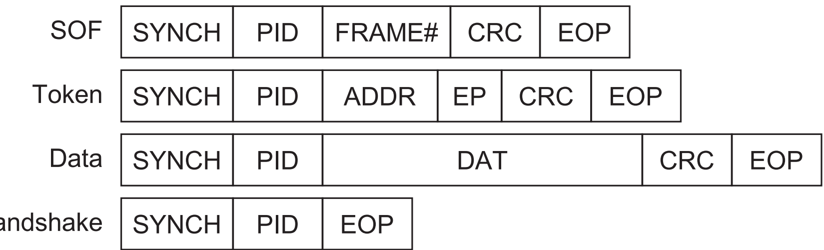


Abbildung 2.20: Aufbau der USB-Pakete

folgt das PID (Packet Identifier) -Feld, das über die spezielle Art der Daten Auskunft gibt. Am Ende steht dann jeweils ein EOP-Feld (End Of Package) (vgl. Abbildung 2.20).

Das SOF (Start-Of-Frame) -Paket wird zu Beginn jedes Frames vom Host als Synchronisationspaket versendet. Die darin enthaltene Frame-Nummer wird zyklisch durchlaufen, damit die angeschlossenen Geräte feststellen können, ob ein fehlerfreier Verkehr mit dem Host gewährleistet ist. Dabei handelt es sich beim SOF um einen Spezialfall des Token-Pakets. Mit dem Tokenpaket kann der Host anzeigen, wer als nächstes das Senderecht erhält. Das PID-Feld unterscheidet vier Typen: IN, OUT, SETUP, SOF. Bei OUT und SETUP wird der Host als nächstes an die angegebene Adresse im ADDR-Feld (Packet identifier) senden, ein IN bedeutet, dass das Gerät mit der entsprechenden Adresse als nächstes sendeberechtigt ist. Da jedes Gerät mehrere Endpoints, vergleichbar mit den Ports einer Netzwerkkarte, besitzt, muss auch diese Information in dem EP-Feld übermittelt werden. Dann folgt noch ein CRC (Cyclic Redundancy Checksum) -Feld zur Fehlerkorrektur und -Erkennung.

Das DATA-Paket übermittelt im Data-Feld neben einem CRC-Feld die eigentlichen Nutzdaten. Es können maximal 1024 Byte in ein Datenfeld geschrieben werden. Um bei langen Übertragungen eine Fehlerkontrolle über verlorene Pakete zu haben, werden die Werte des PID-Feldes zyklisch inkrementiert. Bei langen Übertragungen kann eine Data-Toggle-Synchronisation erfolgen, um mit der Datenübertragung synchron zu

bleiben.
Handshake-Pakete werden als Bestätigung des Datenverkehrs gesendet. Im isochronen Modus von USB kommen diese aber nicht vor, da sie bei einer fehlerhaften Übertragung die Wiederholung des Paketes auslösen. Eine verfälschte Information muss jedoch einer Wiederholung der Übertragung vorgezogen werden, die ein nicht-deterministisches Verhalten zur Folge hätte. Datenfehler können im isochronen Modus daher nur mit Hilfe einer Fehlerkorrektur des CRC beseitigt werden.

Übertragungsrate, Bandbreite

USB 2.0 Unterstützt 3 Übertragungsarten: *Low-, Full-* und *High-Speed* (LS, FS und HS). Damit können jeweils bis zu 1,5 MBit/s (LS), 12 MBit/s (FS) und 480 MBit/s (HS) übertragen werden. Die weitere Untersuchung wird sich auf den High-Speed (HS) Transfer beschränken. Die angegebenen Übertragungsraten beziehen sich auf alle übertragenen Bits, nicht nur auf die Nutzdaten.

Zeitverhalten

Es gibt vier verschiedenen Transferarten in USB:

- Control-Transfer
- Interrupt-Transfer
- Bulk-Transfer
- Isochronous-Transfer

Lediglich der isochrone Modus ist für Echtzeitanwendungen geeignet, da hier eine bestimmte Bandbreite und Latenzzeit zugesichert wird. Fehlerhafte Daten werden allerdings nicht wiederholt, da keine Handshake-Pakete versendet werden. Für jeden Übertragungszyklus wird ein *Frame* generiert, bei HS ein *Microframe*. Ein Frame existiert für die Zeitdauer von 1 ms, ein Microframe 125 μs. Es können maximal 90% eines Microframes für den Isochronen Modus genutzt werden. Die restlichen 10% werden für Control-Transfer freigehalten, der zur Initialisierung und Verwaltung von USB genutzt wird.

Handshake-Verfahren

Bei USB stehen für Daten- und Steuersignale nur die D$\pm$ -Leitungen zur Verfügung, d. h. es muss ein SW-Handshake geben. Eine Transaktion besteht aus mehreren Phasen, die sich je nach Transferart und Richtung unterscheiden: Zuerst sendet der Host ein Token-Paket, welches u. a. die Transferrichtung für nachfolgenden Datenpakete festlegt. Bei einem OUT- oder SETUP-Token sendet der Host in der 2. Phase. Bei einem IN-Token sendet die angesprochene Function. Im isochronen Modus entfällt die 3.Phase

der Empfangsbestätigung. Zur Fehlererkennung werden in der 2. und 3. Phase Timeout-Mechanismen verwendet.

Prioritätsvergabe, Zugriff auf das Medium

Der Host verwaltet neben dem Erkennen von hinzugefügten und entfernten Geräten sowie deren Spannungsversorgung, die Vergabe der Bandbreite. Datenverkehr kann lediglich vom Host iniziiert werden, d. h. ein Gerät bekommt Sendezeit nur auf Anfrage an den Host zugeteilt. Um echtzeitfähigen Verkehr zu ermöglichen muss ein Host implementiert werden, der zyklisch allen Geräten Sendezeit zuteilt, auch wenn diese nichts zu versenden haben. Die im System benötigten Zeiten müssen vorher statisch festgelegt werden und auf jede Function übertragen werden. Ein isochroner Endpoint gibt seine Bus-Zugriffs-Periode in Microframes an. Dabei sind Zweierpotenzen zwischen 1 und 32768 erlaubt. Jedem Endpunkt ist nur ein Datenpaket pro Frame erlaubt, dessen Datenfeld 1024 Byte groß sein darf. Eine Ausnahme davon bilden die High Bandwidth Endpoints. Diese können bis zu drei Datenpakete pro Frame verschicken. Daraus resultiert dann eine maximale Datenrate von 192 MBit/s.

Da USB eigentlich nicht für Echtzeitanwendungen ausgelegt ist, sind viele Restriktionen und Festlegungen notwendig, um ein deterministisches Zeitverhalten zu erreichen. Das macht USB nicht zum bevorzugten Kandidaten bei der Auswahl eines Echtzeitsystems.

2.2.6 FireWire

FireWire ist von der Firma Apple entwickelt und ist 1995 von der IEEE (Institute of Electrical and Electronics Engineers) zum IEEE 1394 Standard erklärt worden. Als man USB 2.0 veröffentlicht hat, folgte auch IEEE 1394a. Darin wurde die ursprüngliche Definition überarbeitet und Erweiterungen bzgl. Performance und Funktionalität vorgenommen. FireWire ist wie USB ein serielles Bussystem, erlaubt aber *Peer-to-Peer* Kommunikation, d. h. Geräte können Daten direkt und ohne Host untereinander austauschen. Ebenso wie USB ist auch FireWire Hot-Plug-and-Play fähig.

FireWire benutzt Memory Mapped I/O. Bei der Addressierung hält es sich an die Control Status Register Architecture (CSR-Architecture) für Mikrocomputer-Busse, die im ISO/IEC 13213 bzw. ANSI/IEEE 1212 Standard definiert ist. Eingesetzt wird ein 64Bit Adress-Modell des CSR (vgl. Abbildung 2.21).

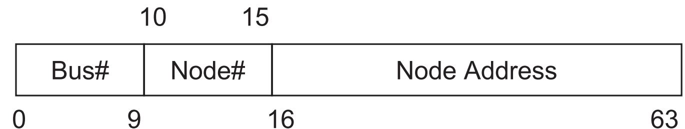


Abbildung 2.21: Adress-Modell von FireWire

Der Adressraum wird in drei Abschnitte eingeteilt. Die ersten 10 Bit spezifizieren ein

Bussegment. Die dem Bussegment zugeordneten Knoten (engl.: Node) werden durch die nächsten 6 Bit festgelegt. Bit 16 - 63 geben dann die jeweilige Adresse in einem Gerät an. Das ergibt einen Adressraum vom 256 TByte pro Gerät. Die Aufteilung des Adressraumes spiegelt sich in der Topologie von FireWire wieder. Über Brücken (engl.: Bridges) lassen sich bis zu 1023 (10 Bit) Bus-Segmente zusammenfügen. Darin können sich jeweils bis zu 63 (6 Bit) Geräte befinden. Das ergibt eine Gesamtsumme von maximal 64.449 anschließbaren Geräten, die entsprechend über Bus- und Node-Nummer anzusprechen sind. Ein Gerät kann zwischen 1 - 16 Ports haben. An Port 2 - 16 können weitere Geräte angeschlossen werden. Das führt, ähnlich wie bei USB, zu einer Baumstruktur (vgl. Abbildung 2.22). Ein Gerät mit nur einem Port wird als *Leaf-Node* bezeichnet, eines mit 2 oder mehr Ports als *Brach-Node*. Bis zu 16 Nodes können so im Daisy-Chaining-Verfahren hintereinander geschaltet werden. Die Angaben zu FireWire stammen aus folgenden Quellen: [And99] [Ada01] [App01].

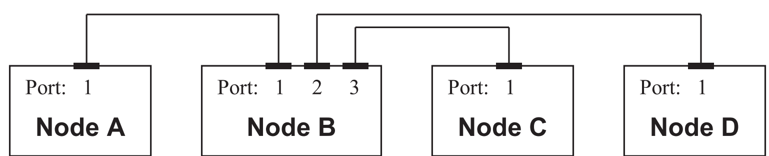


Abbildung 2.22: Topologie von FireWire

Übertragungsmedium

Aus ähnlichen Gründen wie USB verwendet auch FireWire Twisted-Pair-Kabel als Medium. Ebenso beherrscht FireWire „Hot Plug-and-Play“. Das Kabel ist aus sechs Adern aufgebaut: 2 Twisted-Pair-Adern, Masse und Power. Die maximale Länge für ein Kabel beträgt 4,5 Meter. In einem Segment könne bis zu 16 Nodes angeschlossen werden. Das ergibt einen maximalen Abstand von 72 m zwischen zwei Geräten in einem Segment.

Steckverbindungen

FireWire hat an beiden Kabelenden zwei gleiche RJ45 Stecker (vgl. Abbildung 2.14). Im Gegensatz zu USB können so regelwidrige Strukturen durch den Anwender verursacht werden, die den Bus dann außer Funktion setzen. Für Kleingeräte ist von der Firma Sony ein kleinerer Stecker mit nur vier Anschlüssen geplant. Die Spannungsversorgungsleitungen entfallen dort.

Signalform und Codierung

FireWire hat zwei Signalleitungen. Die Daten werden mit Hilfe einer NRZ (Non Return to Zero) -Codierung (vgl. Abbildung 2.23) auf der einen Leitung übertragen. Auf

der Anderen liegt ein Strobe-Signal (Abtast-Signal), das einen Pegelwechsel aufweist, wenn zwei gleiche Datenwerte aufeinander folgen. Mit Hilfe der XOR-Funktion kann dann aus dem Daten- und dem Strobe-Signal der Takt wiederhergestellt werden. Damit synchronisieren sich die beteiligten Geräte.

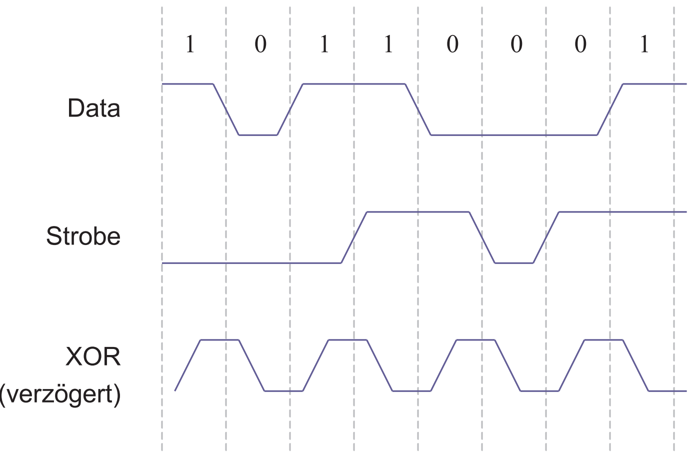


Abbildung 2.23: Signalcodierung von FireWire

Spannungspegel und Signalleistung

Die Spezifikation legt fest, dass als eine maximale Belastung von 1,5 A bei 8 - 40 V nicht überschritten wird. Ein Besonderheit bei FireWire besteht darin, dass Geräte auch Strom in das Netzwerk einspeisen können. Für die Kontrolle der Spannungsversorgung ist der Bus Manager der Bus-Management-Schicht zuständig.

Einordnung in das Schichtenmodell

Das Modell von IEEE 1394 hat drei Schichten: *Transaction-Layer, Link-Layer* und *Physical-Layer* (vgl. Abbildung 2.24). Bei isochronem Verkehr kommt der Transaction-Layer nicht zum Einsatz. Der Link-Layer generiert die Pakete, die dann über den Bus versendet werden und dekodiert die entsprechende Zieladresse der Kanäle bei eingehenden Paketen. Der Physical-Layer stellt die physikalische Verbindung zum Medium her. Außerdem ist er für Arbitrierung, De-/Kodierung und Resynchronisation des Datenstroms, sowie für die Initialisierung des Knotens, zuständig. Das Bus-Management kontrolliert die Konfiguration des Busses und verwaltet die Aktivitäten des zugehörigen Knotens. Bus-Management findet in allen drei Schichten statt. Nicht jeder Knoten muss den *Bus-Management-Layer* voll unterstützen, nur die automatische Bus-Konfiguration ist grundsätzlich immer zu implementieren. Diese Bus-Konfiguration erfolgt, wenn Geräte an den Bus angeschlossen oder von ihm entfernt werden. Es wird zuerst ein Bus-Reset ausgeführt, dann die neue Busstruktur festgelegt und die Adressen für die Nodes (IDs)

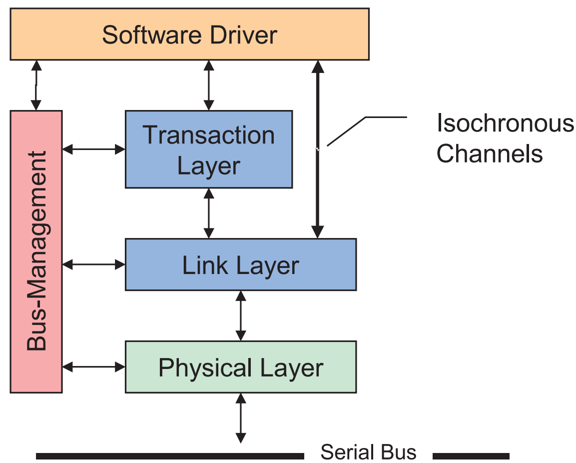


Abbildung 2.24: Bus-Modell von FireWire

vergeben. Danach werden die Rollen für die globalen Verwaltungsaufgaben verteilt. Es gibt drei dieser globalen Verwaltungsstellen in einem IEEE 1394 Bus:

- Cycle-Master
- Isochronous-Resource-Manager (IRM)
- Bus-Manager

Die Rolle des Cycle-Masters wird grundsätzlich von der Root-Node übernommen. Er ist für das Versenden der Cycle-Starts an alle Nodes verantwortlich und sammelt Informationen über die Bus-Topologie, die Stromversorgung und die von den Nodes unterstützte Geschwindigkeit. Die Aufgabe des IRM besteht aus der Verwaltung der Bandbreite für isochrone Transfers. Er stellt diese in Form von Kanälen (Time Slots) zur Verfügung und gibt nicht mehr benötigte wieder frei.

Aufbau der Datenpakete

Zu den zwei Transferarten gibt es analog auch zwei Datenpakettypen: Asynchrone und isochrone Pakete. Ein isochrones Paket enthält die Nummer des isochronen Kanals (engl.: Channel), den Transaktionstyp, sowie Daten und CRC. Acknowledgement-Pakete werden für den isochronen Verkehr nicht benötigt. Ein Paket kann maximal 4096 Byte an Daten fassen, bei einer Übertragungsrate von 400 MBit/s.

Übertragungsrate, Bandbreite

Zur Zeit sind Übertragungsraten von 100, 200 oder 400 MBit/s möglich. Geplant sind Raten von 800 MBit/s, 1,6 und 3,2 GBit/s (IEEE 1394b).

Zeitverhalten

FireWire ist ein *serielles Protokoll*. Für jede Transaktion gibt es einen Sender (Requester) und einen Empfänger (Responder). Es gibt zwei verschiedenen Transferarten, den asynchronen und den isochronen Verkehr. So, wie bei USB die Buszeit in Frames eingeteilt wird, gibt es bei FireWire die Bus-Zyklen (engl.: Bus-Cycles). Ein Bus-Zyklus teilt sich in das Cycle-Start-Paket, isochrone Pakete und danach die asynchronen Pakete auf (vgl. Abbildung 2.25). Alle 125 μs wird ein Cycle-Start-Paket über den Bus geschickt. Maximal 80% eines FireWire-Zyklus können für den isochronen Verkehr genutzt werden. Danach folgen dann mindestens 20% asynchroner Verkehr. Zwischen den Paketen erfolgt jeweils eine kurze Arbitrierungphase. Ausschließlich der isochrone Modus ist für den Echtzeitverkehr geeignet.

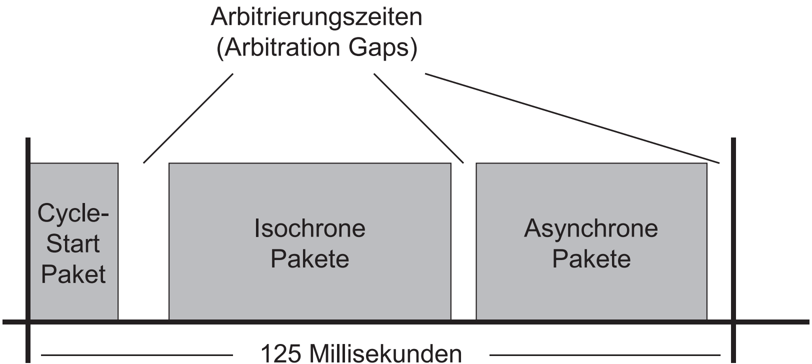


Abbildung 2.25: Bus-Zyklus bei FireWire

Wie bereits erwähnt, benutzt FireWire zwei Signalleitungen, auf denen synchron gesendet wird. Die Synchronisation erfolgt, wie in „Signalform und Codierung“ beschrieben, durch eine XOR-Verknüpfung aus Daten- und Strobe-Kanal.

Handshake-Verfahren

Eine Transaktion findet durch den Versand einer Reihe von Daten- und Informationspaketen statt. Bei isochronem Verkehr wird einer der 64 Kanäle (*Isochronous Channel*) adressiert, der von allen Geräten abgehört werden kann, was *Multicast* ermöglicht. Es wird hier kein Response geschickt und keine Fehlererkennung durchgeführt, im Gegensatz zur asynchronen Übertragung. Die Berechtigung, um auf den Bus zu schreiben wird durch eine Arbitrierungphase bestimmt.

Prioritätsvergabe, Zugriff auf das Medium

Der Zugriff auf das Medium erfolgt in den Arbitrierungsphasen. Jeder Knoten, der ein Request stellen will, muss die Bestätigung von der Root-Node, die sich als oberster Knoten in der Baumhierarchie befindet, erhalten. Dazu wird die Anfrage über die jeweiligen

Vaterknoten bis zur Root-Node hochgereicht. Theoretisch könnten beliebig viele Knoten eine Anfrage stellen. Die Root-Node entscheidet dann, wer das Senderecht erhält. Das wird über den Bus mittels Broadcast bekannt gegeben. Nach einer gewährten Sendephase hat der Knoten einen gewissen Zeitraum mit der nächsten Anforderung zu warten, um im Bussystem Fairness zu gewährleisten.

Für die Echtzeitfähigkeit muss das System leicht modifiziert werden. So muss jeder Knoten im Bus regelmäßig zu seinen vorherbestimmten Zeitpunkten ein Request stellen und der Root-Node, auf dem dann auch der IRM implementiert ist, muss diesen Sendewunsch bestätigen. Da jeweils genau ein Request vorliegt, kann es nicht zu Kollisionen kommen und der IRM hat keine Probleme mit der Zuteilung des Busses. Das Warten ist dann allerdings nicht mehr zufällig, sondern richtet sich nach dem globalen „Arbitrierungsplan“.

2.2.7 Time Triggered Protocols (TTP)

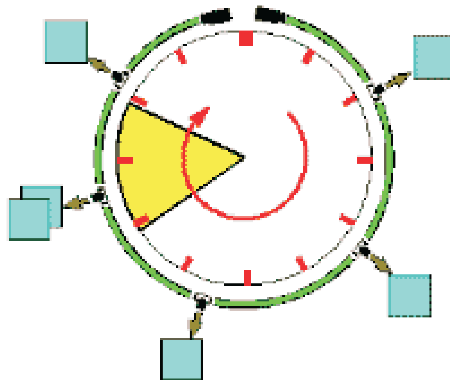


Abbildung 2.26: Logo des TTA-Projektes [ESP98]

In Industrie und Forschung gibt es mehrere Ansätze ein Protokoll zu beschreiben, das geeignet ist, die Kommunikation in verteilten Echtzeitsystemen zu behandeln. Als ein erfolgversprechender Ansatz hat sich die TTA (Time Triggered Architecture) herausgestellt. Das Time Triggered Protocol (TTP) stellt die Protokollschichten innerhalb der TTA dar. Eine Einordnung von TTP in den Kontext der TTA liefert die Abbildung 2.27: Verschiedene Anwendungsgebiete verfügen über kommunizierende, fehlertolerante Einheiten, die auf einen Datenbus zugreifen. Zur Steuerung dieser Einheiten wird neben der Alternative eines Emulations-Boards das TTP eingesetzt. Weitere Informationen dazu finden man auf der folgenden Webseite: [ESP98]. Das Logo der TTA (vgl. Abbildung 2.26) illustriert auf anschauliche Weise den Gedanken der TT-Modelle: Ein Cluster von Echtzeitanwendungen, die auf einen Kanal zugreifen, sind in Form eines Zifferblatts angeordnet. Dies verdeutlicht die dominante Rolle der *Zeit* in einem Echtzeitsystem und veranschaulicht, dass einzelnen Knoten genau festgelegte Zeitscheiben zugeordnet sind. Einen ähnlichen Ansatz wie TTP verfolgt das Projekt *FelxRay* [Fle01, DEC01].

TTP ist kein fest definiertes Protokoll im Sinne von USB oder FireWire. Vielmehr ist eine

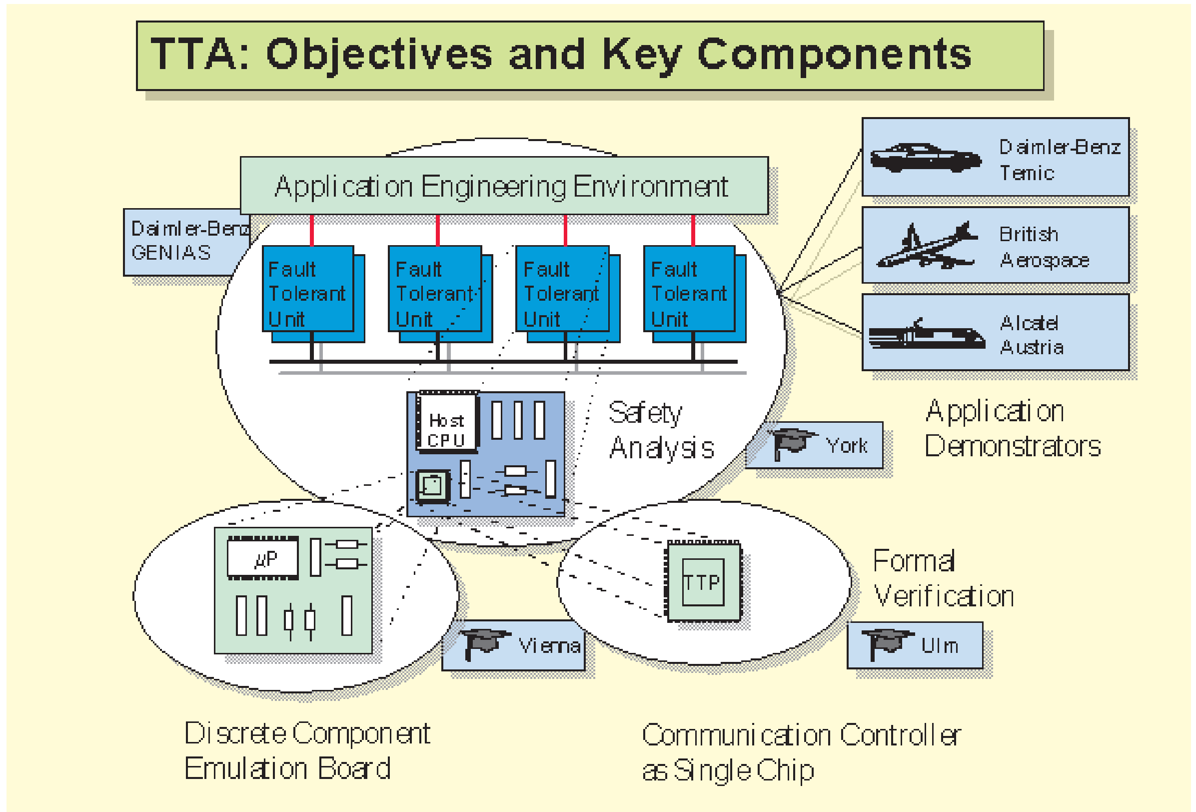


Abbildung 2.27: Das Schema von TTA

Protokollklasse erschaffen worden, mit der die besonderen Anforderungen eines *fehlertoleranten verteilten Echtzeitsystems* zu beschreiben sind [Kop01]. Es gibt zwei Versionen von TTP: Das *TTP/C* und *TTP/A*. In beiden Fällen ist das Protokoll zeitgesteuert (engl.: Time-Triggered) und für den Einsatz in harten Echtzeitumgebungen konzipiert. Der Unterschied besteht in der erreichbaren *Performance* und dem realisierten *Funktionsumfang*. TTP versucht möglichst viel Flexibilität zu gewähren, der Determinismus der Datenübertragung muss jedoch gewahrt bleiben. Das bedeutet, dass die *analytische Vorhersagbarkeit* zu jedem Zeitpunkt möglich sein muss.

Bei der Entwicklung des TTP-Protokolls waren folgende Ziele maßgebliches Entwurfskriterium [Kop01]:

- Nachrichtentransport mit geringer Latenz und minimalem Jitter (Abweichung der Übertragungsverzögerung vom Mittelwert)
- Unterstützung von Composability (Aufnahme von Subkomponenten in ein Design bei absoluter Unterstützung aller Subfunktionen und Eigenschaften)
- Angebot der Anordnung in fehlertolerante Gruppen
- Fehlertolerante Takt-Synchronisation

- Verteilte Verwaltung der Redundanz
- Minimalen Overhead, bzgl. Nachrichtenlänge und -menge
- Skalierbarkeit von hohen Datenraten, sowie effiziente Operationen auf Twisted-Pair- und Glasfaserkabel.

Wie schon erwähnt gliedert sich TTP in zwei Varianten. Ein Vergleich der unterstützten Dienste ist in Tabelle 2.4 gegeben. Mit TTP/C kann die Implementierung eines fehlertoleranten, in Cluster (sinngemäß: Gerätegruppe) aufgeteilten, Echtzeitsystems erfolgen. Um die Ausfallsicherheit zu erhöhen, werden FTUs (Fault Tolerant Units) in mehrfacher Ausführung eingesetzt, die dann auch entsprechend viele Übertragungskanäle erfordern. Auf den einzelnen FTUs können unterschiedliche Strategien gegen den Ausfall eines Knotens implementiert werden. Bei TTP/C übernimmt ein Controller die Kommunikation mit dem Medium. Die Protokoll-Funktionen sind dafür in Hardware bereitgestellt.

TTP/A ist eine „abgespeckte“ Version des vollständigen TTP/C. Die spartanische Variante ist für die Implementierung kostengünstiger Applikationen, z. B. eines Feldbus-Systems gedacht. Der Einsatz erfolgt bei Bussystemen, die nicht fehlertolerant sein müssen bzw. dort wo *nicht mit dem Ausfall von Konten* zu rechnen ist. TTP/A benötigt als HW-Komponente nur einen einfachen *UART*-Baustein und eine *lokale Echtzeituhr*. Die Protokoll-Logik kann in einem einfachen Mikrocontroller implementiert werden. Im Gegensatz zu TTP/C ist TTP/A ein *Multi-Master* und kein verteiltes Protokoll.

Tabelle 2.4: Unterstützte Dienste von TTP/A und TTP/C [Kop01]

Dienst	TTP/A	TTP/C
Takt-Synchronisation	zentraler Multimaster	verteilt, fehlertolerant
Modus Wechsel	ja	ja
Übertragungsfehler-Erkennung	Parität	16 / 24 Bit CRC
Membership-Funktion	einfach	vollständig
Externe Takt-Synchronisation	ja	ja
Zeitredundante Übertragung	ja	ja
Duplex-Knoten	nein	ja
Duplex-Kanäle	nein	ja
Redundanzverwaltung	nein	ja
Schatten-Knoten	nein	ja

Übertragungsmedium, Steckverbindungen, Signalform und Codierung, Spannungspegel und Signalleistung

Da TTP eine Protokollklasse beschreibt, können nur allgemeine Angaben über die HW-Realisierung gemacht werden. Entscheidend ist, dass alle eingesetzten HW-Komponenten

die im TTP Modell vorgeschriebenen Richtlinien erfüllen. Es gibt bereits praxisnahe Entwicklungen im Bereich der Automobil-, Flugzeug- und Eisenbahnindustrie. Ein Anwendungsbeispiel ist die Steuerung der Lenkung und der Bremsen eines Autos mit der *X-by-Wire*-Technologie (vgl. Abbildung 2.28) [TTT01]. Man verzichtet dabei auf jede mechanische Kopplung zwischen Steuereinheit und Aktor. Die gemessenen Daten eines Steuerrads (*Wheel*) transportiert man anstelle dessen mit einem TTP-System zu den Aktoren (vgl. Abbildung 2.28). Für viele Firmen und Universitäten ist TTP aktuelles Forschungsthema, da sich herausgestellt hat, dass TTP ein für die Praxis relevantes System ist, das in ES eingesetzt werden kann. Erste Fallstudien sind bereits realisiert, es sind jedoch noch viele Fragen offen und eine Reihe von Problemen zu bewältigen.

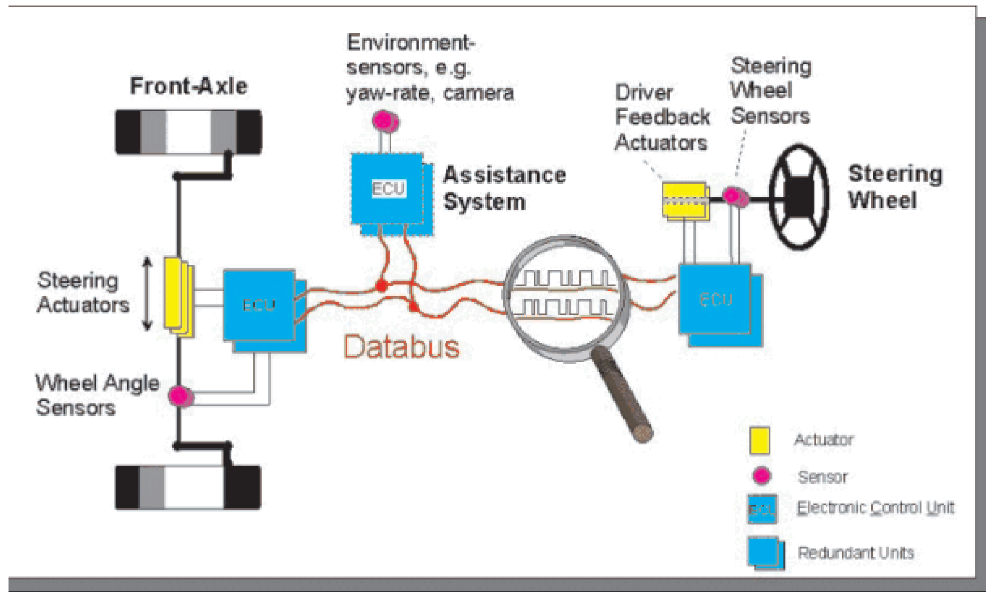


Abbildung 2.28: Ein Beispiel für ein Bussystem von X-by-Wire

Für TTP muss ein echtzeitfähiger Kanal eingesetzt werden, sowie ein Controller, der diesen in Echtzeit ansprechen kann. Angedachte Medien sind Twisted-Pair- und Glasfaserkabel (vgl. Kapitel 2.1). Die Signale und ihre Codierung richten sich nach der Art des Mediums. Bei TTP/C wird die Modified-Frequency-Modulation (MFM) eingesetzt. Als Kanal kann sogar ein CAN-Bus eingesetzt werden, da der Anforderungskatalog dort noch um die Bitarbitrierung erweitert ist. Die Signalleistung ist abhängig von der zur Verfügung stehenden Energie. Je mehr Energie in das Sendesignal geleitet wird, desto stärker und störunanfälliger wird es. Für die Spannungspegel gelten ähnliche Bedingungen: Je größer der Spannungsunterschied ist, desto besser ist die Übertragung gegen Rauschen geschützt. Allerdings hat eine hohe Amplitudendifferenz langsame Umschaltzeiten zur Folge, da ein Wechsel der Pegel nicht in beliebig kurzer Zeit erfolgen kann.

Im Demonstrator wird die Variante TTP/A eingesetzt. Zur Übertragung wird ein UART-Baustein (V24-Verkehr) genutzt. Dafür gelten die entsprechenden Angaben aus dem Abschnitt 2.2.3.

Einordnung in das Schichtenmodell

Die Struktur von TTP wird in Abbildung 2.29 veranschaulicht. Ein Cluster mit fehlertoleranten Einheiten (FTUs), das jeweils aus einem, zwei oder mehreren Knoten (engl.: Nodes) besteht, wird über ein Bussystem verbunden [Kop01].

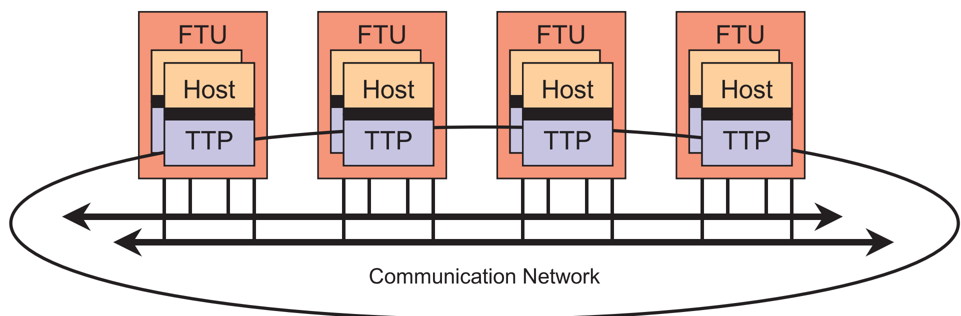


Abbildung 2.29: Hardware Struktur von TTP

Eine fehlertolerante Einheit ist ein Gebilde, das nach außen hin eine abgeschlossene, fest definierte Aufgabe erfüllt. Um die FTU fehlertolerant zu machen, muss in ihr Redundanz enthalten sein. Dazu besteht eine FTU aus den sogenannte Knoten, die auch als SRU (Smallest Replaceable Unit) bezeichnet werden und nach außen eine identische Funktionalität aufweisen. Die interne Realisierung kann von einander abweichen. Im Fehlerfall stellt eine SRU die kleinste neu zu konfigurierende oder austauschbare Einheit im System dar. Ein Knoten besteht aus zwei Komponenten, dem Host und dem für die Kommunikation zuständigen Controller (vgl. Abbildung 2.30). Deren Schnittstelle wird als CNI (Communication-Network Interface) bezeichnet. Das CNI ist aus einem DPRAM (Dual-Ported Random-Access Memory) aufgebaut, welches sowohl dem Host

als auch dem Controller den gleichzeitigen Lese- und Schreibzugriff auf die Schnittstelle ermöglicht. Dazu wird ein NWB (Non Blocking Write) -Protokoll genutzt.

Der Kommunikations-Controller jedes Knotens verfügt über einen lokalen Speicher, der die MEDL (Message Descriptor List) beinhaltet. In die MEDL ist unter anderem die statische Bus Arbitrierung eingetragen, die einem Knoten sagt, wann er Daten zu empfangen oder zu senden hat. Die Größe der MEDL bestimmt sich aus der Anzahl der TDMA (Time Division Multiple Access) -Runden, die zur Übertragung verwendet werden.

Zusätzlich ist in allen Knoten noch für jeden Bus im Kommunkationsnetzwerk ein BG (Bus-Guardian) vorhanden. Das ist eine unabhängige HW-Einheit, die den zeitlichen Zugriff auf das Medium überwacht. Im Falle eines fehlerhaften Zugriffs wird die Verbindung zum Controller unterbrochen.

Ein Schichtenmodell existiert nur für TTP/C (vgl. Abbildung 2.31). Da in dieser Arbeit im Folgenden das TTP/A Protokoll benutzt wird, sind die Anmerkungen zu TTP/C weniger ausführlich erläutert. Eine genaue Beschreibung des Schichtenmodells zu TTP/C kann in [Kop01] nachgesehen werden.

Die einzige Schnittstelle von TTP, auf die eine Applikation Zugriff hat, ist das CNI (Communication Network Interface), das sich in das Status / Kontroll-Feld (vgl. Abbildung 2.32) und das Datenfeld aufteilt. Das Datenfeld speichert die vom Knoten gesendeten oder empfangenen Daten mit einem zusätzlichen Kontrollbyte ab. Das Status- und Kontrollregister ermöglicht eine Kommunikation zwischen Host-CPU und TTP-Controller, wobei das Kontrollregister vom Controller und das Statusregister durch die CPU beschrieben wird.

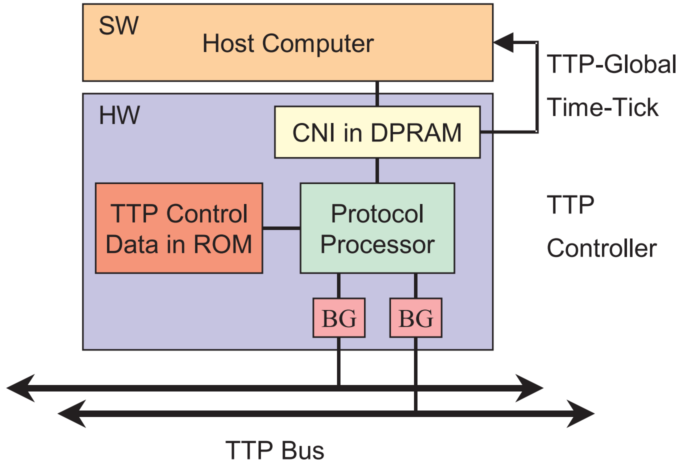


Abbildung 2.30: Hardware Struktur eines TTP Knotens

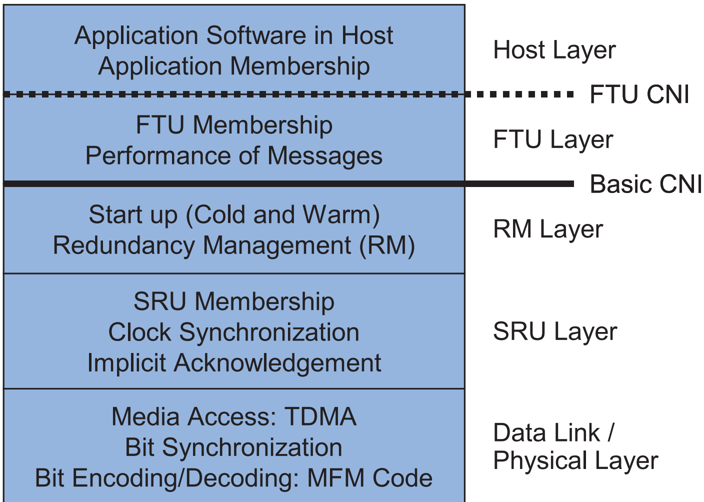


Abbildung 2.31: Schichtenmodell des TTP/C

Status Registers	Control Registers
Global Internal Time	Watchdog
SRU-Time (part of C state)	Timeout Register
MEDL (part of C state)	Moden Change Request
Membership (part of C state)	Reconfiguration Request
Status Information	External Rate Correction

Abbildung 2.32: Kontroll / Daten Feld des CNI

Übertragungsrate, Bandbreite

Die Übertragungsrate wird durch den langsamsten Kennwert des Kommunikationsweges begrenzt. Das kann die begrenzte Bandbreite des Mediums, die Zugriffs- bzw. Verarbeitungsgeschwindigkeit des Controllers oder der eingeschränkte Taktrate der Echtzeituhr sein. Es ist in jedem Fall zu gewährleisten, dass die Daten zeitgerecht übermittelt werden. Selbstverständlich wird eine fehlerfreie Übertragung, die auch die inhaltliche Korrektheit gewährleistet, angestrebt.

Zeitverhalten

Dem Zeitverhalten kommt in Echtzeit-Systemen, so auch bei TTP, eine besondere Bedeutung zu. Um ein deterministisches Zeitverhalten garantieren zu können, wird schon zur Design-Zeit eine statisch festgelegte Busarbitrierung ermittelt, die dann in die MEDL eingetragen wird (vgl. Abbildung 2.33).

SRU -Time	Address	Attributes			
		D	L	I	A

Abbildung 2.33: Die Message Descriptor List (MEDL) des TTP

Das bietet die Option *a priori* eine optimale Verteilung der Sendezeitpunkte zu ermitteln. Die Vorteile dieses Verfahrens liegen auf der Hand:

- Ein Empfänger kann ein verlorenes Paket sofort detektieren, falls die dafür bestimmte Empfangszeit abläuft.
- In das Datenpaket brauchen werden Sender (S) noch Empfänger (E) eingefügt zu werden, da allen Knoten im Netz genau wissen wer zum aktuellen Zeitpunkt S und E ist.
- Bei der Nachrichtenbestätigung (engl.: Acknowledgement) von empfangenen Paketen nutzt man den Vorteil des Broadcasting-Systems aus. Wenn mehrere Empfänger gleichzeitig angesprochen werden, muss nur einer davon eine Bestätigung zurücksenden. Das folgt aus der Annahme, dass durch das allgemeine Broadcast jede funktionsfähige FTU ihre Nachricht erhält. Entweder wird sie alle oder keinen Empfänger erreichen. Allerdings ist eine solche Aussage nur gültig, wenn man einen partiellen Ausfall des Kommunikationsnetzes vernachlässigen kann.
- Ein weiterer Aspekt ist die dadurch ermöglichte „lautlose Fehlerverwaltung“ (engl.: Fail Silence) bezüglich des Zeit- und des Datenbereichs. Ein temporaler Fehler muss niemandem gemeldet werden, er ist, da alle Knoten synchronisiert sind, bereits allen bekannt. Für die Einhaltung der temporalen Korrektheit sorgt der Bus-Guardian. Er beobachtet die ein- und ausgesendeten Daten und trennt den Controller vom Bus für den Fall einer temporalen Zugriffsverletzung. Sollte der Fehler durch einen defekten Knoten entstanden sein, so sorgt die bereitgestellte Membership-Funktion dafür, dass binnen kürzester Zeit ein noch korrekter Knoten der FTU den Ersatz übernimmt. Fehler im Datenbereich kann die Applikation bis zur a priori bekannten Sendezeit noch beheben. Für entstehende Übertragungsfehler wird ein CRC eingesetzt.

Ein wesentlicher Aspekt, der den Aufbau der MEDL bestimmt, ist unter anderem das Buszugriffsverfahren. Eingesetzt wird TDMA, das zum Ziel hat, eine statischen verteilten Buszugriff unter Echtzeitbedingungen zu gewährleisten. TDMA benötigt eine fehlertolerante Uhr in jedem Knoten. Damit kann man dann die Sendebandbreite statisch in feste Zeitscheiben einteilen. Diese werden dann unter den Knoten aufgeteilt, so dass jeder

Knoten genau einen Zeitabschnitt erhält. Diese Aufteilung bezeichnet man als *TDMA Runde* (engl.: TDMA-Round). Damit ist sichergestellt, dass jeder Knoten in eine TDMA Runde senden kann, aber auch muss. Falls ein Knoten im Zeitabschnitt der eigenen Sendeberechtigung keine Daten zu verschicken hat, wird ein Leerrahmen übertragen. Diese TDMA Runden werden in Folge immer wiederholt. Der Inhalt der Frames kann sich dabei von der Vorrunde unterscheiden. Alle so entstehenden TDMA Runden werden zum *Cluster Cycle* zusammengefasst. Der Zeitverlauf dieses Cluster Cycle wird dann in die MEDL eingetragen.

Bei TTP/A wird das Fortschreiten in der MEDL durch Time-Outs gesteuert. Die Resynchronisation kann mit jedem Empfang eines Fireworks-Bytes aufs neue erfolgen. Dazu wird das RDI (Receive Data Interrupt) -Signal des UART Bausteins als globales Synchronisationssignal eingesetzt. Für den Fall, dass der Bus Master ausfällt, wird einer der anderen Knoten aktiv und übernimmt die aktive Rolle das Masters. Ein defekter Bus Master wird durch das Fehlen des Fireworks-Bytes, zu seiner a priori festgelegten Zeit, erkannt.

Um bei dem Aufbau eines TTP/A Systems die Echtzeitfähigkeit zu überprüfen, müssen die einzelnen Zeiten, die im Kommunikationsweg auftreten, bekannt sein (vgl. Abschnitt 2.3). Dazu zählen die Bandbreite des Kanals, die Latenzzeiten der Hardware, die Reaktionszeit des BS (Betriebssystem) vom Knoten, die Granularität des Takts und die Zeit, die für die Protokolllogik verbraucht wird. Daraus resultieren dann die Time-Out Werte der MEDL, die den nächsten Datentransfer initiieren.

Aufbau eines Datenpakets

Der Paketaufbau bei TTP/C entspricht dem in Abbildung 2.34 vorgegebenen Schema. Das Paket beginnt mit einem Header, gefolgt vom Datenfeld. Zum Schluss ist dann noch ein CRC angefügt.

Abbildung 2.34: Datenpaket von TTP/C

Ein Datenpaket von TTP/A hat die Form eines normalen V24-Pakets (vgl. Abschnitt 2.2.3). Es gibt allerdings eine spezielle Anordnung der Pakete im Kommunikationszyklus. Es gibt lediglich ein festes Steuerbyte, das *Fireworks-Byte*, welches zu Beginn jeder TDMA-Runde vom z. Z. aktiven Master geschickt wird. Danach folgen dann die eigentlichen Nutzdaten, die jeweils Byteweise in V24-Paketen verschickt werden. Das Fireworks-Byte wird als globales Synchronisationsereignis für eine neue TDMA Runde und zur Festlegung der aktuell verwendeten MEDL genutzt. Zwischen dem Fireworks-Byte und den Nutzdaten ist eine etwas längere Sendepause einzuhalten, in der die Erkennung des Fireworks-Bytes und die darauf folgende Synchronisation aller Knoten durchgeführt wird. Mit der Übertragung des letzten Paketes endet eine TDMA Runde. Die

nächste Runde beginnt dann wieder mit einem Fireworks-Byte und ist unabhängig von der Vorgängerrunde.

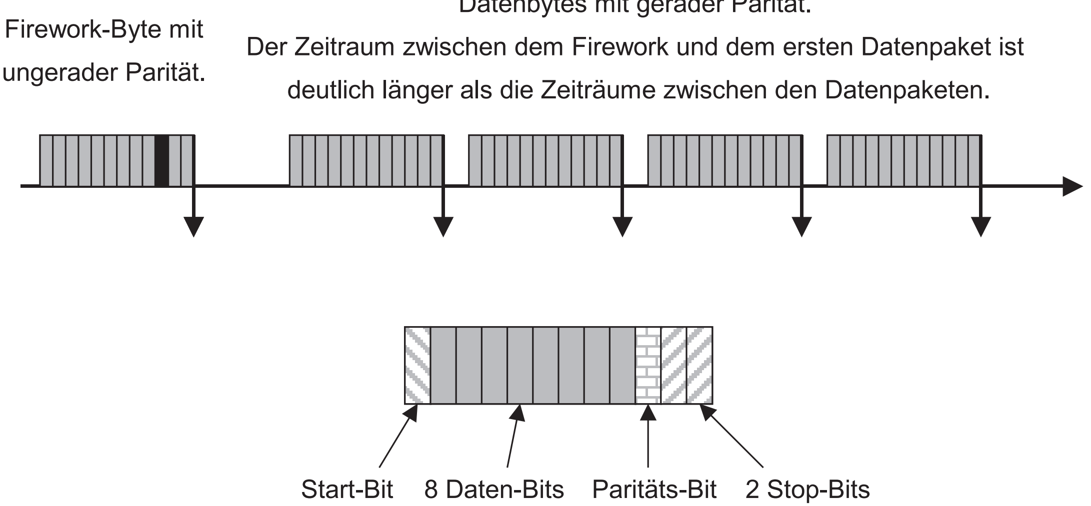


Abbildung 2.35: Datenpaket von TTP/A

Um in der Lage zu sein ein Fireworks-Byte von einem Nutzdaten-Byte zu unterscheiden, wird das Parity-Bit eingesetzt. Im Fireworks-Byte wird die Parität ungerade ergänzt, bei den Daten gerade.

Jede MEDL organisiert genau eine TDMA Runde. Die sich wiederholende Übertragung von gleichen TDMA Runden wird als ein Modus bezeichnet. Ein Modus-Wechsel kann vom aktuellen Bus Master zu Beginn einer TDMA Runde im Fireworks-Byte angezeigt werden.

Handshake-Verfahren

Einer der großen Vorteile von TTP ist, dass kein Handshake benötigt wird. In der MEDL sind alle Sende- und Empfangszeiten protokolliert. Ein Knoten, der mit Senden oder Empfangen an der Reihe ist, legt seine Daten auf die Leitung bzw. hört den Kanal ab. Die Sende- oder Empfangsbereitschaft eines Knotens ist dabei Grundvoraussetzung, muss also entsprechend auch bei der a priori stattfindenden Planungsphase der MEDL berücksichtigt werden.

Prioritätsvergabe, Zugriff auf das Medium

Der Zugriff auf das Medium erfolgt zu festgelegten Zeitpunkten über die Bus-Guardians. Diese Zeitpunkte sind in einer statischen Datenstruktur, der MEDL, gehalten, die in jedem TTP Controller vorkommt. Ein MEDL Eintrag besteht aus drei Feldern: Dem Zeit-Feld, dem Adress-Feld und dem Attribut-Feld (vgl. Abbildung 2.33). Das Zeit-Feld

legt fest, zu welcher globalen Zeit das Datum, welches im Adress-Feld beschrieben wird, für die Kommunikation vorgesehen ist. Das Adress-Feld verweist auf einen Speicherplatz im CNI-Speicher. Dort wird die entsprechende Nachricht entweder gelesen oder abgelegt. Das Attribut-Feld enthält vier Einträge:

- D (Direction): Das Richtungsbit legt fest, ob die Nachricht Input oder Output ist.
- L (Length): Gibt die Länge der zu übertragenden Nachricht an.
- I (Initialization): Gibt an, ob es sich bei dem Datum um eine normale oder eine Initialisierungs-Nachricht handelt.
- A (Additional Parameter): Gibt zusätzliche Informationen zu Modus- und Knoten-Wechseln.

Ein Host kann nur Modus-Wechsel bewirken, die auch in der MEDL vorgesehen sind. Zur effizienten und korrekten Erzeugung der MEDL werden Tools eingesetzt, die bereits eine automatische Generierung ermöglichen. Ein solches Tool ist z. B. der *Cluster Compiler*, vorgestellt in [Kop95].

2.3 Determinismus

Determinismus liegt vor, wenn sich ein System streng gesetzmäßig mit der Zeit entwickelt [Bec01]. *Determinismus* bedeutet die exakte *Vorhersagbarkeit* des Verhaltens eines Systems auf alle möglichen Eingaben zum aktuellen Zeitpunkt. Der Gegensatz zu deterministisch wäre zufällig oder stochastisch.

Als Determinismus kann man hier auch die *analytische Vorhersagbarkeit* bezeichnen. Diese bezieht sich nicht nur auf das Zeit-, sonder auch auf das Zustands-Verhalten. Je nach Art des Systems kann dieses Verhalten wie folgt dargestellt werden: In der HW können Zustände durch Spannungspegel oder Flanken auf Leitungen oder in Speicherelementen dargestellt werden. Die Datenbusbelegung und Registerinhalte sind die entsprechenden Äquivalente in der SW. Die jeweiligen Werte müssen, den physikalischen Gesetzten entsprechend, ebenso wie der Zeitaspekt eindeutig vorausberechenbar sein.

Ein deterministischen Verhalten kann nur garantiert werden, wenn das gesamte Systemverhalten hinreichend bekannt ist. Das bedeutet, dass die Laufzeiten und Übergangsfunktionen der kleinsten Elemente nachvollziehbar sind. Daraus lässt sich dann in einem *Bottom-Up*-Verfahren das gesamte Systemverhalten spezifizieren. Die zeitliche Vorhersagbarkeit ist Grundlage einer jeden Echtzeit-Betrachtung. Damit beschäftigt sich das folgenden Unterkapitel 2.4.

2.4 Echtzeit

Echtzeit (engl.: Real-Time) ist ein Begriff, der in den letzten Jahren an Bedeutung gewonnen hat. Eingebettete Systeme übernehmen im täglichen Leben immer mehr Aufgaben,

teilweise hinter den Kulissen, dem Anwender verborgen. Wenn das Aufgabenfeld eines solchen Systems von spielerischen Anwendungen, wie z. B. dem Asimo von Honda in anwendungskritische (engl.: Mission-Critical) Bereiche wandert (vgl. Abbildung 2.36), muss entsprechend auch das zeitliche Verhalten wesentlich exakteren Anforderungen genügen um hohen Schach- oder Personenschaden zu vermeiden. Corollary beschreibt die Anforderungen folgendermaßen: „Ein System muss immer so ausgelegt sein, um der schlimmst möglichen Kombination von Umständen bestehen zu können."

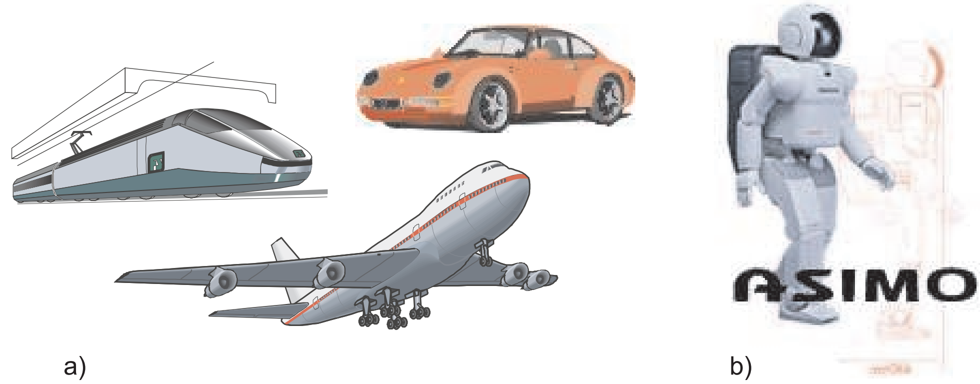


Abbildung 2.36: a) Harte Echtzeitanwendungen, b) Weiche Echtzeitanwendungen: der Asimo der Firma Honda

Literaturangaben über Konzepte und formale Definitionen von Echtzeitsystemen (Real-Time Systems) enthalten folgender Bücher: [But00, Bur01, Tin95, Ver93, Lap97]. Es finden sich dort auch Angaben zu Beweisbarkeit und Validierung von Echtzeit. Weitere Anregungen zum Thema Echtzeit entstammen den Vorlesungen *RTOS, Real Time Operating Systems* von Prof. Franz Rammig [Ram01], *Hardware / Software-Codesign* von Prof. Jürgen Teich und Dr.habil Wolfram Hardt [HT00] und *Eingebettete Systeme* von Bernd und Lisa Kleinjohann [Kle00].

2.4.1 Definition

Es existieren mehrere bekannt gewordenen Definitionen von Echtzeit. Eine aus neuerer Zeit stammt von Herrmann Kopetz [Kop01]:

„A real-time computer system is a computer system in which the correctness of the system behaviour depends not only on the logical results of the computation, *but also on the physical instant at which these results are produced*".

Der hauptsächliche Unterschied zu einer allgemeinen Computerberechnung liegt darin, dass nicht nur die logische Korrektheit gegeben, sondern auch eine Zeitschranke (engl.: deadline) eingehalten werden muss. Echtzeit bedeutet dabei, dass ein Reaktion auf externe Signale mit deren Geschwindigkeitsmaß erfolgen muss. Das entsprechende System benötigt dazu eine Zeitbasis, die mindestens so schnell wie die des Umfeldes ist. Dabei

Abbildung 2.37: Bedeutung der Echtzeit [Ram01]

muss ein echtzeitfähiges System nicht einmal besonders schnell reagieren, es ist lediglich eine Frage des Bezugssystems (vgl. Abbildung 2.37). Wichtig ist lediglich, dass die gegebenen Zeitschranken auch unter den widrigsten Umständen eingehalten werden.

Eine Echtzeitrestriktion umfasst alle Prozesse, die zwischen dem Erfassen einer Information und der Ausgabe der berechneten Reaktion beteiligt sind. Dabei spielt es keine Rolle, ob ein Prozess in Hardware oder Software implementiert ist. Entstehen, bedingt durch die Systempartitionierung, Schnittstellen zwischen Ausführungseinheiten, die einer Zeitrestriktion unterliegen, so werden diese Schnittstellen unter Berücksichtigung der selben Zeitrestriktionen im System aufgenommen und entworfen.

Echtzeit kann in vier Kategorien unterteilt werden (vgl. Abbildung 2.38). Zum einen wird *weiche* und *harte* Echtzeit unterschieden, zum anderen die *statische* und die *dynamische* Echtzeit. Diese Unterscheidungsmerkmale liegen orthogonal zueinander, so dass vier Kombinationsmöglichkeiten entstehen. Statische Echtzeit bedeutet, dass die Ablaufplanung des Systems schon vor Inbetriebnahme berechnet werden kann. Die dynamische Echtzeit dagegen bezeichnet ein System, dass die Berechnung einer Ablaufplanung während des Betriebs erfordert, da sich erst zur Laufzeit die notwendigen Informationen ergeben. Die Unterscheidung zwischen harter und weicher Echtzeit wird im folgenden Abschnitt vorgestellt und an Beispielen ausführlich erläutert.

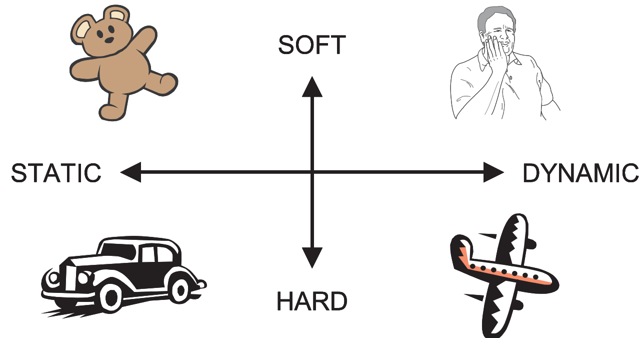


Abbildung 2.38: Einteilung der Echtzeit [Ram01]

2.4.2 Anwendungsbezug

In der täglichen Anwendung begegnet man sowohl Bespielen für harte als auch weiche Echtzeit (vgl. Abbildung 2.36). Weiche Echtzeit bedeutet, dass im Allgemeinen die Bedingungen für Realzeit eingehalten werden. Sollten kleine Störungen auftreten, ist das zwar ärgerlich, führt aber zu keinen gravierenden Auswirkungen. Im Fehlerfall versucht man das Beste aus der Lage zu machen. Ein solches Verhalten wird als *Best Effort*-Verhalten bezeichnet. Sollte z. B. ein Routennavigationsprogramm einmal eine verspätete oder gar falsche Nachricht anzeigen, wird der Fahrer vielleicht den falschen Weg einschlagen. In der harten Realzeit werden dagegen systemkritische Fälle betrachtet, die im Schadensfall „katastrophale“ Auswirkungen zur Folge haben. Deshalb weicht das System dann auf ein definiertes *Fail Save*-Verhalten aus, das einen Mindestdienst garantiert. Große Automobilfirmen planen Leitsysteme, mit denen Fahrzeuge auf der Straße ganz alleine fahren können. Um eine möglichst vollständige Sicherheit zu garantieren, wird das Problem der harten Realzeit zugeordnet. Selbst bei Systemfehlern soll unter allen Umständen ein Unfall vermieden werden. Moderne Motorsteuerungen gewährleisten das, indem sie in den Fehlermodus umschalten und nur noch begrenzt Fahrzeugfunktionen zulassen. Eine automatische Geschwindigkeitsbegrenzung kann die Folge sein.

2.5 Systematisierung

Die Verwendung von *Modellen* zur Systematisierung von Schnittstellenproblemen ist ein gebräuchliches Hilfsmittel. Bereits in der Einleitung ist ein Modell (vgl. Abbildung 1.1) eingesetzt worden, um das Thema der Arbeit aufzuschlüsseln. In Abschnitt 2.2 wurde ein abstraktes Modell vorgestellt, um die Definition von Protokollen zu unterstützen. Die folgenden Modelle bauen als Beispiele komplexer *Schichtenmodelle* auf diesem abstrakten Modell auf. Bei ihrem Entwurf waren verschiedene Zielsetzungen ausschlaggebend, woher auch ihre Unterschiede resultieren. Die Auflistung in dieser Arbeit demonstriert den Stand der Technik für drei in der Praxis angewendete *Netzwerkprotokolle*.

2.5.1 ISO/OSI

Das ISO / OSI (International Standards Organization / Open Systems Interconnection) -Modell ist sehr komplex und wurde entwickelt, um eine internationale Standardisierung von Netzwerkprotokollen offener Systeme zu ermöglichen (vgl. Abbildung 2.39). Die Vollständigkeit dieses Modells ist aber auch seine Schwäche. Für jede der sieben Schichten bestehen Dienste und Protokolle. Die Kommunikation über die Schnittstellen erfolgt jeweils über die Service Access Points (SAP). Eine vollständige Implementierung des ISO / OSI-Modells ist sehr aufwendig und deshalb werden häufig nur die benötigten Teilkomponenten realisiert. Die wichtigste Rolle des ISO / OSI-Modells ist die Anwendung als Referenzmodell, mit dessen Hilfe andere Modelle bewertet werden können.

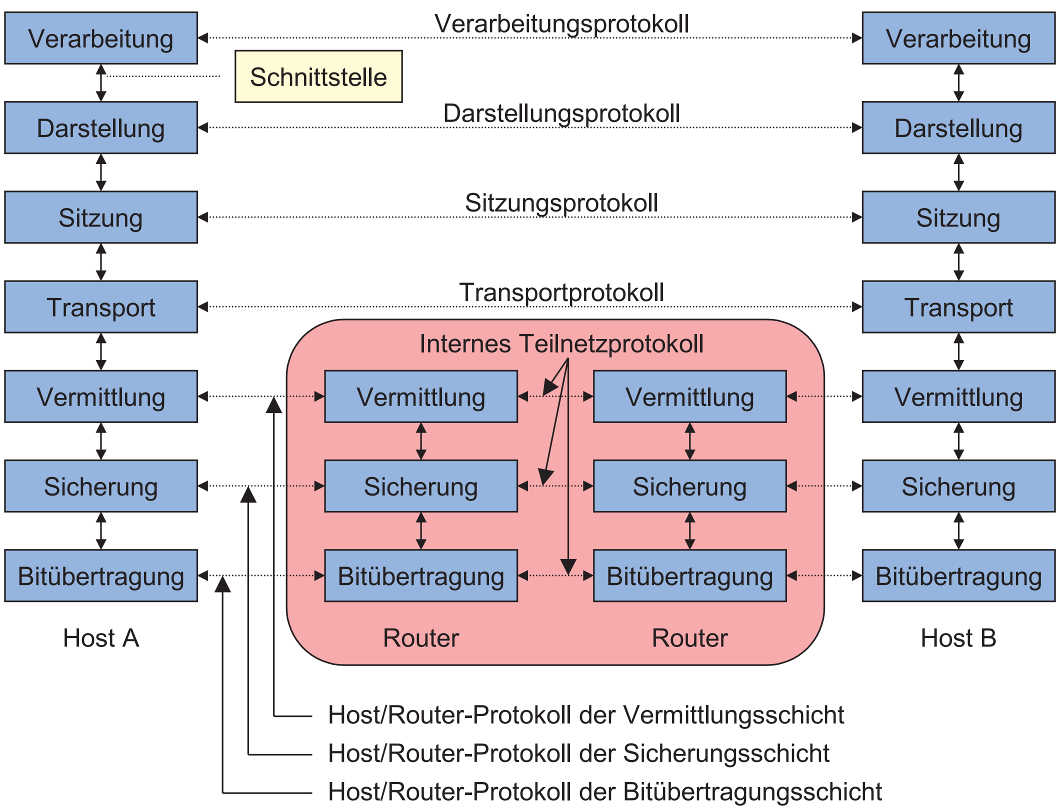


Abbildung 2.39: ISO / OSI-Modell

2.5.2 TCP/IP

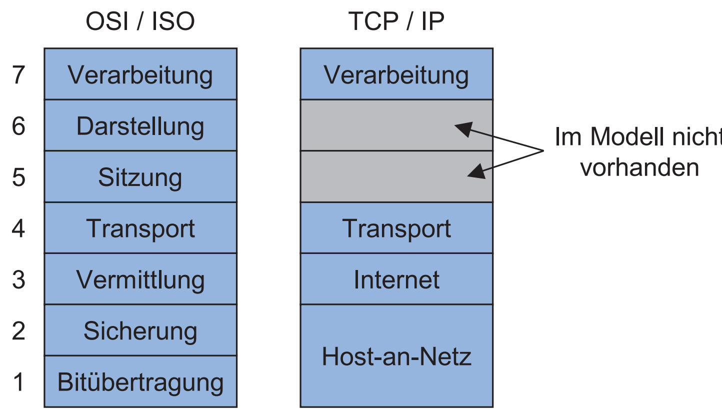


Abbildung 2.40: Vgl. ISO/OSI und TCP/IP

Einfachere Modelle wie das TCP / IP (Transmission Control Protocol / Internet Protocol) haben sich im Vergleich zu ISO / OSI am Markt durchgesetzt. Mit Hilfe des TCP / IP

erfolgt die Kommunikation im Internet, das im Laufe der letzten Jahre weltweite Verbreitung gefunden hat. Im Gegensatz zum ISO/OSI-Modell fehlen die Darstellungs- und die Transportschicht (vgl. Abbildung 2.40). Für jede Schicht sind verschiedene Protokolle definiert, die je nach Anwendung des TCP / IP eingesetzt werden.

2.5.3 ATM

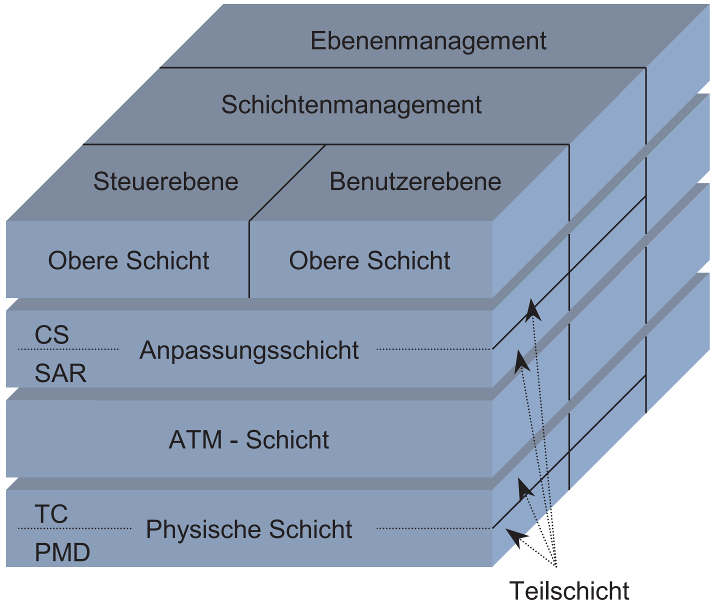


Abbildung 2.41: ATM Modell

ATM ist ein Echtzeitprotokoll für die Datenübertragung bei weicher Echtzeit. Mit dessen Hilfe können z. B. Videodaten über das Ethernet geschickt werden. Implementiert ist das im *RealPlayer* der Firma Netscape. Die Daten werden kontinuierlich versendet und bei einem Übertragungsfehler erfolgt keine Wiederholung. Das Verfahren wird als *Streaming* bezeichnet.

Der Aufbau von ATM ist in Abbildung 2.41 dargestellt. Um einen Vergleich zum ISO /-OSI-Modell zu ermöglichen, ist die Tabelle 2.5 eingefügt.

2.6 Bewertungskriterien

Bedingt durch verschiedene Rahmenbedingungen und Lösungsansätze haben sich im Laufe der Zeit die unterschiedlichsten Formen von Schnittstellen entwickelt. Es gibt Unterscheidungsmerkmale, anhand derer man Schnittstellen differenzieren und klassifizieren kann. Diese Unterscheidungsmerkmale sind die Parameter des Schnittstellenentwurfs. Man kann diese Parameter in die Kategorien Entwurf und ReUse aufteilen. Es

Tabelle 2.5: Vergleich von ATM mit ISO/OSI

OSI-Schicht	ATM-Schicht	Teil-schicht	Funktionalität
3 und 4	AAL	CS	Bereitstellung der Standardschnittstelle (Konvergenz)
		SAR	Segmentierung und erneute Zusammenstellung
2 und 3	ATM		Flusssteuerung
			Erzeugen/Extraktion des Zellen-Headers
			Management des virtuellen Pfades bzw. der Verbindung
			Multiplexen/Demultiplexen der Zellen
2		TC	Entkoppelung der Zellenrate
			Erzeugung und Verifikation der Header-Prüfsumme
			Erzeugen der Zellen
			Ein-/Auspacken der Zellen in/aus dem Umschlag
			Erzeugung von Rahmen
1	Physisch	PDM	Bitzeitgabe
			Physikalischer Netzzugriff

liegt auf der Hand, Konzepte zu definieren mit deren Hilfe große Teile dieser Schnittstellenklassen einfach, systematisch und einheitlich beschrieben werden können. In dieser Arbeit werden einige Lösungsansätze für die Unterklasse der Echtzeitschnittstellen betrachtet, die im Rahmen der Fallstudie vorkommen.

2.6.1 Kriterien für den Entwurf

Folgende Schnittstellenparameter haben sich als maßgeblich für den Entwurf herausgestellt. Eine Bewertung der Parameter ist abhängig von der Intention des Betrachters. In Bezug auf diese Arbeit kann man sich die Parameter folgendermaßen aufgeführt vorstellen:

- Implementierungsart (HW-, SW-Design oder HW/SW-Codesign [Har96])
- Geschwindigkeit und Determinismus
- Konfigurierbarkeit
- Leistungsverbrauch und Spannungspegel
- Wiederverwendbarkeit (ReUse)
- Entwurfs- und Fertigungskosten sowie deren Zeitbedarf (Time to Market)
- Platzbedarf, Größe
- Robustheit

2.6.2 Kriterien für den ReUse

Das Thema Wiederverwendung von Komponenten, im Englischen auch ReUse von IPs (Intellectual Properties) genannt, ist in der SW und der HW auf Gatterebene schon lange üblich. In allen großen Programmiersprachen gibt es Bibliotheken, aus denen vorgefertigte Codeelemente in den Quelltext eingebunden werden können. Auch in der Hardware entwickelt sich ein solcher Trend. Forschungsgruppen arbeiten daran Schnittstellenparameter und -Klassen zu definieren. Aktuelle Forschungsprojekte auf europäischer Europa sind Toolip und IPQ [HV01].
Kriterien für die Wiederverwendung sind:

- Effizienz und Effektivität der Komponente
- Aufwand der Adaption
- Wartbarkeit
- Dokumentation
- Verfügbarkeit

2.7 Zusammenfassung

In diesem Kapitel wurde der Stand der Technik bezüglich Schnittstellen und ihrer Parameter zusammengefasst. Die Schnittstellenparameter *Übertragungsmedien*, die *Protokolle*, das *Zeitverhalten* und der *Determinismus (Vorhersagbarkeit)* wurden sehr ausführlich analysiert. Dabei ist besonderer Wert auf eine strukturierte Darstellung der einzelnen Komponenten gelegt worden. Die Übertragungsmedien sowie deren Protokolle sind im Allgemeinen sowie an entsprechenden Beispielen erläutert worden. Die Betrachtung ist dabei immer unter dem Aspekt der Echtzeit vorgenommen worden.

Ein weiterer Teil des Kapitels befasst sich mit dem Aspekt des *Determinismus (Vorhersagbarkeit)* und der *Echtzeit* in Bezug auf den Schnittstellenentwurf. Dabei sind für sowohl für den Determinismus als auch für die Echtzeit Definitionen angegeben, die eingehend an Beispielen erläutert worden sind.

Eine Einordnung in bereits bestehende Modelle liefert das Teilkapitel *Systematisierung von Modellen.* Es werden drei komplexe Schichtenmodelle vorgestellt und miteinander verglichen.

Die Formulierung von *Bewertungskriterien für Schnittstellenparameter* von IF-Komponenten beschließt das Kapitel. Sie verdeutlichen, welche Aspekte einer Schnittstelle im Rahmen dieser Arbeit im Vordergrund stehen. Im Anschluss an dieses Kapitel wird die Modellierung der für das Entwurfskonzept wichtigen Komponenten beschrieben.

3 Modellierung von Echtzeitschnittstellen

Dieses Kapitel befasst sich mit der Modellierung von Schnittstellen. Anhand eines erarbeiteten *Modellierungskonzepts* wird ausführlich beschrieben, wie Schnittstellen modelliert werden können und welche Werkzeuge dazu geeignet sind. Danach wird im Kapitel 4 (Entwurf) das im Rahmen dieser Arbeit entwickelte *Entwurfskonzept* vorgestellt. Die Darstellung erfolgt mit Hilfe der hier vorgestellten Modellierungs-Werkzeuge.

Der allgemeine Prozess, um komplexe Probleme zu bearbeiten, lässt sich in zwei Vorgehensweisen unterteilen: *Top-Down* oder *Bottom-Up*. In dieser Arbeit wird der Top-Down-Entwurf von Schnittstellen favorisiert. Dabei beginnt man auf einer hohen Abstraktionsebene (Top) beim Gesamtproblem und gliedert es systematisch in Teilkomponenten. Um diese Komponenten eindeutig beschreiben zu können, ist eine Modellierung erforderlich. An dieser Stelle sollen Teilkonzepte von UML zum Einsatz kommen.

Ein bekannt gewordenes, fast schon historisches, Beispiel für die Modellierung von Schnittstellen kann man in Abbildung 3.1 sehen. Es handelt sich um die erste Skizze der Ethernetschnittstelle (vgl. [Sho85]). Dr. Robert M. Metcalfe zeichnete diese Skizze, um das Ethernet auf der National Computer Conference im Juni 1976 vorzustellen.

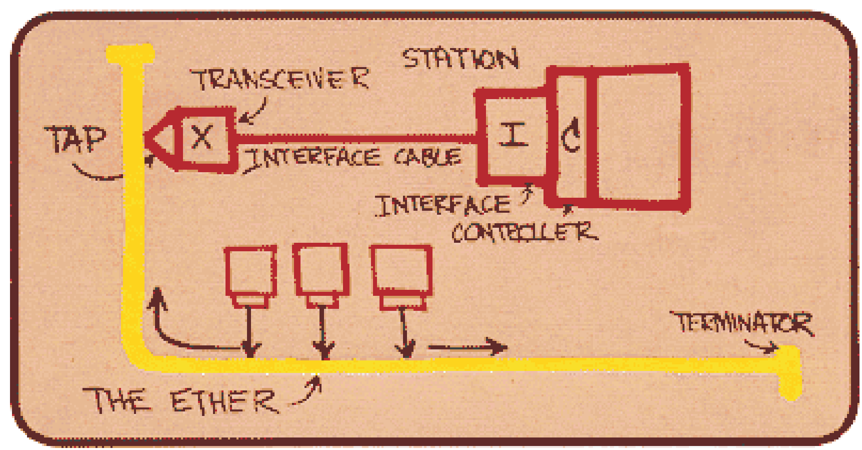


Abbildung 3.1: Original Modellierungsskizze der Ethernet Schnittstelle

3.1 Modellierungsansatz

Um die im Kapitel 2 beschriebenen Schnittstellenparameter und Merkmale einer Echtzeitschnittstelle erfassen und exakt beschreiben zu können, werden *formale Beschreibungstechniken* eingesetzt. Diese formalen Techniken beruhen auf formalen Modellen, die eine eindeutige Definition von Systemen erlauben. Außerdem sind formale Modelle standardisiert, so dass die Definition technikunabhängig ist. Das ermöglicht den Vergleich von unterschiedlichen Realisierungen einer Schnittstelle, und so können auch verschiedene Implementierungen auch auf Äquivalenz geprüft werden. Eine Anwendung finden die Methoden der formalen Beschreibung bei der Adaption und der Wiederverwendung von Komponenten. Außerdem müssen SW-Komponenten formal beschrieben werden, um patentfähig zu sein. Nicht zuletzt ist die formale Beschreibung Grundlage des automatisierten Entwurfs mit dem Ziel, notwendige Syntheseschritte mit Hilfe von Werkzeugen durchzuführen und so den Entwurf auf einer höheren, abstrakteren Ebene anzusetzen.

Einen weiteren Vorteil, den ein formales Modell mit sich bringt, ist die Möglichkeit der Verifikation und des gezielten Testens (Validierung). Dadurch besteht die Option ein Modell schon vor der eigentlichen Implementierung theoretisch auf Korrektheit zu prüfen. Häufig werden auch Simulatoren eingesetzt, mit denen man unter Vorgabe von Testwerten das Verhalten eines Modells auswerten kann.

3.2 Formale Methoden

In der Vergangenheit sind verschiedenen Konzepte zur Modellierung entstanden. Mit der Entwicklung der Technik haben sich auch die Modelle zu deren Beschreibung immer weiterentwickelt. Oft sind bestehende Konzepte erweitert worden oder durch eine Verschmelzung verschiedener Modellformen entstanden. Dadurch besteht eine gewisse Ähnlichkeit und / oder Redundanz in den Ausdrucksmöglichkeiten einiger formaler Methoden. Die formalen Methoden, mit dessen Hilfe eine Beschreibung einer Echtzeitschnittstelle möglich ist, lassen sich wie folgt gliedern [Tar91, Eng01, Dou00, Dou98]:

- Transitions-Modelle (Beschreibung der Dynamik)
  - Abstrakte Zustandsmaschinen
  - Endliche Automaten (FSM)
  - Prädikaten-Transitions-Netze (Pr/T-Netze)
  - Petri-Netze
  - Grammatiken

- Interaktionsdiagramme (Beschreibung der Funktionalität)
  - Message Sequence Charts (MSC)
  - Aktivitätendiagramme
  - Kollaborationsdiagramme
  - Specification and Description Language (SDL)
- Programmiersprachen (Beschreibung der Implementierung)
  - Java, VHDL, C, C++, Pascal, ...
- Algebren (Mathematische Modell-Beschreibung)
  - Prozess- und Typen-Algebra

In jedem Entwicklungsstadium von Modellierungssprachen können bestimmte Vertreter von formalen Methoden logisch zusammenfasst werden. Diese Einteilung ermöglichte es, zu jedem Zeitabschnitt das Spektrum an Ausdrucksmöglichkeiten, das vom Entwurf gefordert wurde, abdecken zu können. Besonders anschaulich sind die Darstellungsformen der Automaten- und Graphentheorie. SDL und Petri-Netze sind schon verhältnismäßig alt, genauso wie die Grammatiken und die Algebra-Beschreibung von Modellen, die bereits sehr früh in die Informatik eingingen. Diese Modelle haben den Vorteil, bereits umfassend erforscht zu sein und bieten meist viele spezielle Varianten und Ausprägungen.

Aktueller dagegen ist die Modellierung mit der *Unified Modelling Language (UML)*. Dabei handelt es sich um eine Kollektion moderner Konzepte und Methoden zur Beschreibung von objektorientierten Modellen. Die einzelnen Diagrammsprachen aus UML lassen sich problemlos in den oberen Kontext einordnen [Eng01, Oes01]. Die eingesetzten Diagramme von UML werden, mit einer Begründung der Auswahlkriterien und Einsatzmöglichkeiten, im Folgenden näher erläutert.

3.3 Modellierung von Schnittstellen mit UML

In dieser Arbeit erfolgt die Modellierung der Echtzeitschnittstellen durch die *Unified Modelling Language (UML)*, um auf einem möglichst aktuellen Stand der Forschung zu sein. Es gibt bereits Literatur [Dou00, Dou98, Gom00, SGW94] die darlegt, dass die Modellierung von Beispielen der harten Realzeit mit UML möglich ist. Inhalt dieser Bücher ist die Modellierung von Applikationen und Echtzeit-Systemen. Darauf aufbauend wird an dieser Stelle die Modellierung von Schnittstellen, konform zu den bereits bestehenden Ansätzen, beschrieben.

Ein weiterer Vorteil von UML ist, dass bereits viele Ansätze für *Codegenerierung* bestehen. So ergibt sich die Möglichkeit, aufbauend auf einer wohldefinierten Modellierung, eine Codegenerierung, wie sie für die Programmiersprache Java bereits besteht, für VHDL

oder Verilog aufzusetzen. Das stellt einen wichtigen Baustein in der Entwicklung eines *automatisierten Entwurfs* dar.

Die Auswahl der UML-Diagramme beschränkt sich hier auf folgende Diagrammtypen, was jedoch ausreicht, um das notwendige Spektrum an Ausdrucksmächtigkeit bereitzustellen:

- UseCase Diagramm
- Klassendiagramm (mit Paketdiagramm)
- Statecharts
- Sequenzdiagramm
- Aktivitätendiagramm

Im Kontext der formalen Modelle ergibt sich, dass die Transitions-Modelle durch Statecharts und die Interaktionsdiagramme durch Aktivitäts- und Sequenzdiagramme dargestellt werden. Hinzu kommen die in UML definierten UseCase Diagramme, mit denen die Anforderungsanalyse und Funktionalitätsprüfung sehr einfach erfolgen kann. Weiterhin erleichtern UseCase Diagramme durch ihren Aufbau die Top-Down Strukturierung von Problemfällen. Literatur zum Thema Modellierung mit UML findet man in [Eng01, Dou00, Dou98, Gom00, Oes01, JBR99]. Als Anmerkung möchte ich hinzufügen, dass die UML Diagramme zu großen Teilen mit dem Tool *Rational Rose Enterprise Edition* der Firma Rational erstellt wurden. Die Ausdrucksmöglichkeiten dieses Tool für Statecharts, Sequenzdiagramme und Aktivitätendiagramme sind stark begrenzt, wie das durchaus auch bei anderen Werkzeugen der Fall ist. Die hier erarbeiteten, sehr anspruchsvollen Modelle werden gemäß dem Rational-Rose „Design-Style“ vorgestellt.

3.3.1 UseCase Diagramm

UseCase Diagramme werden verwendet, um eine *Anforderungsanalyse*, d. h. eine Identifikation der Systemoperationen, durchzuführen und Benutzerschnittstellen festzulegen. Ein UseCase, ein so genannter „Nutzfall“ oder „Anwendungsfall“, beschreibt eine eigenständige Operation, die ein entsprechender Nutzer in Anspruch nehmen kann. Im Allgemeinen werden UseCases für die Schnittstelle Mensch – Maschine eingesetzt. Dazu äquivalent ist ein Szenario, in dem eine Task auf entsprechend vorgegebene Dienste zugreift (vgl. Abbildung 3.2).

Die wichtigen Elemente des UseCase Diagramms sind zum einen die Aktoren (engl.: Actor) und zum anderen die Anwendungsfälle (UseCases). Die entsprechenden Beziehungen zwischen den Elementen werden durch Assoziationen (eng.: Associations) dargestellt. Abhängigkeiten innerhalb des UseCase Diagramms können durch die *Includes*- und *Extends*-Kanten modelliert werden. Auch *Generalization*, d. h. die Vererbung innerhalb der UseCases, ist hier möglich.

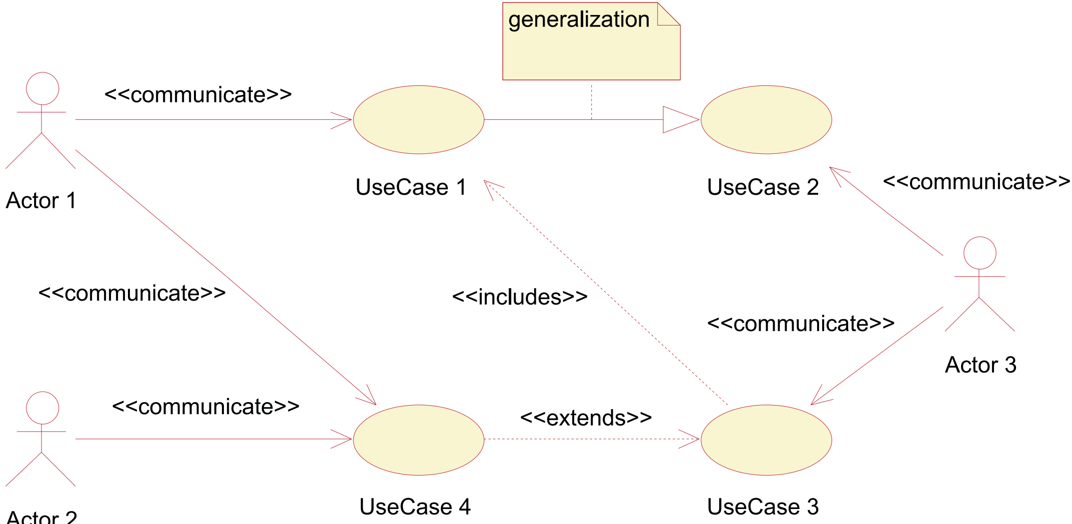


Abbildung 3.2: UML: UseCase Diagramm

3.3.2 Klassendiagramm (Class Diagramms)

Das Klassendiagramm zählt zu den *Strukturdiagrammen* und legt das „Wer ist enthalten“ im Objektmodell fest. Damit können die Struktur der Daten und alle statischen Abhängigkeiten modelliert werden. Das bedeutet, dass dadurch sowohl die Anzahl von vorhandenen Komponenten, als auch deren Abhängigkeiten festgelegt werden (vgl. Abbildung 3.3). Ein UML Klassendiagramm besteht aus der Definition von Klassen und Beziehungstypen. Eine Klasse wiederum besteht aus Attributen (Struktur) und Operationen (Verhalten) gleichartiger Objekte. Die Beziehungstypen definieren die erlaubten Beziehungen zwischen diesen Objekten.

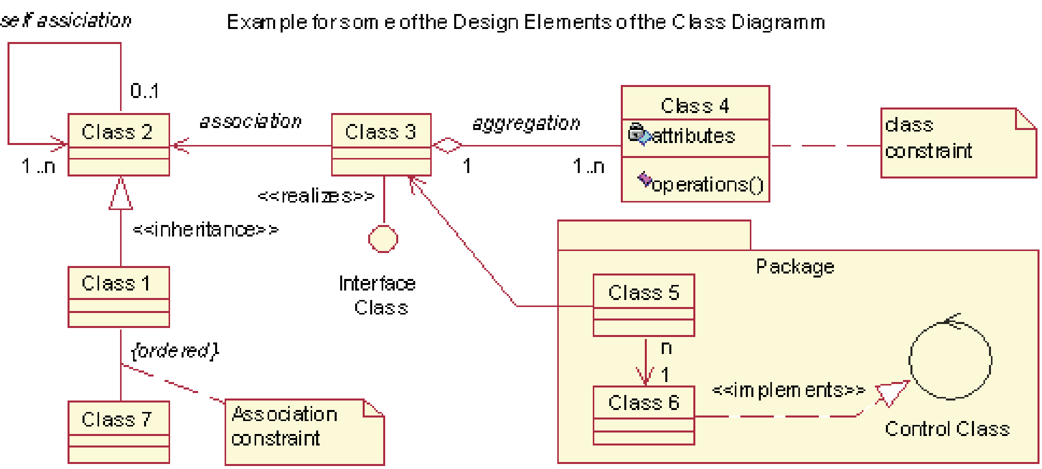


Abbildung 3.3: UML: Klassendiagramm

Weitere Elemente des Klassendiagramms sind die sogenannten *Stereotypen*. Sie bezeichnen ganz speziell definierte Typen von Klassen, wie z. B. die Klasse *Interface*, dargestellt durch einen kleinen Kreis, oder die Klasse Control, beschrieben durch einen großen Kreis mit einer Pfeilspitze im oberen Bereich der Grafik (vgl. Abbildung 3.3). Die Realisierung dieser Stereotypen wird durch eine gestrichelte Verbindungslinie mit geschlossener Pfeilspitze angedeutet. Verbindungslinien im Klassendiagramm werden als Assoziationen bezeichnet und können durch *Aggregation* (weiß gefüllte Raute) oder *Komposition* (schwarz gefüllte Raute) erweitert werden (zwischen *Class 3* und *Class 4*). Zusätzlich können an den Assoziationen noch Richtung (engl.: Direction), Anzahl (engl.: Multiplicity), Rolle (engl.: Roles), Leserichtung (engl: Navigability) und Ordnung (engl.: Ordered) angegeben werden. Auch wird hier die Vererbung, in UML als *Generalization* bezeichnet, durch eine geschlossene weiß gefärbte Pfeilspitze modelliert (zwischen *Class 1* und *Class 2*). Des weiteren können *Notes* und *Constraints* angegeben werden.

Eine besondere Ausprägung des Klassendiagramms ist das Paketdiagramm (engl.: Packet Diagramm). Das Paket selbst ist keine Klasse und dient ausschließlich der Strukturierung und der Beschreibung von Hierarchie in einem Klassendiagramm. In einem Paket können sich sowohl Klassen als auch weitere Pakete befinden. Falls nur Pakete enthalten sind, spricht man von einem Paketdiagramm. Meistens werden aber Pakete im Zusammenwirken mit Klassen eingesetzt, dann bezeichnet man das Diagramm wiederum als Klassendiagramm.

3.3.3 Zustandsdiagramme (Statecharts)

Zustandsdiagramme sind als Graphen dargestellte endliche Automaten. Sie zählen zu den Verhaltensdiagrammen und beschreiben die Dynamik, das „Wann ändert sich etwas" eines Systems. Zustandsdiagramme sind ähnlich wie Automaten als n-Tupel definiert und können bei realistischen Beispielen schnell eine unübersichtliche Größenordnung annehmen. Statecharts sind erweiterte Zustandsdiagramme, die zusätzliche Ausdrucksmöglichkeiten bieten. Die Zustandsübergänge, die durch Ereignisse (engl.: Events) ausgelöst werden, können mit Bedingungen (engl.: Guards) versehen werden und während des Zustandsübergangs Aktionen (engl.: Actions) oder Ereignisse auslösen (vgl. Abbildung 3.4).

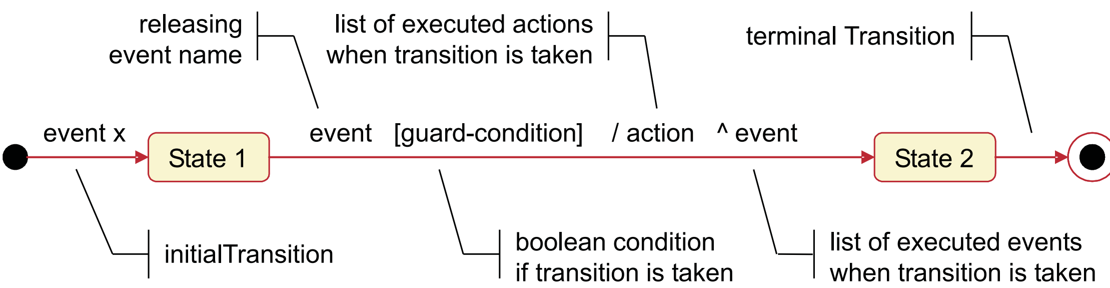


Abbildung 3.4: UML: Statechart – Die Zustandsübergangsfunktion

Zusätzlich Ausdrucksmöglichkeiten des Statecharts sind Hierarchie (engl.: Hierarchy), Nebenläufigkeit (engl.: Concurrency), Bedingungen (engl.: Conditions) und der History-Mechanismus (engl.: History-Mechanism). Das in Abbildung 3.5 dargestellte Statechart beschreibt den Sendeprozess eines FireWire-Zyklus. Der Automat beginnt mit dem *Cycle-Start* und geht dann zum Zustand *Transmit* über, der hierarchisch weitere Zustände in Bezug auf den FireWireSend-Zustand kapselt. Der darin enthaltenen Zustand *Do Send* beschreibt in *Transmit* eine weitere Hierarchiestufe. Im Weiteren Verlauf teilt sich der Automat nach dem Zustand *Init* in zwei nebenläufige Prozesse, die zum Zustand *Medium Access* wieder zusammengeführt werden. Nachdem alle erforderlichen Daten in *DO Send* versendet wurden, wird der Zustand *Transmit* verlassen und über den Verzweigungsmechanismus (engl.: Condition-Mechanism) der Nachfolger-Zustand bestimmt. Zuerst folgt dabei der isochrone Sendemodus und, falls dessen vorgegebene Zeit abgelaufen ist, der asynchrone Modus. Wenn auch die dafür vorgesehene Zeit verstrichen ist, wird der Sende-Zyklus durch das Verlassen des FireWireSend-Zustands beendet.

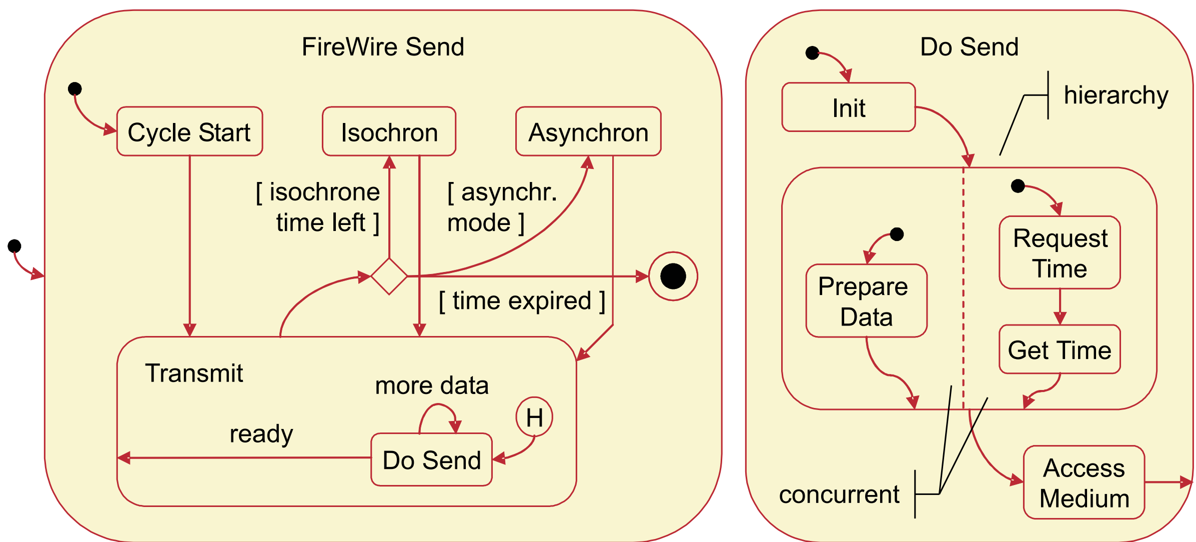


Abbildung 3.5: UML: Statechart – Hierarchie und Nebenläufigkeit

Als weiteren Vorteil bieten Statecharts die in einem Zustand angeordnete Entry- / Do- / Exit- / Event- Funktionalität (vgl. Abbildung 3.6). Damit können beim Betreten, während des Verweilens, beim Verlassen oder zu bestimmten Ereignissen sowohl Aktionen als auch Ereignisse ausgelöst werden.

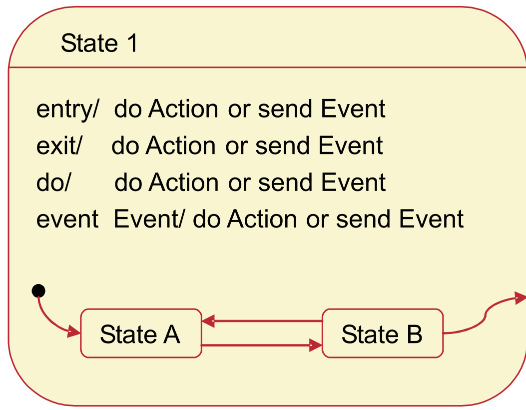


Abbildung 3.6: UML: Statechart – Aktionen und Operationen

Mit Hilfe der Statecharts kann das dynamische Verhalten einer Schnittstelle dargestellt werden. Komplexe Sequenz- und Protokollgeneratoren sowie deren Handshakes, kann man so definieren. Zeitliche Abhängigkeiten dagegen sind nur sehr eingeschränkt durch Statecharts modellierbar. Dazu eignen sich die folgenden Diagramme, die das funktionale Modell eines Systems beschreiben.

3.3.4 Sequenzdiagramme (Sequencecharts)

Sowohl Sequenz- als auch die Aktivitätendiagramme beschreiben die Funktionalität eines Systems, dass sogenannte „Was verändert sich“. Ein Sequenzdiagramm ist szenarioorientiert, was bedeutet, dass man das Verhalten eines Systems an repräsentativen Fällen definiert. Historisch gehen die Sequenzdiagramme auf die Message Sequence Charts (MSC) und die Event Trace Diagramme zurück. In den Sequenzen wird die Interaktion zwischen konkreten Objekten durch den Austausch von Nachrichten über die Zeit beschrieben (vgl. Abbildung 3.7). Es kann sowohl synchroner als auch asynchroner Verkehr modelliert werden und anhand der Lebenslinien kann man exakt festlegen, wann Objekte aktiv bzw. inaktiv sein sollen.

Das in Abbildung 3.7 beschriebene Szenario bezieht sich auf den in Abbildung 3.5 beschriebenen Sendezyklus von FireWire und zeigt den Ausschnitt eines möglichen Ablaufs. Am Anfang (Zeitpunkt a) aktiviert das Ereignis einer externen Task die FireWire-Klasse Send. Diese ruft dann später, zum Zeitpunkt b über den Verzweigungsmechanismus alternativ den isochronen oder den asynchronen Sendemodus auf. Dabei ist hier das zwischenzeitliche Senden des Cycle-Starts vernachlässigt worden. Aus dem angewählten Sendemodus wird dann die Transmit-Klasse aktiviert, die bis zu ihrem Verlassen interne

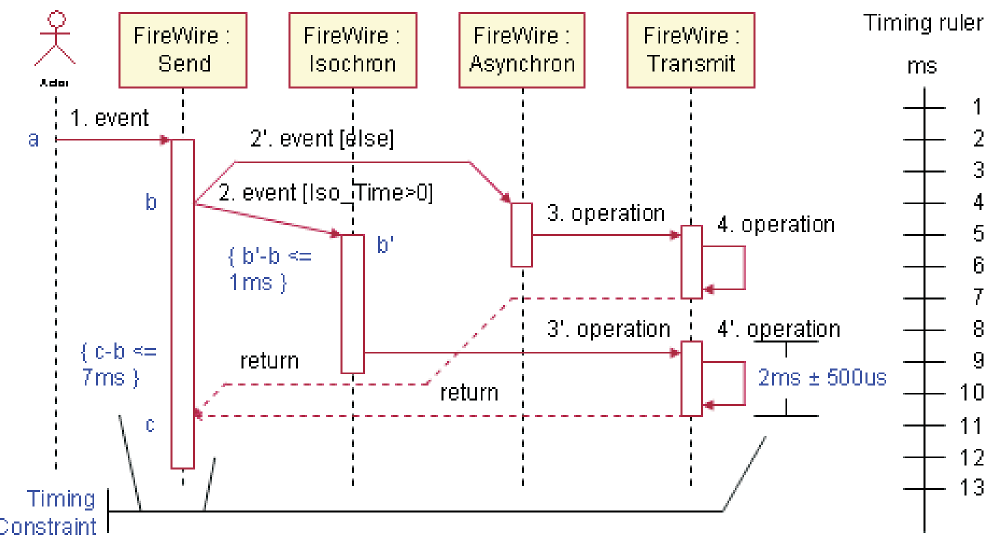


Abbildung 3.7: UML: Sequence Chart

Operationen ausführt. Die Zeitvorgaben (in Abbildung 3.7 blau dargestellt) werden hier auf zwei alternative Methoden modelliert: Als Differenz zweier Aufrufzeitpunkte lassen sich sehr gut Laufzeiten und komplexe Zeitintervalle beschreiben (links im Bild). Mit der rechts im Bild aufgezeigten Notation kann man adäquat Berechnungszeiten und Zeitspannen auf einer Lebenslinie darstellen. Um eine Referenz zur Zeitbasis zu haben, kann man alternativ ein Zeitlineal in die Grafik einfügen.

Sequenzdiagramme eignen sich besonders bei komplexen, tief geschachtelten Aufruffolgen, die über zeitliche Abhängigkeiten gekoppelt sind. Dieser Fall liegt bei Schnittstellen z. B. im Fall von hierarchischen Schichtenmodellen und ineinander geschachtelten Dienst-Aufrufen vor. Auch Protokoll- und Task-Zeitanforderungen können so modelliert werden.

3.3.5 Aktivitätendiagramme (Activity Diagramms)

Aktivitätendiagramme werden, ähnlich wie die Sequenzdiagramme, verwendet, um die zeitlichen Abhängigkeiten eines Systems zu definieren (vgl. Abbildung 3.8). Der Unterschied besteht darin, dass ein Aktivitätendiagramm datenflussorientiert ist. D. h. es besteht eine starke syntaktische Verwandschaft zu Statecharts (vgl. Abbildung 3.5), allerdings mit einer anderen semantischen Interpretation. Ein Zustand wird dadurch charakterisiert, dass eine Aktivität ausgeführt wird. Die Zustandsübergänge werden durch das Ende einer Aktivität ausgelöst.

Das in Abbildung 3.8 dargestellte Aktivitätendiagramm bezieht sich ebenso wie das eben vorgestellte Sequenzdiagramm, auf den im Statechart (vgl. Abbildung 3.5) vorgestellten

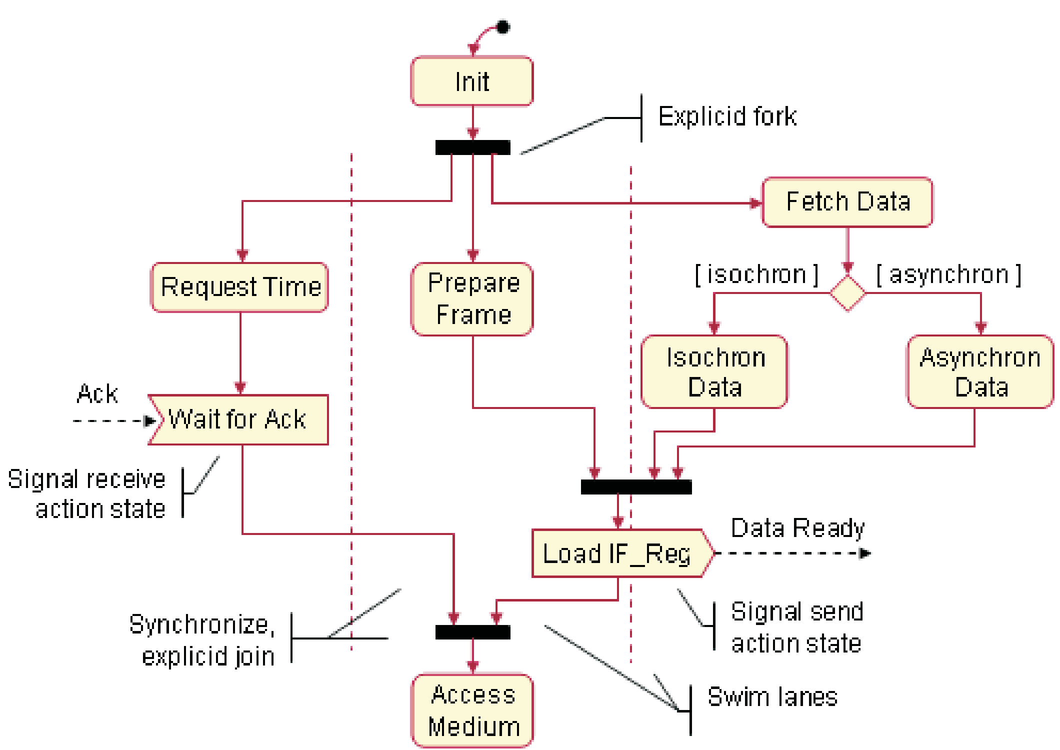


Abbildung 3.8: UML: Activity Diagramm

Sendezyklus von FireWire. Allerdings wird hier ein anderer Teil des Modells betrachtet, der sich mit einem Aktivitätendiagramm besser beschreiben lässt als mit einem Sequenzdiagramm. Die *Do Send* Klasse ist datenflussorientiert und beinhaltet nebenläufige Operationen. So werden nach der Initialisierung durch die Aktivität Init drei parallele Aktivitäten angestoßen, die durch die so genannten *Swim Lanes* getrennt werden. In der linken Spalte erfolgt die Zeitanforderung, die dann auf die Eingabe *Acknowledge* wartet. In der Mitte wird die Generierung des zu versendenden Rahmens vorbereitet und rechts im Bild erfolgt das Bereitstellen der zu versenden Daten. Dabei kann zwischen isochronen und asynchronen Daten unterschieden werden. Ist dieser Vorgang zusammen mit dem Vorbereiten des Versende-Rahmens abgeschlossen, wird ein Data-Ready-Signal als Ausgabe versendet. Zusammen mit der Sendefreigabe durch das Acknowledge kann dann der Buszugriff (Bus Access) erfolgen.

Das Aktivitätendiagramm eignet sich besonders gut, um eine übersichtliche Ablaufstruktur in datenflussorientierten Modellen darstellen zu können und die komplexen Abhängigkeiten zwischen den Knoten erkennbar zu machen. Auf diese Weise können Elemente einer Schnittstelle, wie z. B. das Handshake, modelliert werden.

3.4 Modellierungskonzept

Bisherige Konzepte zur Beschreibung von Schnittstellen waren jeweils auf wenige Blickrichtungen der *Modellierungssichten* beschränkt. In dem Zusammenhang gibt es drei verbreitete Konzepte (vgl. Tabelle 3.1)

Tabelle 3.1: Modelle zur Beschreibung von Schnittstellen [HT01]

Name	Bezeichnung	Anmerkung
SLIF	the System-Level Interface	Standard zur Beschreibung des Verhaltens
OCB	the On-Chip Bus Virtual Component Interface	Beschreibung mittels Busstandards, Schnittstellen als virtuelle „Wrapper-IF“
SLDI	the System-Level Data-Type Initiative	Definition eine Menge von Konfigurations- und Operationstypen

Wünschenswert wäre eine Beschreibung mit der dazu gehörigen Modellierung, mit deren Hilfe eine ausreichenden Definition aller wichtigen Elemente einer Echtzeitschnittstelle möglich ist. Um eine umfassende Sicht auf eine solche Schnittstelle zu erhalten, beinhaltet das *Modellierungskonzept* in dieser Arbeit folgende Komponenten:

- Task
  - HW-Block
  - SW-Block
- Kanal
- Protokoll

Weitere Informationen zu den einzelnen Komponenten finden sich in [HP94]. Das Protokoll ist zu großen Teilen vom verwendeten Kanal abhängig, deshalb werden von nun an der Kanal und das Protokoll als eine Einheit betrachtet. Aus dieser Einteilung ergeben sich zwei Hauptgruppen: Die *Task* auf der einen und der *Channel* auf der anderen Seite der Schnittstelle (vgl. Abbildung 3.9).

Jede dieser Hauptgruppen wird mittels UML durch die entsprechend notwendigen Diagramme modelliert. Das im nächsten Kapitel eingeführte *Entwurfskonzept*, welches durch den *Interface-Block (IFB)* realisiert ist, befindet sich als Bindeglied zwischen Task und Channel. Durch den IFB ist die *Verbindung zwischen verschiedenen Tasks und Channels, denen eine Menge von, für diesen Kanal spezifizierten, Protokollen zugeordnet wird, erst möglich* [SGW94].

Die Modellierung für die *Adaption* von Komponenten ist in ganz ähnlicher Weise denkbar, wenn man den Channel durch eine bereits existierende Task *Existing_ Task* ersetzt (vgl. Abbildung 3.10). Die Schnittstelle *IF_ Channel* wird dann zu *IF_ ExistingTask* und die zu integrierende Task wird noch in TaskToAdapt umbenannt.

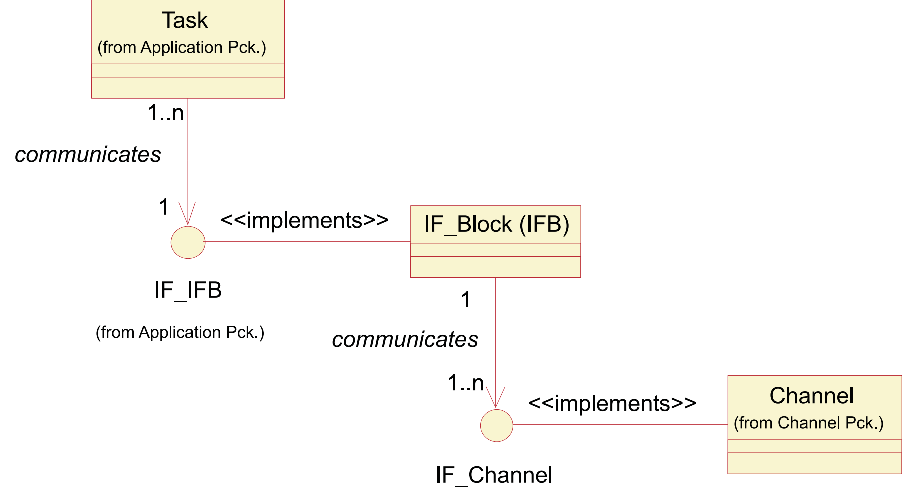


Abbildung 3.9: Modellierungskonzept für die Schnittstelle Task - Kanal

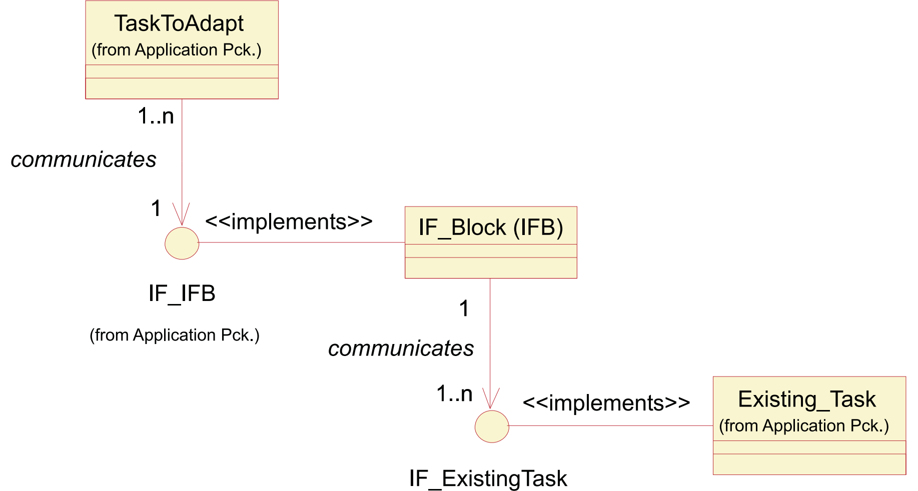


Abbildung 3.10: Modellierungskonzept für die Adaption einer Komponente

Prinzipiell handelt es sich dabei um den gleichen Vorgang, allerdings unterscheiden sich die Anforderungen einer Task etwas von denen eines Kanals. Dabei sind die Gemeinsamkeiten jedoch so tiefgreifend, dass man den gleichen IFB in einer etwas modifizierten Form einsetzen kann.

3.4.1 Anforderungsanalyse

Wie bereits beschrieben, erfolgt die Anforderungsanalyse dieses Modellierungskonzepts mit Hilfe von UseCases. Abbildung 3.11 zeigt die allgemeinen Anwendungsfälle aus Sichtweise einer Task, bzw. von n Tasks. Die wichtigsten Tätigkeiten sind das Versenden

(Send) und das Empfangen (engl.: Receive) von Daten. Zusätzlich schreibt jede Task eine benötigte Bandbreite vor (engl.: demand Bandwidth), die für die jeweilige Tätigkeit der Task erforderlich ist.

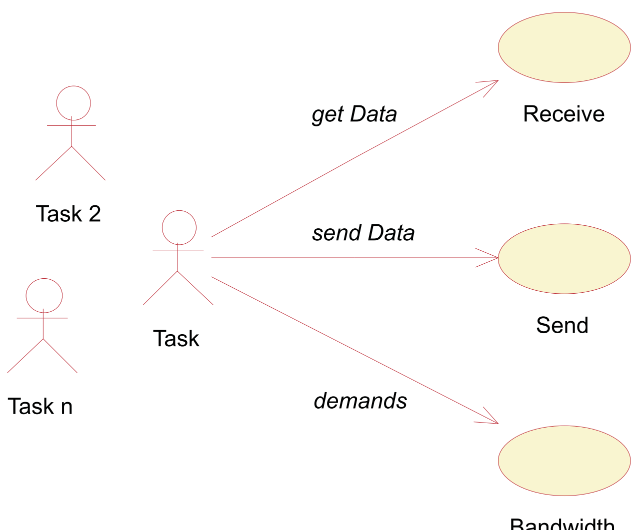


Abbildung 3.11: UseCase: Sichtweise der Task

Ein Echtzeitkanal (engl.: Real-Time Channel) lässt sich dadurch charakterisieren, dass er eine dedizierte Bandbreite zur Verfügung stellt und eine Menge von Protokollen unterstützt. Abbildung 3.12 verdeutlicht grafisch die Sichtweise für 1 bis m Channels. Alle aktiven Elemente, wie hier der Kanal oder in Abbildung 3.11 die Task, werden als Strichmännchen dargestellt. Das ist das allgemein übliche Symbol der Gruppe *Actor*. Die in Anspruch genommenen Aufgaben werden durch gelb gefüllte Ellipsen charakterisiert. Weiterhin ist eine Assoziation zwischen dem Protokoll und der Bandbreite mit der Bezeichnung *muss den Anforderungen entsprechen* (engl.: must fit) eingefügt. Damit wird angegeben, dass nur solche Protokolle eingesetzt werden können, die auch der zur Verfügung gestellten Bandbreite genügen.

In Abbildung 3.13 sind dann die in Abbildung 3.11 und 3.12 vorgestellten Sichten zusammengefasst und zusätzlich mit der Anzahl der beteiligten Komponenten markiert. Hier sind auch die Berührungspunkte der Task und des Kanals ersichtlich. Das Senden und Empfangen der Task stützt sich auf das Protokoll, welches vom Kanal angeboten wird. Dabei muss der Kanal der von der Task geforderten Bandbreite genügen.

Aus diesem Szenario lassen sich bereits durch eine Kapselung der Berührungspunkte die jeweiligen Schnittstellen extrahieren, die von der Task und dem Kanal vorgegeben werden. In Abbildung 3.14 ist diese Zweiteilung ersichtlich: Es gibt zum einen die Kanal-Schnittstelle (*Channel Interface*) und zum anderen die Task-Schnittstelle (*Task Interface*).

Diese Einteilung führt direkt zu der Frage, was die Anforderungen dieser beiden Schnittstellen sind und wie sie sich beschreiben, oder besser gesagt, modellieren lassen. Die

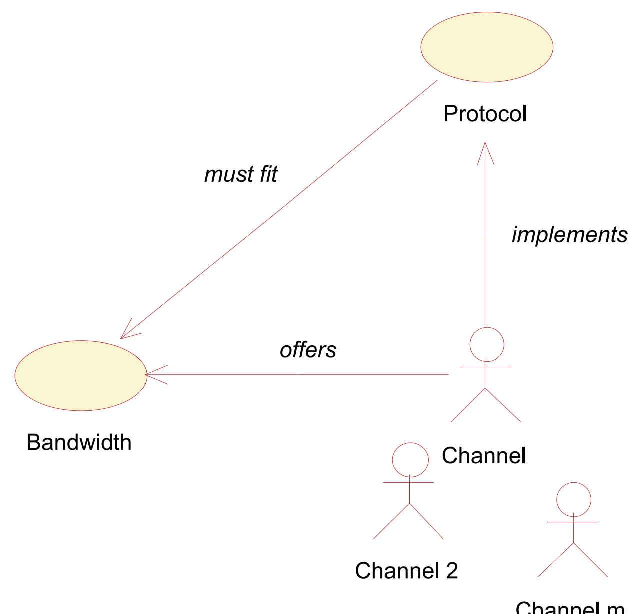


Abbildung 3.12: UseCase: Sichtweise des Channels

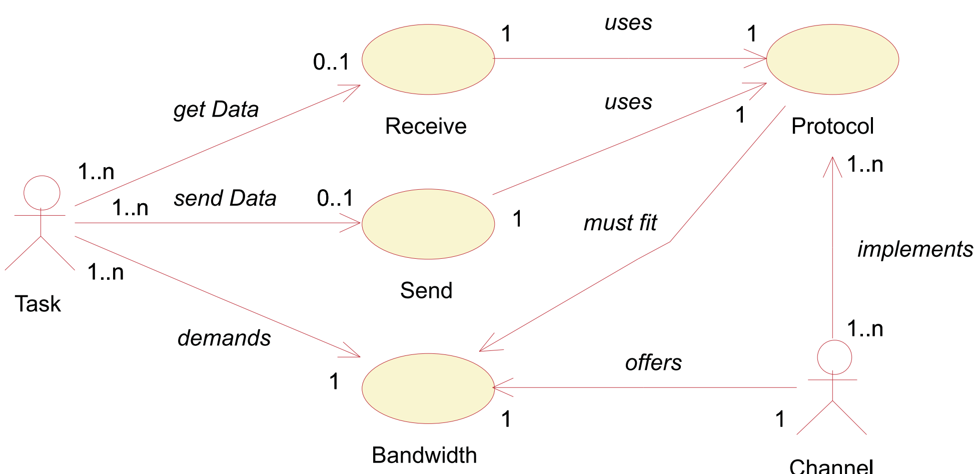


Abbildung 3.13: UseCase: Gesamtszenario

beiden nachfolgenden Abschnitte geben eine Antwort auf genau diese Frage.

3.4.2 Taskanforderungen

Eine wesentliche Erkenntnis ist, dass bei der Betrachtung der Schnittstellen nicht der gesamte Aufbau der Task eine Rolle spielt. Das bedeutet, die Aufgabe besteht darin, eine HW- oder SW-Task allgemein auszudrücken und zu analysieren, um dann die für die Schnittstelle bedeutenden Parameter zu extrahieren und zu modellieren.

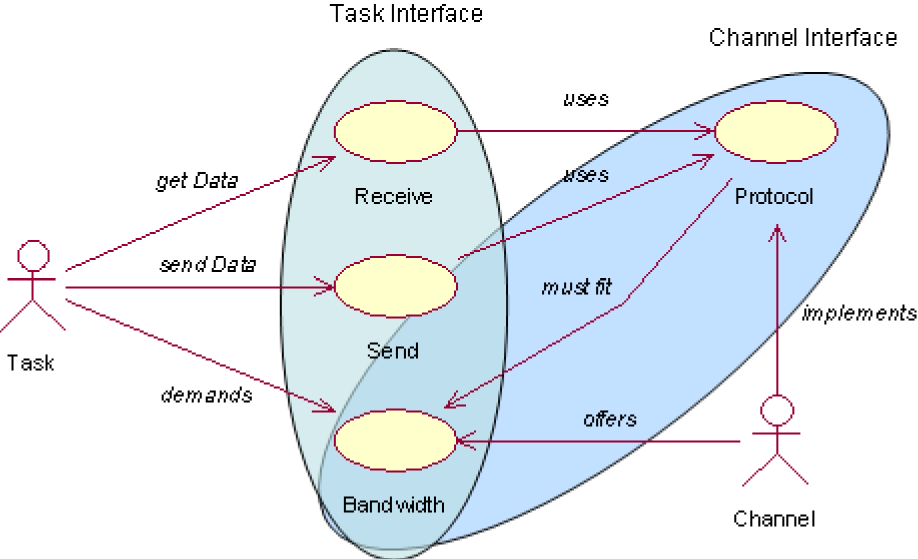


Abbildung 3.14: UseCase: Einteilung in Task- und Channel-Interface

Die Analyse liefert das Ergebnis, dass eine Task diesbezüglich durch die Daten an der Schnittstelle in Zusammenhang mit deren Zeitverhalten charakterisiert werden kann. Diese grundlegende Struktur findet sich auch im folgenden Modell wieder.

Um die Task zu modellieren, ist ein entsprechendes Klassendiagramm ausgearbeitet worden, das alle wesentlichen Bestandteile, die zu den Anforderungen der Task beitragen, beschreibt (vgl. Abbildung 3.15). Im oberen Bereich der Abbildung ist die Task als zentrale Klasse abgebildet. Diese besteht aus (engl.: consists of) den zugehörigen Daten (engl.: Data) und den zugeordneten (engl.: has applied) Zeitrestriktionen (engl.: Time Restrictions). Dabei müssen die Zeitrestriktionen den Daten genügen (engl.: fits to). Wie schon erwähnt, interessieren allerdings nicht alle Daten und Zeitrestriktionen einer Task, sondern ausschließlich die für die Schnittstelle relevanten Werte. Die Zeitrestriktionen werden daher ausschließlich durch die *Schnittstellen-Ankunftszeit* (engl.: Interface Arrival-Time) bestimmt (engl.: pretend). Damit sind alle Zeitpunkte gemeint, zu denen Daten an der Schnittstelle anliegen müssen. Die Schnittstellen-Ankunftszeit wiederum setzt sich aus drei Komponenten zusammen: Den Deadlines der Task, den Berechnungszeiten (engl.: Computation Time) und den in der Task auftretenden Laufzeiten (engl.: Latency).

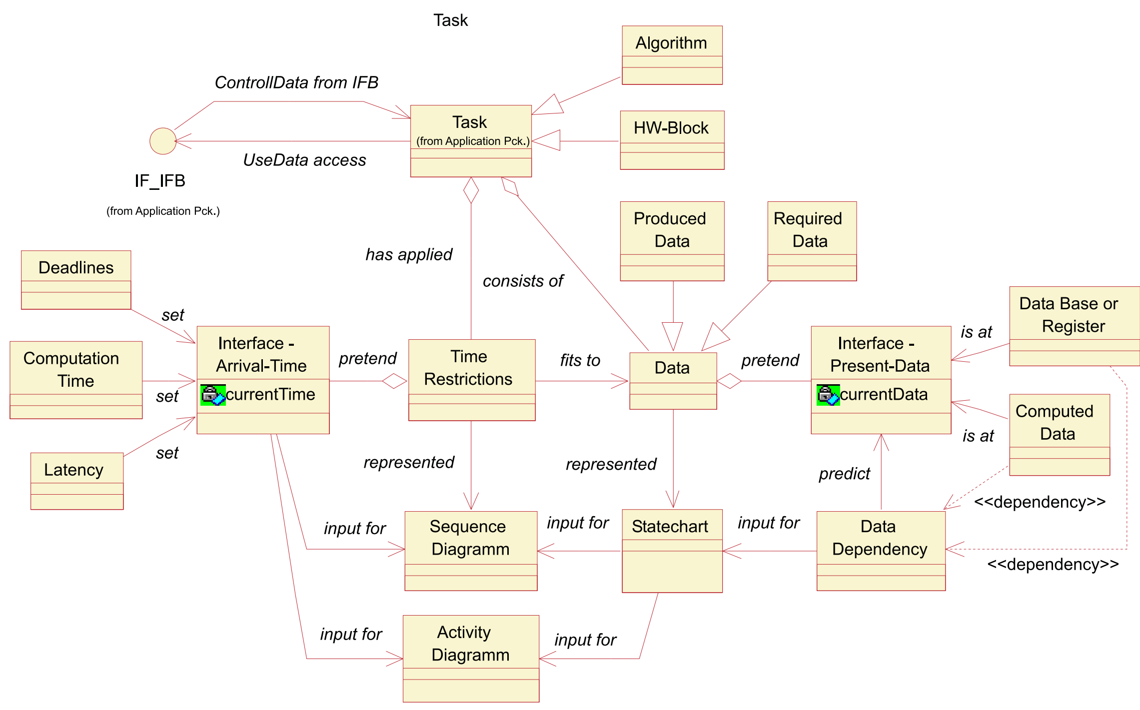


Abbildung 3.15: Modellierung einer Task

Die Daten werden entsprechend den Zeiten durch die *an der Schnittstelle anliegenden Daten* (engl.: Interface Present-Data) festgelegt (engl.: pretend). Für diese Daten kommen zwei Typen in Frage: Statische Daten, wie z. B. aus einer Datenbank oder einem Register (engl.: Data Base or Register) oder dynamisch berechnete Daten (engl.: Computed Data). Für diese Daten bestehen Abhängigkeiten (engl.: Data Dependency), die genau vorhersagen (engl.: predict), welche Daten an der Schnittstelle verfügbar sein müssen. Diese Datenabhängigkeit ist ebenfalls Eingabe (engl.: input) für ein Statechart, mit dessen Hilfe die Daten dargestellt (engl.: represented) werden können. Unter Zuhilfenahme der Interface Arrival-Time und des Statecharts lassen sich schließlich Sequenzdiagramme generieren, mit deren Hilfe dann die Zeitrestriktionen ausgedrückt (engl.: represented) werden.

Zur Kommunikation verwendet die Task die Schnittstelle des Interface-Blocks (*IF_IFB*). Kontrolldaten (engl.: ControllData from IFB) können auf diese Weise vom Interface-Block an die Task weitergeleitet werden. Der Nutzdaten-Verkehr erfolgt ebenfalls über den Zugriff auf den IF_IFB (UseData access).

Im Diagramm 3.15 wird die Realisierung von Klassen durch Vererbung dargestellt. Eine Ausprägung der eben definierten Task kann ein Algorithmus (engl.: Algorithm) sein, eine andere ein Hardware-Block (HW-Block). So gibt es auch zwei verschiedene Typen von Daten: Die produzierten Daten (engl.: Produced Data) und die konsumierten Daten (engl.: Required Data).

Das soeben beschriebene Modell wird von nun zur Repräsentation von Tasks genutzt.

Der Zusammenhang zum Interface-Block wird in Kapitel 4 hergestellt. Das gleiche gilt für die Kanalanforderungen, die im Folgenden modelliert sind.

3.4.3 Kanalanforderungen

Die Gegebenheiten für die Modellierung des Echtzeitkanals sind ähnlich zu denen der Task. Die Analyse des Kanals ergibt, dass eine primäre Abhängigkeit zum eingesetzten Medium und dem Protokoll besteht.

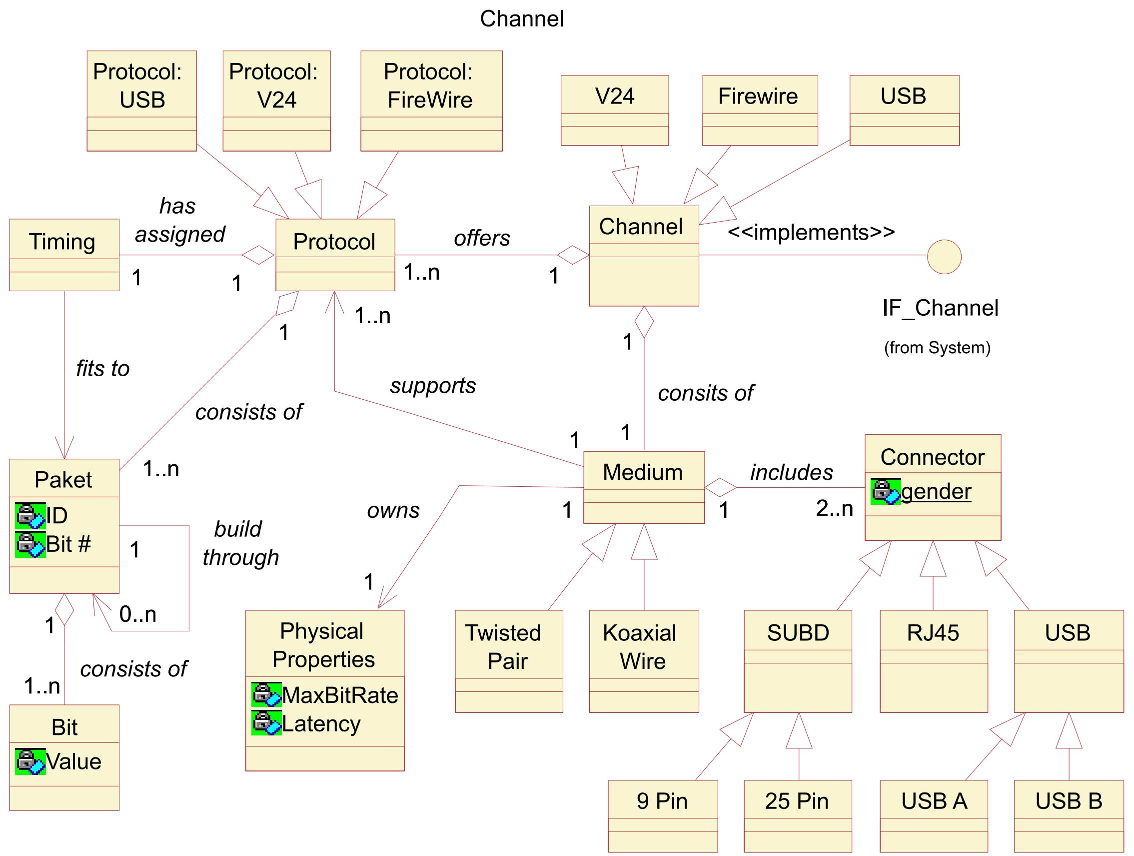


Abbildung 3.16: Modellierung des Kanals

Im entwickelten Klassendiagramm (vgl. Abbildung 3.16) wird das dadurch sichtbar, dass die zentrale Klasse Kanal (Channel) aus genau einem Medium besteht (consists of) und bis zu n Kanäle anbietet (offers). Wie schon im UseCase betont, kommen allerdings nur solche Protokolle zum Einsatz, die vom Medium unterstützt (supports) werden. Jedes Medium beinhaltet (includes) mindestens 2 Steckverbindungen (Connector), die als Attribut das Geschlecht (gender) vorweisen. Mögliche Realisierungen solcher Steckverbindungen können, wie bereits im Kapitel 2 vorgestellt, SUBD-, RJ45- oder USB-Stecker sein. Bei den SUBD-Steckverbindungen gibt es noch die Alternative der 9- oder der 25-poligen Variante. An deren Stelle sind bei USB die Steckertypen USB A und USB B

verfügbar. Zusätzlich zu den Steckverbindungen besitzt (owns) jedes Medium noch seine physikalischen Eigenschaften (Physical Properties). Dazu zählen die Bandbreite, auch als maximale Bitrate (MaxBitRate) bezeichnet, und die Verzögerungs- und Laufzeiten des Mediums.

Das Protokoll (Protocol) des Kanals setzt sich aus eins bis n Paketen zusammen (consists of) und verfügt über (has assigned) exakt ein Timing, das die genaue Zeiteinteilung des Paketes beschreibt (fits to). Ein Datenpaket trägt einen Paketbezeichner (ID) und die beinhaltete Anzahl von Bits (Bit#) und besteht aus Null bis n weiteren Paketen. So können auch aufwendige Paketstrukturen, wie sie beim Durchreichen von Paketen durch verschiedenen Schichten entstehen, beschrieben werden. Letztendlich besteht (consists of) jedes Paket aus Bits, die durch den Wert Null oder Eins charakterisiert werden.

3.4.4 Timing dependency Graph (TDG)

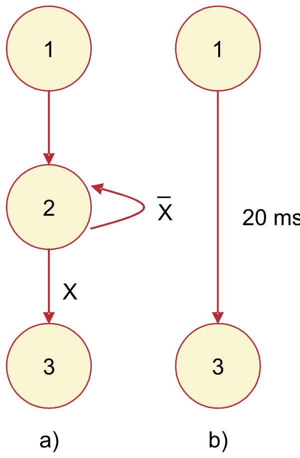


Abbildung 3.17: a) Protokoll-FSM und b) Zeitrestriktionen im Protokoll-FSM

Zum Abschluss wird noch auf eine weitere Möglichkeit verwiesen, wie die Modellierung des Zeitaspekts erfolgen kann. Die Idee dazu stammt aus [HLM00]. Der Timing Dependency Graph (TDG) ist nicht Bestandteil von UML. Es handelt es sich dabei um ein Graphenmodell, das die Zeitabhängigkeiten eines zu steuernden Kommunikationsgraphen überwacht. Äquivalent zu den Kommunikationsgraphen können auch FSM bzw. Statecharts eingesetzt werden. Den TDG kann man durch Transformation einer Kommunikations-FSM, z. B. eines Statechart, entwickeln. In der Kommunikations-FSM sind die Datenabhängigkeiten gegeben, wie in Abbildung 3.17 a) zu sehen ist. Um die zeitlichen Abhängigkeiten zu beschreiben, werden unter Punkt b) zuerst alle Kanten des Graphen entfernt und dann zwischen Knoten, bei denen zeitliche Vorgaben bestehen, neue Kanten eingefügt. Diese Kanten beschreiben die maximalen Berechnungszeiten oder Time-Outs.

Um den TDG zu implementieren, werden im Kommunikations-Graph notwendige Trans-

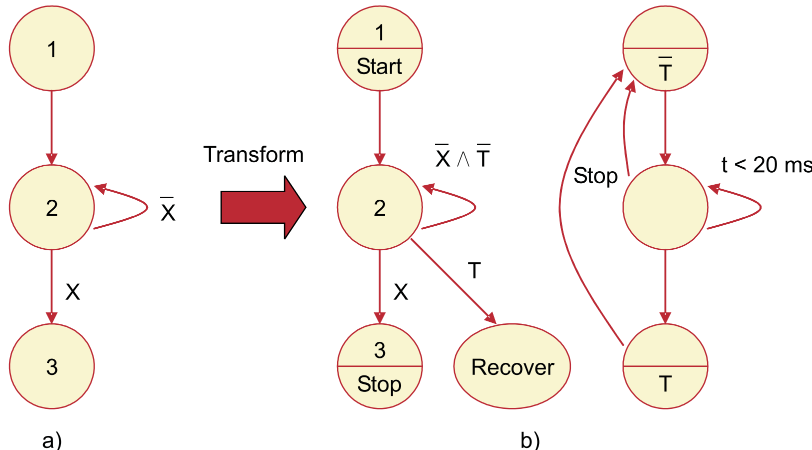


Abbildung 3.18: a) Protokoll-FSM und b) Modifikation der Protokoll-FSM mit zusätzlicher Timer-FSM

formationen vorgenommen (vgl. Abbildung 3.18 a)) und, wie unter b) dargestellt, eine Timer-FSM zusätzlich eingeführt. Diese Timer-FSM beruht auf den Zeitschranken der Kommunikations-FSM und überwacht dort die Einhaltung der Zeitabhängigkeiten. Können die Zeitvorgaben nicht eingehalten werden, so sorgt die Implementierung des TDG für ein deterministisches Fail-Save-Verhalten im Kommunikationsgraphen.

3.5 Ergebnisse

Dieses Kapitel hat eine Modellierung aller erforderlichen Aspekte von Echtzeitschnittstellen vorgestellt. Das Ergebnis wurde als *Modellierungskonzepts* formuliert. Ausgehend von den sehr abstrakten Anforderungen einer Task und den Vorgaben eines Kanals ist eine systematische Top-Down-Modellierung der zugehörigen Schnittstelle in Form von UML-Diagrammen entstanden. Dazu wurde eine ausführliche und allgemeine Analyse zur Modellierung von Tasks und Echtzeitkanälen durchgeführt.

Im Kontext des Modellierungskonzepts ist dann die Bedeutung des Entwurfskonzepts aufgezeigt worden. Die Implementierung und die Realisierungsmöglichkeiten des Entwurfskonzepts werden im folgenden Kapitel unter Zuhilfenahme dieses Modellierungskonzepts erläutert. Das Modellierungskonzept kann als Ausgangspunkt einer automatischen Codegenerierung dienen und ist somit auch eine Grundlage des automatisierten Entwurfs.

4 Entwurf von Echtzeitschnittstellen

In diesem Kapitel *Entwurf von Echtzeitschnittstellen* wird das im Rahmen dieser Arbeit entwickelte Entwurfskonzept vorgestellt. Die Darstellung erfolgt mit Hilfe der Werkzeuge aus dem Abschnitt Modellierung. Im Abschnitt 4.4 wird das Konzept am Beispiel der seriellen Schnittstelle demonstriert. Als ein komplexeres Beispiel ist im Kapitel 5 ein TTP-Modell unter Anwendung des Entwurfskonzepts modelliert worden.

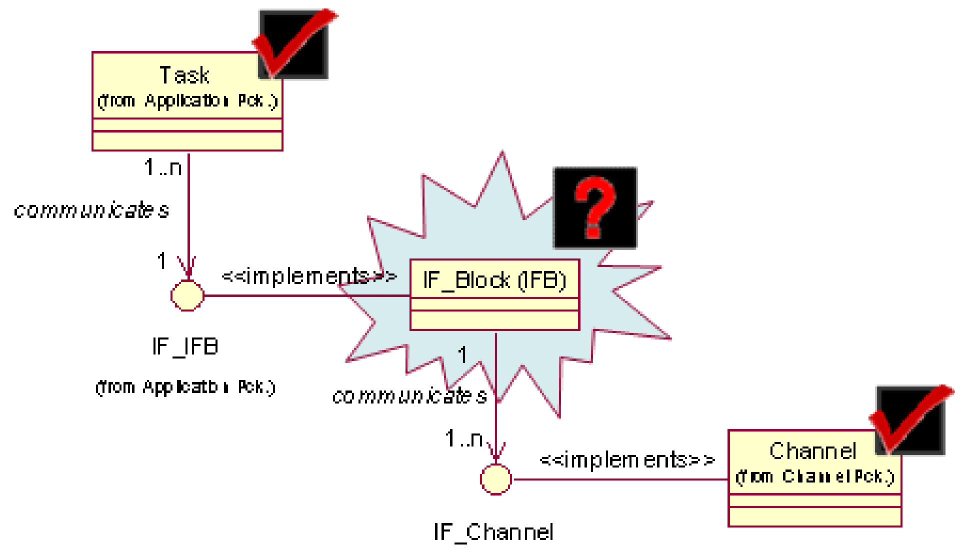


Abbildung 4.1: Der Interface-Block im Modellierungskonzept: Task – Channel

Das Kapitel 3 ist bereits ausführlich auf die Task und den Kanal eingegangen. Es wurden dort zwei Fälle betrachtet: Zum einen der Anschluss einer Task an einen Kanal und zum anderen die Adaption einer Task an eine bereits bestehende Task (vgl. Abbildung 4.1). Das Bindeglied zwischen Task und Kanal wurde zuvor einfach als Interface-Block (IFB) bezeichnet und bisher nicht näher betrachtet. Der IFB ist der zentrale Baustein, der die Kommunikation zwischen Task und Kanal ermöglicht. Genau an dieser Stelle setzt das Entwurfskonzept auf. Ebenso wie bei der Kommunikation zwischen Task und Kanal kann der IFB bei der Adaptierung eingesetzt werden. Dabei verbindet der IFB eine zu adaptierende Task (*TaskToAdapt*) mit einer im System bereits vorhandenen Task (*Existing_ Task*)(vgl. Abbildung 4.2). Die weiteren Erläuterungen beziehen sich auf den ersten Fall, d. h. der Kommunikation zwischen Task und Kanal. Am Ende dieses

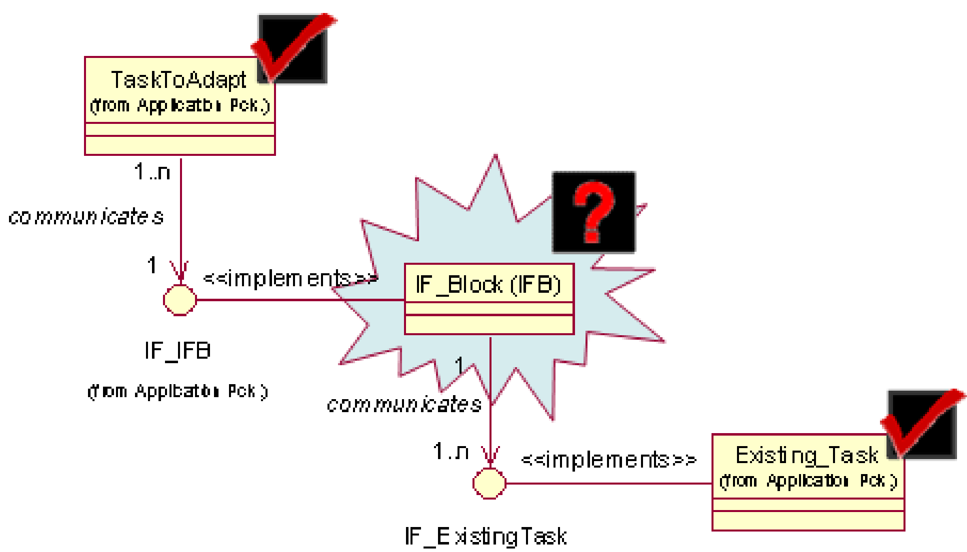


Abbildung 4.2: Der Interface-Block im Modellierungskonzept: Adaption einer Task

Abschnitts wird dann noch eine kurze Betrachtung in Bezug auf die Adaptierbarkeit angestellt.

4.1 Die Idee: Das Entwurfskonzept

Um das Entwurfskonzept strukturiert zu beschreiben, wird zunächst der Aufbau anhand von Paket Diagrammen eingeführt. Durch die Verwendung von Paketen wird das Konzept hierarchisch in Ebenen (Level) eingeteilt. Auf der obersten Ebene befindet sich das *System-Level* (vgl. Abbildung 4.3). Darin enthalten ist nur ein Paket mit der Bezeichnung *System*, welches die gesamte Schnittstelle kapselt .

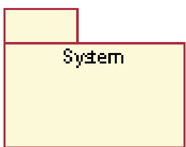


Abbildung 4.3: Modellierung des System-Level

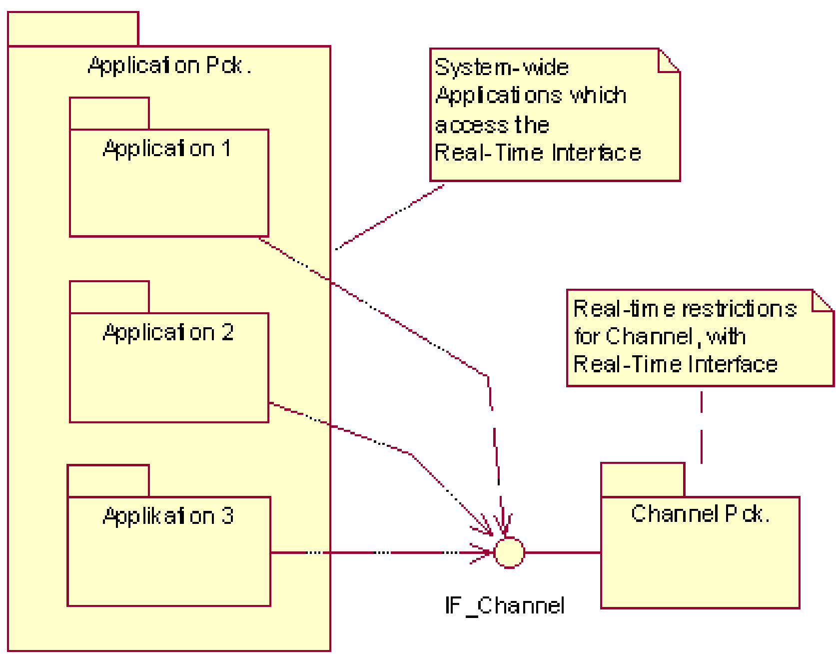


Abbildung 4.4: Modellierung des Application-Channel-Level

Eine Hierarchiestufe darunter befindet sich das *Application-Channel-Level*, in der die Applikation und der Kanal in Form von Paketen beschrieben sind. Wie in Abbildung 4.4 zu erkennen ist, setzt sich diese Ebene aus dem Kanal und seiner Schnittstelle (*IF_ Channel*) sowie einer Reihe von Applikationen, die Teile des Applikations-Paketes sind, zusammen. Es ist offen gelassen, ob eine solche Applikation durch schreibende, lesende oder bidirektionale Operationen auf die Schnittstelle des Kanals zugreift.

Der Begriff *Applikation* (Application) wird an dieser Stelle zum ersten mal erwähnt. Im Rahmen dieser Arbeit soll der Begriff auf folgende Weise definiert sein: *Eine Applikation besteht aus einer Menge von einer bis n Tasks und verfügt über genau einen zugehörigen Interface-Block (IFB).* In Abbildung 4.5 wird dieser Sachverhalt durch ein UML-Packetdiagramm veranschaulicht. Dort ist das *Task-IFB-Level* zu sehen, die Ebene, auf der das Task-Paket (Task Package) und der IFB modelliert sind. Das Task-Paket enthält eine Reihe von einer bis zu m Tasks, die auf die vom IFB bereitgestellte Schnittstelle, die als *Interface des IFB* (*IF_ IFB*) bezeichnet wird, zugreifen können. Die Richtung und die geforderten Zeitpunkte des Zugriffs werden prinzipiell von der Task vorgegeben. Allerdings verwaltet der IFB die Kontrolle seiner Schnittstelle autonom und kann die Tasks dementsprechend steuern. Im *Application-Channel-Level* ist die Schnittstelle des Kanals zum Applikations-Paket als *IF_ Channel* eingeführt worden. Da der IFB die Kommunikation zwischen Task und Channel abwickelt (vgl. Abbildung 3.9), wird auf dieser Ebene (*Task-IFB-Level*) die Verbindung zwischen dem IFB und dem *IF_ Channel*

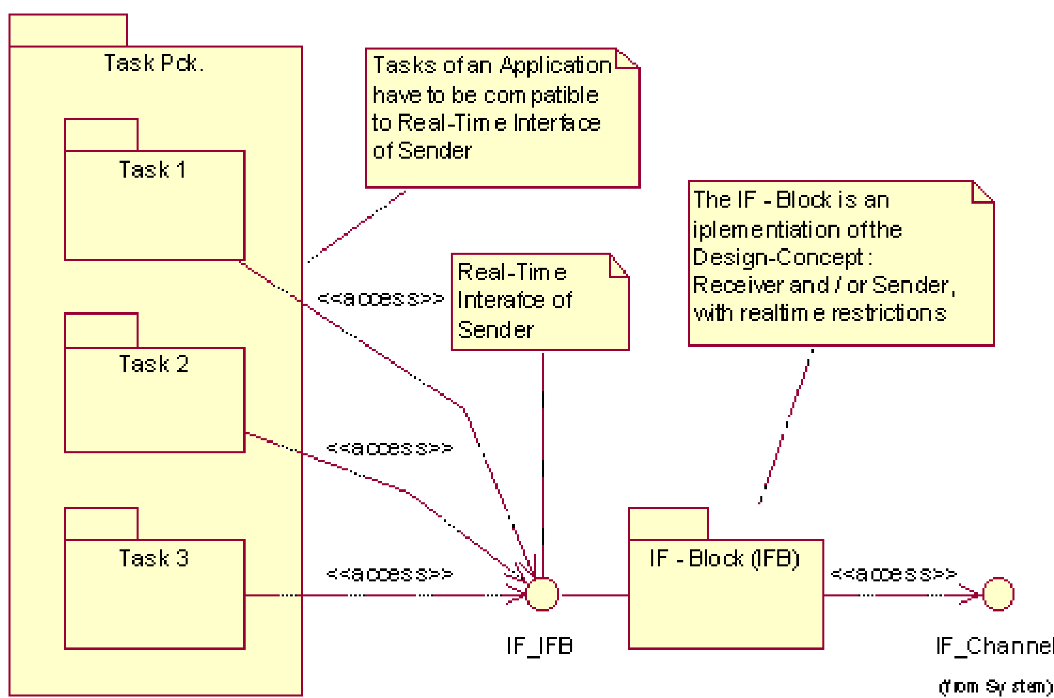


Abbildung 4.5: Modellierung des Task-IFB-Level

hergestellt. Damit stellt sich die Frage, wie genau ein solcher IFB aufgebaut ist und welche Funktionalität er bereitstellt. Eine Antwort darauf geben die folgenden Teilkapitel.

4.1.1 Der Interface-Block (IFB)

Der innere Aufbau eines IFB ist in Abbildung 4.6 durch ein UML-Klassendiagramm modelliert. Die zentrale Klasse im Klassendiagramm des Interface-Blocks ist als (*IF_ Block)*) bezeichnet. Wie anhand der Aggregationen ersichtlich ist, setzt sich der *IF_ Block* aus drei Komponenten zusammen:

- Kontrolleinheit (*Control Unit*),
- Sequenzgenerator (*Sequence Generator*) und
- Protokollgenerator (*Protocol Generator*).

Die *Kontrolleinheit* (*Control Unit*) ist eine Realisierung der Klasse *IFB_ Control*, die zu den Kontroll-Stereotypen zählt. Von ihr ausgehend verlaufen drei Assoziationen zu weiteren Klassen: der UML-Interface-Klasse *IF_ IFB*, dem Sequenzgenerator *Sequence Generator* und dem Protokollgenerator *Protocol Generator*. In jedem IFB gibt es jeweils

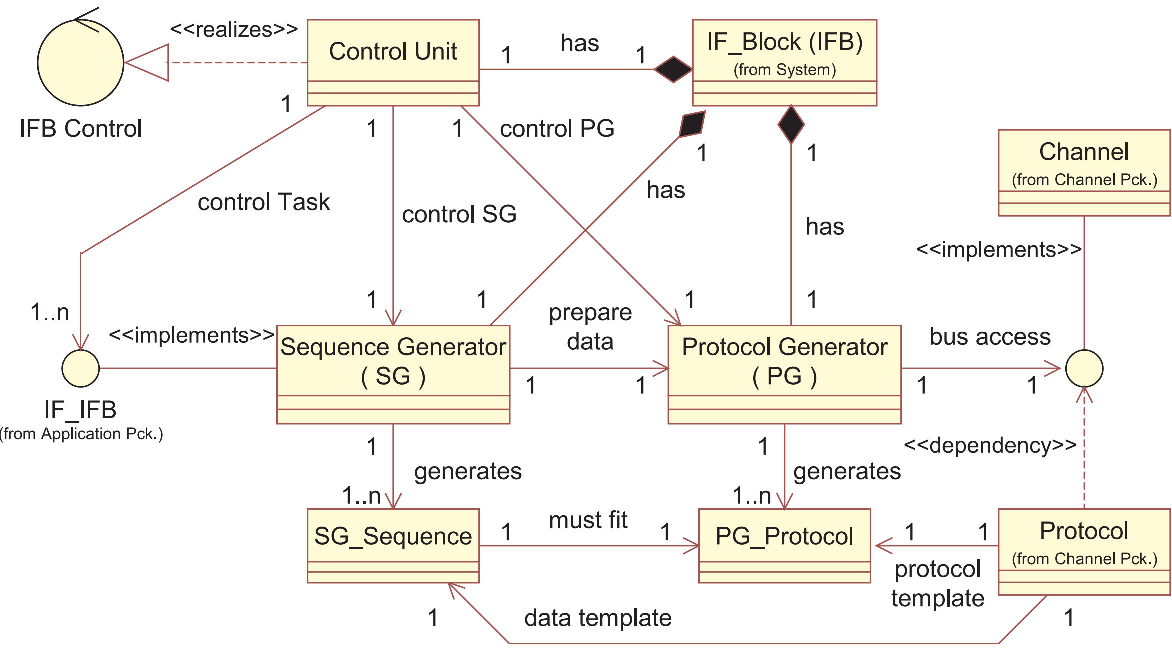


Abbildung 4.6: Modellierung des Interface-Blocks (IFB)

genau eine Kontrolleinheit, einen Sequenz- und einen Protokollgenerator. Der Sequenzgenerator implementiert die Schnittstelle *IF_IFB* und stellt damit die Verbindung zu den Tasks zur Verfügung. Es können eine bis n verschiedenen Sequenzen (*SG_Sequence*) in dieser Generatorstufe erzeugt werden (*can generate*). Weiterhin bereitet der Sequenzgenerator die Daten für den Protokollgenerator vor (*prepare data*). Dieser stellt auf der anderen Seite, kontrolliert durch die Kontrolleinheit, den Zugriff auf die Schnittstelle des Kanals (*bus access*) über Handshake- und Datenleitungen zur Verfügung. Dazu fügt der Protokollgenerator alle Daten protokollgerecht zusammen (*can generate PG_Protocol*). Das Ziel dieses internen Aufbaus eines IFB besteht darin, eine möglichst flexible Funktionalität als Kommunikationsknoten bereitzustellen. Sowohl der Sequenzgenerator als auch der Protokollgenerator richten sich in ihrer Realisierung nach den vorgegebenen Protokollen des Kanals (*data template, protocol template*).

Um die genauen Funktionen der drei Komponenten des IFB zu erklären, wird noch einmal kurz rekapituliert, welche Anforderungen an einen IFB gestellt werden. Eine Echtzeitschnittstelle muss ein absolut vorhersagbares Verhalten haben und eine geforderte Bandbreite einhalten. Handelt es sich bei der Task um ein statisches System, so kann man, ähnlich wie bei einem Betriebssystem, ein statisches Scheduling-Verfahren benutzen, um alle Ereignisse vorher zu berechnen und zu verteilen. Sollten die Tasks in der Summe eine größere Bandbreite fordern als die eingesetzten Kanäle bereitstellen, so ist keine Ablaufplanung (Schedule) durchführbar und das System somit nicht realisierbar. Da die Echtzeitbedingungen nicht verletzt werden dürfen, muss eine statische Kanalzuordnung erfolgen (siehe Kapitel 2.2.1, Prioritätsvergabe und Zugriff auf das Medium). Soll das dynamische Verhalten einer Task mit einbezogen werden, muss der maximale

Wert, der in einem Sendezyklus benötigten Bandbreite, von vornherein statisch in die Vergabe der Sendezeiten eingeplant werden. Wie die Task diese Bandbreite dann ausnutzen wird, bzw. welche Daten als *Interface Present Data* an der Schnittstelle liegen und wie sie dorthin gelangen, verbleibt im Problemfeld der Task. Lediglich die Art der Daten muss vorher allgemein bekannt sein, da die Aufbereitung der Daten von deren Typ abhängig ist.

Die drei wesentlichen Komponenten, die Kontrolleinheit (CU), der Sequenzgenerator (SG) und der Protokollgenerator (PG) werden in den folgenden Abschnitten erläutert.

4.1.2 Die Kontrolleinheit (CU)

Um all die angesprochenen Funktionen zu koordinieren, wird die Kontrolleinheit eingesetzt. Da auf jede Schnittstelle eines Kanals n Applikationen Zugriff haben, sind auch n Kontrolleinheiten in einem System beteiligt. Unter diesen muss zuallererst die statische Kanalaufteilung festgelegt werden, d. h. jede Kontrolleinheit muss für sich wissen, wann sie sende- bzw. empfangsberechtigt ist. So wird der gesamte Sendezyklus unter allen einbezogenen Kontrolleinheiten aufgeteilt. Jede Instanz einer solchen Einheit verfolgt aber auch noch weitere Aufgaben. Sie steuert den Sequenz- und den Protokollgenerator sowie alle in der jeweiligen Applikation eingebundenen Tasks. Diese Funktionen werden bei der Erläuterung des Sequenz- und des Protokollgenerators ersichtlich. Darüber hinaus kann die Kontrolleinheit eine Wächter-Funktion (Guardian-Function) erfüllen, die alle illegalen Operationen außerhalb der festgelegten Kommunikation sofort unterbindet.

4.1.3 Der Sequenzgenerator (SG)

Auf dem Weg von der Applikation zum Kanal ist eine Transformation der Daten in das vom Kanal vorgegebenen Protokoll zu erledigen. Dazu durchlaufen die Daten zwei Generatorstufen, die über Handshakes untereinander an die Task und an den Kanal gekoppelt sind. Der erste Generator erstellt Datensequenzen. Aus diesem Grund wird er als Sequenzgenerator bezeichnet. Er ist über Datenleitungen und Steuerleitungen, die für das Handshake benötigt werden, an die entsprechenden Tasks angeschlossen. Der Sequenzgenerator wird aus den folgenden drei Gründen benötigt.

Erstens muss er die Daten der Tasks, die in einer Sequenz untergebracht werden sollen, sammeln und zu Sequenzen zusammenfassen. Dies geschieht in Zusammenarbeit mit der Kontrolleinheit. Falls nur eine Task existiert oder die Tasks eines statischen Systems über einen eigenen Controller verfügen, der die eigene Schnittstelle und das dazugehörige Protokoll verwalten kann, ist es relativ einfach. Unter diesen Bedingungen kann die Task, bzw. können die Tasks, einfach an den Sequenzgenerator angeschlossen werden und die Kommunikation wird durch das Handshake zwischen Task und Sequenzgenerator hinreichend geregelt. Dann übernimmt die Kontrolleinheit lediglich die Funktion als Wächter und kontrolliert die Einhaltung des vereinbarten Datenverkehrs (*monitoring of Data-Patterns*), der in der Ablaufplanung festgelegt wurde. Sind allerdings nebenläufige Tasks vorhanden, die keine Möglichkeit haben ihre Schnittstelle zu beeinflussen, wie

z. B. eine Gruppe einfacher Sensoren oder liegt ein dynamisches System vor, so muss die Kontrolleinheit aushelfen und die Kommunikation zwischen Tasks und Sequenzgenerator steuern. Dazu werden von der Steuereinheit die entsprechenden Schnittstellen zu den betroffenen Tasks aktiviert, bzw. deaktiviert. Das Schema für den Aktivierungsprozess der Tasks, d. h. die lokale Ablaufplanung, ist in diesem Fall in der Kontrolleinheit implementiert und resultiert aus der globalen Ablaufplanung, in der jeder Task des Systems eine exakte Sendezeit und Bandbreite zugesichert wurde.

Ein zweiter Grund, warum der Sequenzgenerator benötigt wird, ist die Anpassung der Daten an die Zieltask. In dieser Generatorstufe können alle gesammelten Daten angeordnet und in Pakete verpackt werden. Weiterhin ist der Sequenzgenerator in der Lage, gesteuert durch die Kontrolleinheit, zu den gesammelten Daten automatisch generierte Daten hinzuzufügen. Das ist sehr praktisch, wenn Zieltasks einen bestimmten Datenstrom erwarten. Dann muss die Quelltask lediglich den dynamischen Teil der Daten generieren, der statische Teil wird im Sequenzgenerator hinzugefügt.

Als dritte wichtige Aufgabe des Sequenzgenerators ist noch die Vorverarbeitung aller Daten für den Protokollgenerator zu erwähnen. Die notwendigen Informationen über das Format der vorzubereitenden Daten bezieht der Sequenzgenerator aus den Angaben des Kanals (*data template*). Beide Generatorstufen stehen über Daten- und Handshake-Leitungen in Verbindung. Um eine möglichst flexible Funktionalität anzubieten, sind verschiedene Arbeitsmodi sowohl im Sequenz- als auch im Protokollgenerator erlaubt. Jeder Modus bezeichnet die Erstellung einer bestimmten Datensequenz bzw. eines bestimmten Protokolls. Welcher Modus der beiden Generatorstufen jeweils aktiv ist, bestimmt die Kontrolleinheit. Eine Kommunikation kann nur zwischen den beiden aktiven Komponenten der Generatorstufen stattfindet. Die Anzahl der unterschiedlichen Modi richtet sich nach dem Bedarf der Task und des Kanals.

4.1.4 Der Protokollgenerator (PG)

Viele Medien unterstützen mehrere Protokolle, was im Modellierungskapitel bei der Beschreibung des Kanals berücksichtigt wurde. Deshalb benötigt man einen Protokollgenerator, der die Daten für den Transfer auf dem Kanal vorbereitet und dann protokollgerecht versendet. Dazu stehen Daten- und Handshakeleitungen zur Verfügung. Die Kommunikation des Protokollgenerators mit dem Sequenzgenerator und dem Kanal wird von der Kontrolleinheit gesteuert, ebenso wie der Wechsel des zu generierenden Protokolls. Die Änderung des aktiven Protokolls wird als Moduswechsel bezeichnet. Zu jeder implementierten Protokollvariante des Protokollgenerators existiert als Vorlage ein vom Kanal bereitgestelltes Protokoll (*protocol template*).

Die Realisierung des Protokollgenerators erfolgt im Allgemeinen durch HW, da er unter anderem die physikalische Schnittstelle zum Kanal darstellt. Eine Einschränkung soll hier erwähnt werden: Der IFB ist bisher nur für digitale Systeme mit einer eben solchen Datenübertragung entworfen. Durch den Einsatz von Modulatoren könnte man die physikalische Schnittstelle um das analoge Versenden von Daten erweitern. Zum Erstellen protokollgerechter Daten gehört weiterhin die Fähigkeit der Transformation von Daten.

Auch das ist eine Funktion, die der Protokollgenerator im IFB ermöglicht, wenn entsprechende Hardware vorhanden ist. In Systemen mit nur einem Spannungslevel muss dazu eine Zusatzhardware, wie z. B. eine Umsetzerkarte, vor den Kanal geschaltet werden.

Die bisherigen Erläuterungen beziehen sich ausschließlich auf das Versenden von Daten. Für den Empfang von Daten wird der IFB in „umgekehrter“ Datenflussrichtung eingesetzt. D. h. die Daten werden durch den Protokollgenerator vom Kanal aufgenommen und entpackt. Diese bereitet der Sequenzgenerator dann für die einzelnen Tasks auf.

4.2 Implementierung der Komponenten

Die Komponenten des IFB werden zu großen Teilen durch Automaten (FSM) implementiert. Diese ermöglichen den strukturellen Entwurf von Steuerwerken unter Berücksichtigung des Echtzeitaspekts. Das ist möglich, weil das Schaltverhalten von Automaten unter Berücksichtigung der Semantik der Darstellungsform exakt berechenbar ist. Auch wird die Modellierung der Funktionalität des IFB durch Statecharts dadurch besonders einfach. Ein weiterer Vorteil der Automaten ist die einfache Synthetisierbarkeit zu HW-Schaltungen.

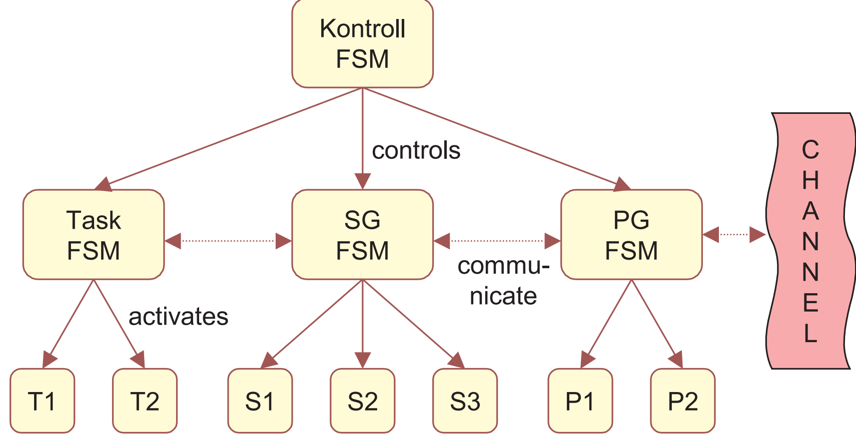


Abbildung 4.7: Automatenhierarchie des IFB

Wenn die Kontrolleinheit, der Sequenzgenerator und der Protokollgenerator durch Automaten implementiert sind, können die bestehenden Abhängigkeiten in Form von kommunizierenden FSM ausgedrückt werden. Das betrifft insbesondere die Umsetzung der Handshakes und die Steuerung der Generatorstufen durch die Kontrolleinheit. Dazu sind die Automaten hierarchisch strukturiert worden (vgl. Abbildung 4.7). Auf der obersten Ebene befindet sich der Automat der Kontrolleinheit, der mit allen weiteren FSM der einzelnen Komponenten kommuniziert. Stellt man sich die Tasks auch als Automat vor, so gibt es auf der Hierarchiestufe unter der Kontroll-FSM drei weitere endliche Automaten: Einen, der die Schnittstelle zu allen Tasks bildet, einen Zweiten als oberste Instanz des Sequenzgenerators und den dritten Automat, der die Modi des Protokollgenerators

kapselt. Jede Task und jeder Modus einer Generatorstufe werden in der dritten Hierarchieebene wiederum von einem Automaten umgesetzt. Die Aktivierung erfolgt durch eine der drei übergeordneten Kontroll-Instanzen.

4.2.1 Bezug zur Adaptierbarkeit

Ebenso wie bei der Kommunikation einer Task mit einem Kanal, ist es mit einem IFB möglich, eine Task an eine bereits existierende Task (*Basiskomponente*) zu adaptieren. Der Aufbau des IFB bleibt erhalten, jedoch die Aufgaben der beiden Generatorstufen ändern sich geringfügig.

Der Sequenzgenerator erzeugt aus den Daten der zu integrierenden Task die für die Basiskomponente benötigten Datensequenzen. Diese werden dann vom Protokollgenerator mit der Basiskomponente ausgetauscht. Dabei besteht die hauptsächliche Funktion des Protokollgenerators bei der Adaption darin, die Daten zeitgerecht auf die Schnittstelle zur Basistask zu geben. Weiterhin kann hier auch die bereits erwähnte Transformation der Daten erfolgen.

4.2.2 Automatischer Entwurf

Die Modellierung in Zusammenhang mit dem IFB ist ein Schritt in Richtung des automatisierten Entwurfs, da sie für viele Problemstellungen eine einheitliche und beherrschbare Realisierung bietet. Der Entwurfsprozess beschränkt sich bei Anwendung des Entwurfskonzepts darauf, die Daten der Tasks und des Kanals, deren Vorgaben in den UML Modellen enthalten sind, zu extrahiert und den Aufbau des IFB daraus zu vollziehen. Die Modellierung des Kanals und der Task müssen dazu gegebenenfalls noch verfeinert und ergänzt werden. Das setzt allerdings die Möglichkeit der Auswertung von UML-Diagrammen und einer automatischen Codegenerierung voraus. Ein Ansatz dafür wird im folgenden Teilkapitel erläutert.

4.2.3 Ansatz einer Codegenerierung

Für UML gibt es bereits einige bestehende Ansätze der Codegenerierung. Die Zielsprachen sind bisher allerdings reine SW-Hochsprachen, wie z. B. Java. Die Idee besteht nun darin, anstelle einer solchen SW-Hochsprache eine hardwarenahe Sprache wie VHDL oder Verilog aus den Diagrammen zu generieren. In Teilbereichen ist eine Umsetzung leicht vorstellbar, wie z. B. die Transformation von Statecharts zu endlichen Automaten in VHDL-Code. Dazu ist jedoch eine exakte Definition der Syntax und Semantik von Statecharts in Bezug auf eine Auswertung für VHDL notwendig. Diese muss eindeutig sein und mit den Möglichkeiten der Zielsprache kooperieren. Es treten dabei zwangsläufig Restriktionen auf, welche die Freiheitsgrade der Modellierung einschränken. Im folgenden Kapitel werden einige Restriktionen davon beschrieben, die sich in dieser Arbeit abgezeichnet haben.

4.3 Restriktionen im HW-Entwurf

Je nach Granularität erfasst die Modellierung unterschiedlich viele Systemeigenschaften. Durch den Einsatz spezieller Hardware, auf der ein IFB realisiert werden soll, handelt man sich zusätzlich einige Restriktionen ein, die bisher noch nicht betrachtet wurden aber an die verwendete HW gebunden sind. Ebenso haben Programmiersprachen und Entwurfswerkzeuge ihre Grenzen. In dieser Arbeit waren folgende Restriktionen während des Entwurfsprozesses zu berücksichtigen:

- HW-Restriktionen
  - Beschränkung auf HW-Komponenten
  - FPGAs als Rapid Prototyping Plattform
- SW-Restriktionen
  - VHDL
  - Synthese
  - Angewendete Programmierwerkzeuge

Beschränkung auf HW-Komponenten

Dieser Arbeit beschränkt sich auf den Einsatz von HW-Komponenten. Darunter ist zu verstehen, dass Tasks, IFB und Kanal in HW vorhanden oder realisiert sind. Um die Testbarkeit und Realisierung möglichst einfach zu gestalten, ist der IFB auf einem FPGA-Board implementiert worden. Dieses kann im Rahmen des Entwurfs von Echtzeitschnittstellen als Spezial-HW mit eigenen Restriktionen angesehen werden. Ebenso geben die Roboter des Demonstrators (siehe Kapitel 5) genau definierte Randbedingungen vor.

FPGAs als Rapid Prototyping Plattform

Ein *FPGA* (Field Programmable Gate Array) ist eine im Demonstrator verwendete HW-Komponente, die unter anderem dazu eingesetzt wird, um darauf den IFB zu implementieren. Grob gesagt besteht ein FPGA aus einer Menge von konfigurierbaren Logikblöcken und deren Verbindungsnetzwerk. Dazu kommen noch die I / O-Funktionalität und diverse Speicherelemente. Die FPGAs werden zu den *Rapid Prototyping Plattformen* gezählt, mit deren Hilfe in kurzer Zeit HW-Schaltungen aufgebaut und überprüft werden können. Für die Ein- / Ausgabe verwendet das hier eingesetzte Spyder-Board, ein FPGA der Firma Xilinx [Xil01], zwei Erweiterungsstecker (Extensionheader, EH). Programmiert wird ein FPGA durch einen Bitstream, der durch die Synthese von Programmcode einer hardwarenahen Hochsprache, wie VHDL oder Verilog, erzeugt werden kann. Dazu werden u. A. Werkzeuge und Compiler der Firma Synopsys und Altera eingesetzt.

VHDL und Synthese

Der Vorgang, um ein FPGA zu programmieren, verläuft folgendermaßen: Zuerst wird der Programmcode in einer hardwarenahen Hochsprache verfasst, der anschließend compiliert wird. Danach wird das Design analysiert um, es auf der Gatterebene darzustellen. Die Synthese generiert daraus den Bitstream, der zur Programmierung der entsprechenden Zielplattform eingesetzt werden kann. Die HW-Restriktionen des FPGAs, wie z. B. die maximale Anzahl an verfügbaren Taktleitungen, die maximale Taktrate oder die Laufzeit des kritischen Pfades, überträgt sich durch die Synthese rückwirkend auf die Programmiersprache. Auch die Synthese selbst gibt spezielle Einschränkungen vor, welche Konstrukte abgebildet werden können. Die Hardware-Beschreibungssprache VHDL beinhaltet neben speziellen auch viele Elemente einer „gewöhnlichen“ Programmiersprache, wie z. B. Java oder C. Dadurch besitzt VHDL eine große Ausdrucksmächtigkeit, aber nur ein bestimmter Teil aller Möglichkeiten sind synthetisierbar. So können selbst Programme, die sich in der Simulation als korrekt erwiesen haben, fehlerhaft durch den Synthesevorgang abgebildet werden. Das kann zum einen daran liegen, dass die Simulation unter nicht ausreichenden Bedingungen stattgefunden hat oder dass bei der Programmierung nicht alle Restriktionen eingehalten wurden. Es ist allerdings auch nicht auszuschließen, dass die eingesetzten Werkzeuge völlig fehlerfrei arbeiten. Angaben zu VHDL finden sich in der Literatur unter [ML92, Mäd01a, TH95, LSU89], Hinweise zur HW-Synthese stammen aus folgenden Büchern und Skripten [Smi00, Mäd01b, Mäd99, CC94].

Bei all den Restriktionen ist zu beachten, dass es allerdings auch bestimmte Konstrukte und Ausdrucksmöglichkeiten einer Programmiersprache wie VHDL gibt, die besonders gut synthetisiert werden können. Es ist wichtig diesen Punkt zu beachten und die Modellierung danach auszulegen, weil es implizit ein Ziel des automatisierten Entwurfs ist, für Spezialhardware Sourcecode zu generieren. Dazu ist es nicht ausreichend lediglich zu schauen, wie sich UML-Diagramme am besten in VHDL abbilden lassen, sondern es müssen alle eben angedeuteten Restriktionen dabei mit einbezogen werden. Daraus resultiert auch der Rückschluss, dass es unnötig ist, etwas zu modellieren, was nicht fehlerfrei auf die Zielhardware abgebildet werden kann.

Eine allgemeine Einschränkung des Hardwareentwurfs ist die stark eingeschränkte Möglichkeit des Debuggens, die manchmal nur unter erheblichen Aufwand möglich ist. Im Gegensatz zur Software, in der schrittweises Debuggen und Testausgaben die Fehlersuche erleichtern, muss in der Hardware oft eine zusätzliche Komponente eingefügt oder aufwendig gemessen werden. Manchmal hilft auch die Interpretation eines fehlerhaften Verhaltens auf den Fehler rückzuschließen.

Angewendete Programmierwerkzeuge

Es gibt mehrere Anbieter von FPGA-Werkzeugen, z. B. Synopsys [Syn98] und Altera [Alt01]. Man muss dabei beachten, dass der Funktionsumfang und der Ablauf der Synthese von Tool zu Tool variieren. Altera bietet meistens Komplettlösungen an, bei denen man durch das Drücken einiger Knöpfe bereits fertigen Bit-Code erhält. Alternativ dazu gibt es Sammlungen von kompatiblen Werkzeugen, die man nacheinander anwenden

muss, um das FPGA zu programmieren. Das erfordert ein gewisses Maß an Vorwissen, bietet aber den Vorteil, im Synthesevorgang einen größeren Einfluss auf das Ergebnis nehmen zu können. Im Rahmen dieser Arbeit wurde eine solche Kollektion von Werkzeugen eingesetzt:

- nedit – Texteditor mit Syntaxhighlighting für VHDL
- SGE – Synopsys Graphical Editor, CAD-Tool
- DesignAnalyzer – Konvertierung von VHDL in Gatterschaltung
- DsgnMgr – Synthesewerkzeug für Xilinx-FPGA: Spyder Virtex

4.4 Entwurfsbeispiel: Serielle Schnittstelle (V24)

Um eine bessere Vorstellung davon zu bekommen, wie das Entwurfskonzept angewendet werden kann, wird es am Beispiel der seriellen Schnittstelle illustriert. Die serielle Schnittstelle ist fest definiert und deshalb hier als Beispiel ausgewählt. Hinzu kommt, dass die V24-Schnittstelle nicht zu komplex ist und daher als einleitendes Beispiel gut geeignet ist. Außerdem wird die V24-Schnittstelle mehrfach im Demonstrator verwendet, daher ist die Erklärung aus dem Folgekapitel nur vorgezogen.

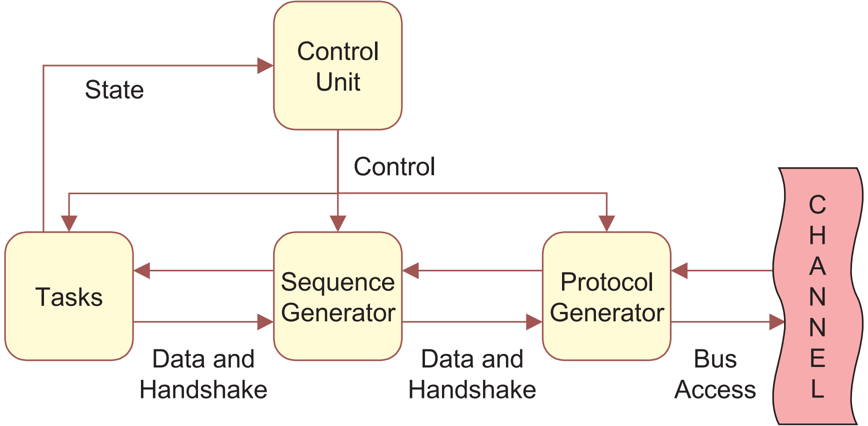


Abbildung 4.8: Modellierung der Task für die manuelle Robotersteuerung

In Abbildung 4.8 ist das Entwurfskonzept auf das Modell der V24-Schnittstelle, die im Demonstrator die Steuerung des Roboters ermöglicht, angewendet worden. Es handelt sich bei dem Modell um einen Spezialfall des in Kapitel 4.2 allgemein beschriebenen FSM-Modells. Enthalten ist die Kontrolleinheit, die die Tasks, den Sequenzgenerator und den Protokollgenerator steuert. Als Rückkopplung kann die Kontrolleinheit den Zustand (*State*) der Tasks auswerten. Die Verbindung zwischen Task, PG und SG ist durch Daten- und Handshakeleitungen realisiert, außerdem ist der PG mit dem Kanal

verbunden (*Bus Access*). Nachfolgend wird zunächst die Task, dann die Kontrolleinheit und die beiden Generatorstufen beschrieben. Ein abschließendes Diagramm geht näher auf die Verbindungen der Automaten ein.

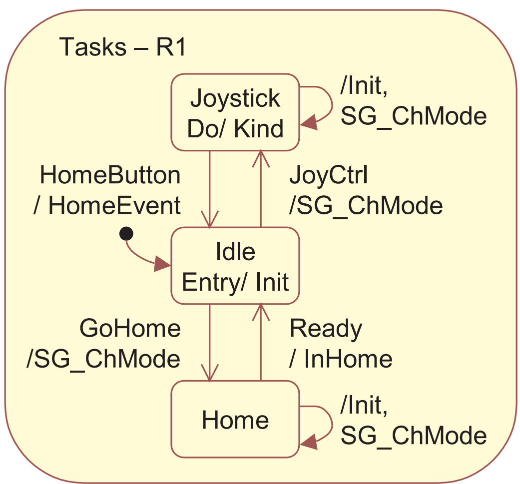


Abbildung 4.9: Modellierung der Task der Roboter

Der Automat der Task, der die manuelle Steuerung eines Roboters symbolisiert, besteht aus drei Zuständen: *Idle, Joystick* und *Home* (vgl. Abbildung 4.9). Der Moduswechsel zwischen Joystick und Home verläuft immer über den Idle-Zustand. In den Zustandsübergang zwischen der manuellen Steuerung (Joystick) und dem Homing-Prozess (Home) ist die Kontrolleinheit einbezogen. Die Automaten kommunizieren über Ereignisse, die durch die Zustandsübergänge erzeugt werden, z. B. */HomeEvent oder /InHome* oder Ereignisse, die innerhalb einer Task angestoßen werden (Entry-, Do-, Exit-Funktionalität). Im Automaten der Kontrolleinheit (vgl. Abbildung 4.10) lösen die generierten Ereignisse der Task Zustandsübergänge aus. Diese wiederum erzeugen Signale, die zur Steuerung der Task, des SG oder des PG genutzt werden.

Wie in Abbildung 4.10 zu sehen ist, verwaltet die Kontrolleinheit drei nebenläufige Prozesse: Die Steuerung der Tasks und die Verwaltung der beiden Generatorstufen. Links im Bild befindet sich der zur Task zugehörige Kontrollautomat. Im mittleren Bereich der Abbildung ist die Verwaltung der Modi des Sequenzgenerators dargestellt. Aus Platzgründen sind nur die Übergänge der zweiten Sequenz modelliert, die beiden anderen Fälle sind jedoch äquivalent dazu. Für den Protokollgenerator, der in diesem Fall nur einen Modus hat, ist eine einfache Resetlogik auf der rechten Seite beschrieben. Die hier aufgeführten Grafiken sind aus Gründen der besseren Darstellbarkeit nicht ganz vollständig und weichen daher leicht von der Implementierung ab. Die charakteristischen Merkmale des Entwurfskonzepts sind jedoch vollständig enthalten.

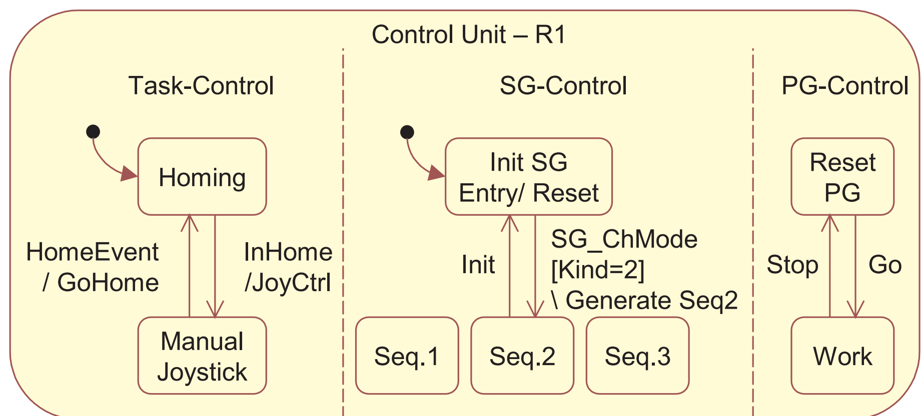


Abbildung 4.10: Modellierung der Kontrolleinheit der V24-Schnittstelle

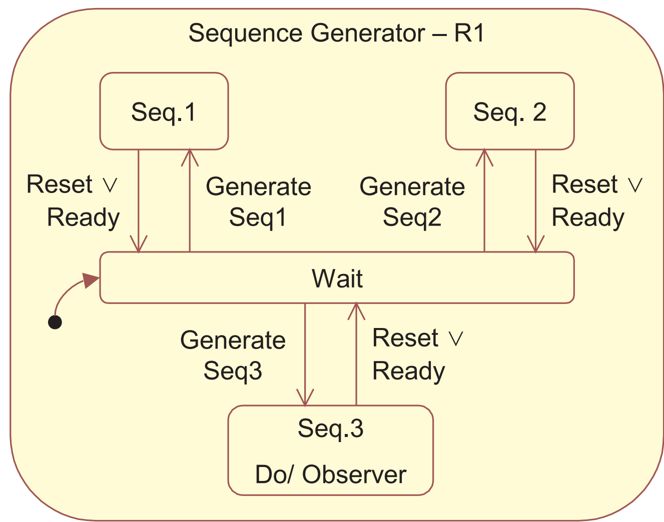


Abbildung 4.11: Modellierung des Sequenzgenerators der V24-Schnittstelle

Die Abbildung 4.11 zeigt den Sequenzgenerator, der aus einer Menge von drei verschiedenen Sequenzen (Modi) besteht. Der Modus *Seq.3* ist dadurch ausgezeichnet, dass ein Observer alle darin ausgeführten Operationen mitprotokolliert. Jeder Zustand im SG steht für die Erzeugung einer bestimmten Sequenz. Nur der Zustand *Wait* bildet eine Ausnahme: Er ist der Ausgangspunkt, von dem aus in die unterschiedlichen Modi verzweigt wird und wohin der Automat nach Fertigstellung einer Sequenz zurückkehrt. Jeder Modus-Zustand in diesem Automat kapselt hierarchisch einen weiteren Automaten, der den Ablauf zur Erzeugung einer Sequenz steuert (vgl. Abbildung 4.12).

Die Aufgabe, die ein solcher Automaten erfüllt, besteht darin, die Daten und das benötigte Handshake für die Generierung der Sequenz zu verwalten. In Abbildung 4.12 ist die Sequenz *Seq.1* des SG dargestellt, die im Demonstrator dazu eingesetzt wird, um ein

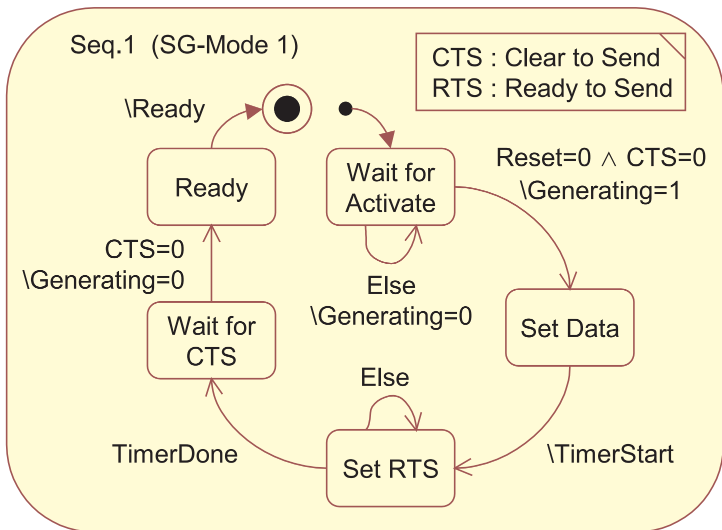


Abbildung 4.12: Modellierung der Modi eines Sequenzgenerators

einzelnes Byte an den Roboter zu schicken. Der Ablauf dazu sieht folgendermaßen aus: Der Automat wartet im *Wait for Activate*-Zustand auf seine Aktivierung und legt nach dem Zustandsübergang zu *Set Data* die Daten auf die Leitung. Darauf folgt einen Takt später das *Ready to Send (RTS)* -Signal, welches angibt, dass die Daten des SG nun zum Lesen bereit sind. Nach Ablauf einer festgelegten Zeit wird das RTS wieder zurückgesetzt und auf das Signal *Clear to Send (CTS)* gewartet, das den empfangsbereiten PG charakterisiert. Zum Abschluss des Vorgangs erfolgt das Senden des *Ready*-Signals, was im SG-Automaten den Übergang in den *Wait*-Zustand hervorruft.

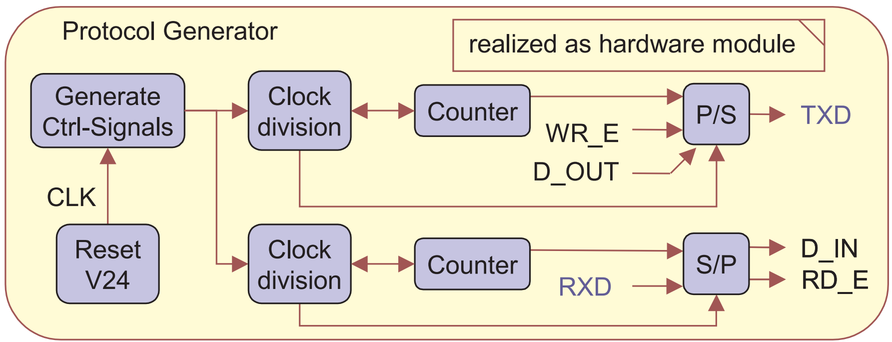


Abbildung 4.13: Modellierung des Protokollgenerators der V24-Schnittstelle

An dieser Stelle wird der Protokollgenerator erläutert, der hier nur einen Modus hat: Das V24-Protokoll, das im PG durch einen UART-Baustein in HW realisiert ist (vgl. Abbildung 4.13). Da im PG nur ein Modus vorhanden ist, kann die Modus-Kapselung durch eine spezielle PG-FSM entfallen. An dessen Stelle tritt sofort der in HW realisierte

Automat. Wichtige HW-Funktionseinheiten sind im PG fliederfarbend dargestellt. Ein weiterer Bestandteil sind die Handshake- und Datenleitungen des UART-Bausteins.

Die Schnittstelle zum Medium sind die *TxD*- und die *RxD*-Datenleitung, auf die Modellierung der Signalmasse ist verzichtet worden.

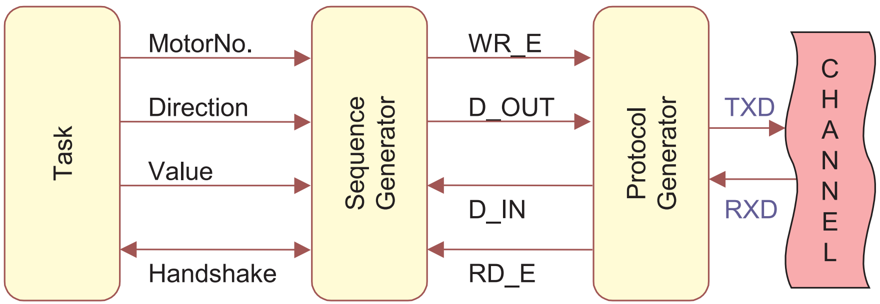


Abbildung 4.14: Modellierung der V24-Schnittstelle

Die Verbindungen zwischen den Generatorstufen sind in Abbildung 4.14 dargestellt. Der Sequenzgenerator kommuniziert mit dem Protokollgenerator über die Write Enable (*WR_E*), Read Enable (*RD_E*), Data Out (*D_OUT*) und Data In (*D_IN*) -Leitungen. Die Verbindung der Task mit dem SG ist durch die Handshakeleitungen und die Datenleitungen realisiert: *MotorNo.*, *Direction* und *Value*.

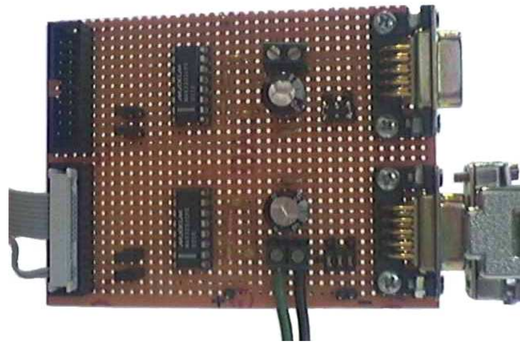


Abbildung 4.15: Foto der Pegelumsetzer-Karte für die V24-Schnittstelle

Im vorherigen Abschnitt 4.1 wurde erwähnt, dass eine Funktion des Protokollgenerators die Transformation von digitalen Signal ist. Das hier vorgestellte Beispiel der seriellen Schnittstelle ist aus dem Demonstrator-Kapitel (vgl. Kapitel 5) vorgezogen. Der Datenverkehr zwischen FPGA und Roboter erfolgt seriell über die V24-Schnittstelle, deren Parameter im Kapitel 2.2 aufgeführt sind. Da das FPGA interne Spannungen von 0 V für logisch Null und 3,3 V für logisch Eins verwendet, ist eine Transformation der Signale auf den V24-Standard notwendig. Diese kann jedoch nicht im eigentlichen Protokollgenerator erfolgen, da das FPGA nicht in der Lage ist das Spannungslevel auf die erforderlichen $\pm 12V$ zu heben. Eine Zusatzhardware, die vor den eigentlichen Kanal

geschaltet wird, leistet diese Funktionalität, was im Demonstrator durch eine speziell angefertigte Pegelumsetzterkarte realisiert ist (vgl. Abbildung 4.15). Das Schaltbild der Karte ist in Abbildung 4.16 dargestellt. Zu sehen sind zwei übereinander angeordnete Kanäle, die jeweils eine Transformation von (0 V, 3.3V) auf (6 V, -6 V) vornehmen. In der Spezifikation von V24 sind Amplituden von $\pm 12V$ vorgegeben, die Werten von $\pm 6V$ reichen in der Steuereinheit schon für eine korrekte Erkennung aus.

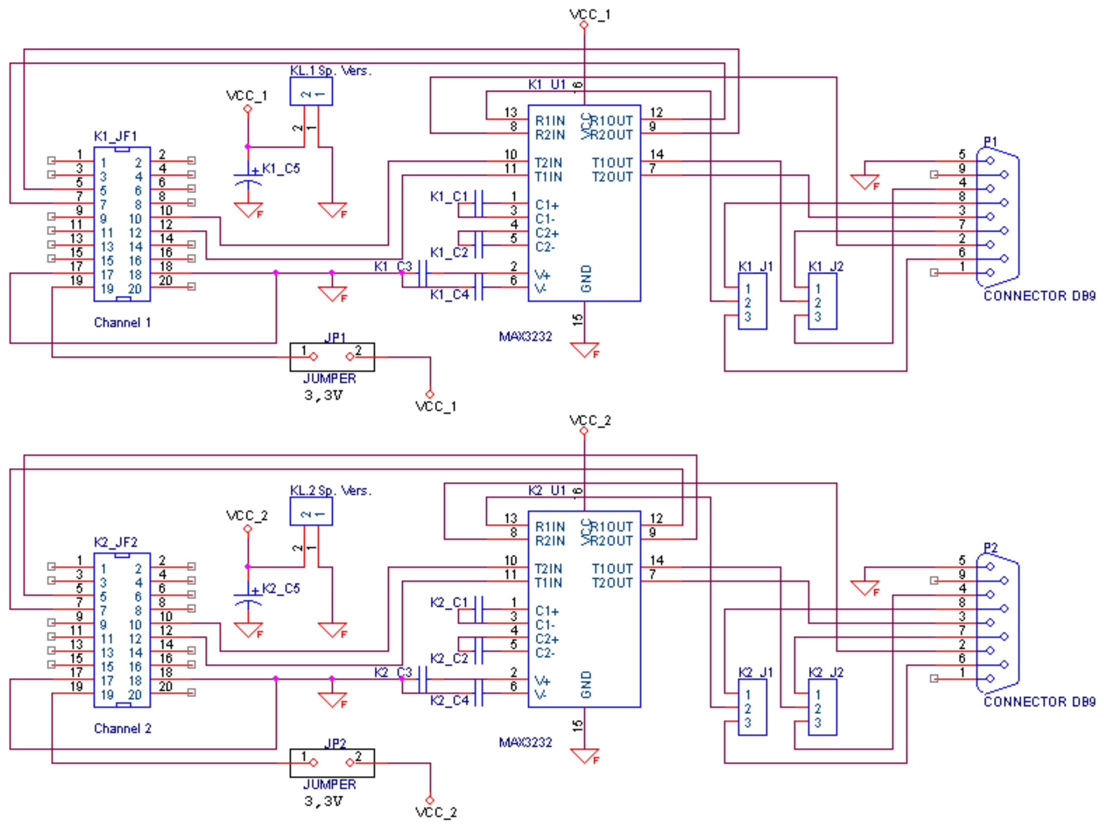


Abbildung 4.16: Schaltung der Pegelumsetzer-Karte für V24

Um zu belegen, dass mit der so generierten Schnittstelle echtzeitfähig Daten transferiert werden können, ist der Datenverkehr mit Hilfe eines Speicheroszilloskops aufgezeichnet worden. Der zeitliche Verlauf des Signals wird auf diese Weise in den Diagrammen sichtbar gemacht, die mit steigender Auflösung den Sendevorgang von Bewegungsdaten für den Roboter aufzeigen. Im Rahmen der Genauigkeit der Messkarte (hier eine Mikrosekunde) und der begrenzten Auflösung des zugehörigen DOS-Programms sind Messungen im Diagramm möglich aber Messfehler nicht auszuschließen.

In Abbildung 4.17 ist eine längere Sendesequenz über vier Pakete dargestellt. Die gemessene Frequenz, die unten links im Diagramm zu entnehmen ist, ist der Kehrwert der gemessenen Zeitdifferenz. Sie ist ein Maß für verstrichene Zeit. Im dargestellten Signal ist der Sendebeginn zu erkennen: Die erste steigende Flanke gehört zum Startbit des

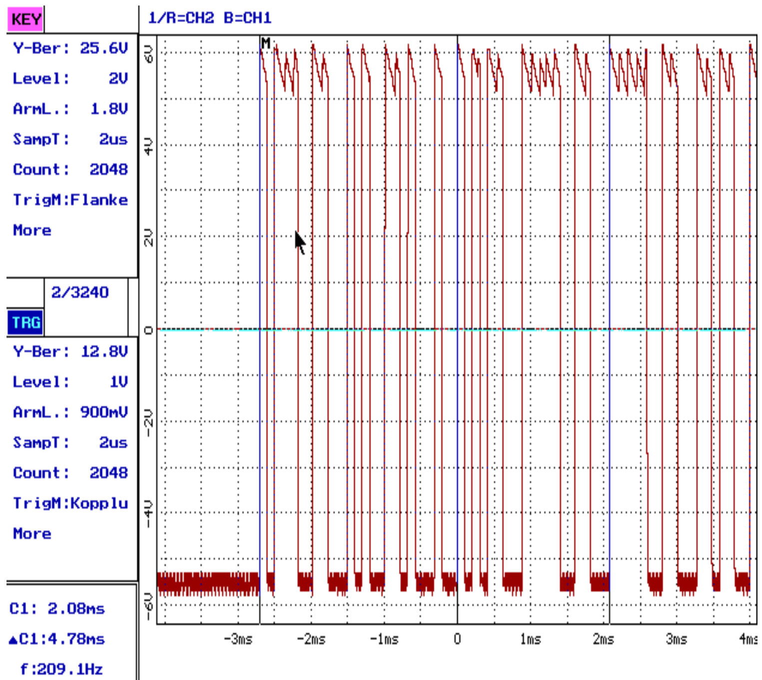


Abbildung 4.17: V24-Protokoll: Sendesequenz

ersten Pakets.

Abbildung 4.18 zeigt ein einzelnes Paket des Sendevorgangs in einer größeren Auflösung. In dem Diagramm sind die einzelnen Bit gut zu erkennen. Die gemessene Zeit beschreibt die Zeitdauer der Übertragung eines Paketes. In solchen Paketen werden die Befehlssequenzen zum Roboter verschickt. Das hier gezeigte Paket lautet in digitaler Form *0 1 0 0 0 1 1 0 0 1 1*. Die führende Null ist das Startbit, die beiden Einsen am Ende symbolisieren die Stopbits. In der Mitte verbleiben weitere acht Bit, die Nutzdaten. Das LSB ist links und das MSB auf der rechten Seite vorzufinden. Interpretiert man dieses Byte als ASCII-Zeichen, so erhält man die „1“ als Ergebnis. Die Bedeutung wird im Kapitel 5 in der Tabelle 5.1 erklärt.

Um die Sendedauer eines einzelnen Bit exakt bestimmen zu können, misst man die Zeit für eine bestimmte Anzahl an Bits und dividiert durch deren Anzahl. Auf diese Weise besteht die Möglichkeit, das unerwünschte Nebeneffekte das Ergebnis verfälschen. Deshalb ist in Abbildung 4.19 zur Kontrolle nur ein Bit vermessen worden. Die so ermittelte Frequenz von 9620 Hz weicht um 20 Hz vom Sollwert 9600 Hz ab. Wie genau die Messung an dieser Stelle wirklich ist, lässt sich nur schwer sagen, da die Auflösung des Messprogramms bereits voll ausgeschöpft war.

Zwischen 2 Paketen treten kurze Verzögerungszeiten (Gaps) auf. Diese Zeitspannen repräsentieren die Laufzeiten des Handshakes und der Hardware. Um diese Latenzzeit zu

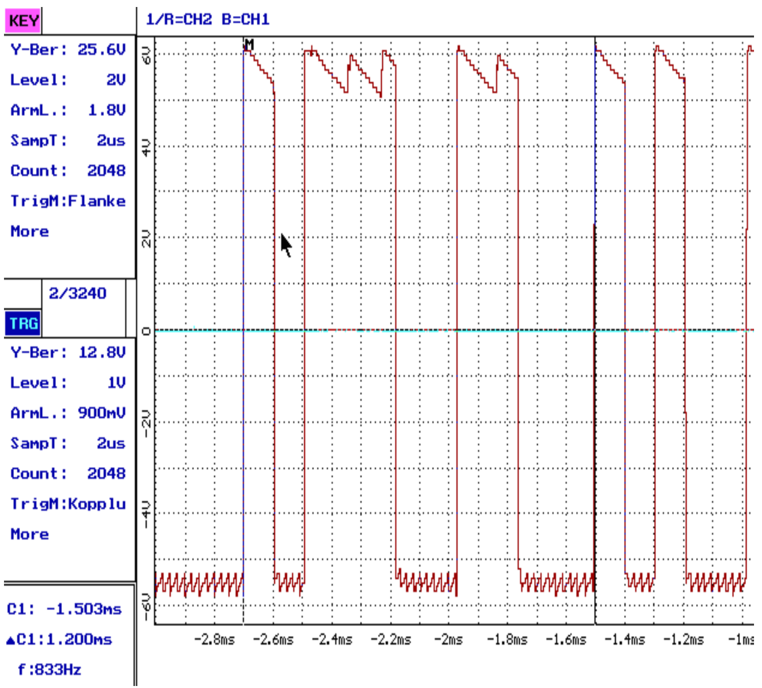


Abbildung 4.18: V24-Protokoll: ein Datenpaket

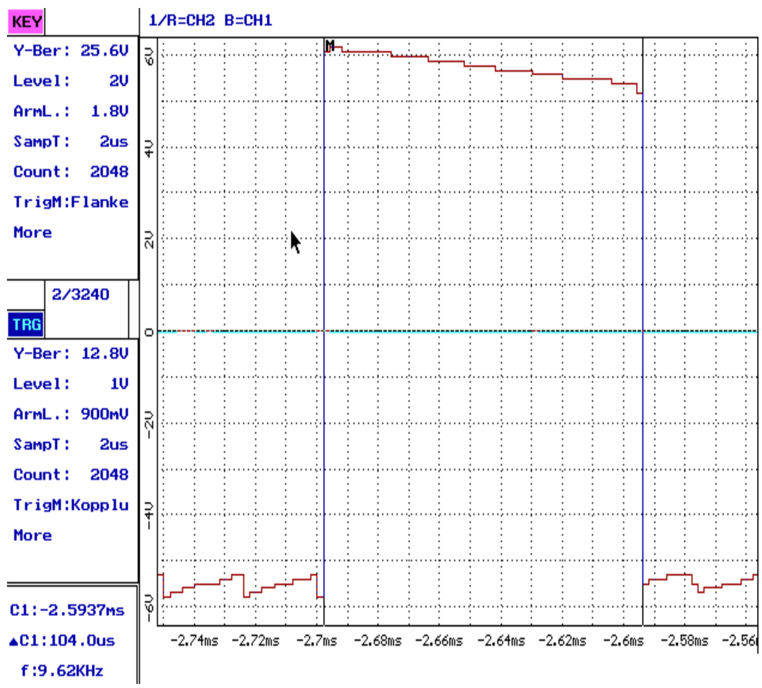


Abbildung 4.19: V24-Protokoll: ein einzelnes Bit

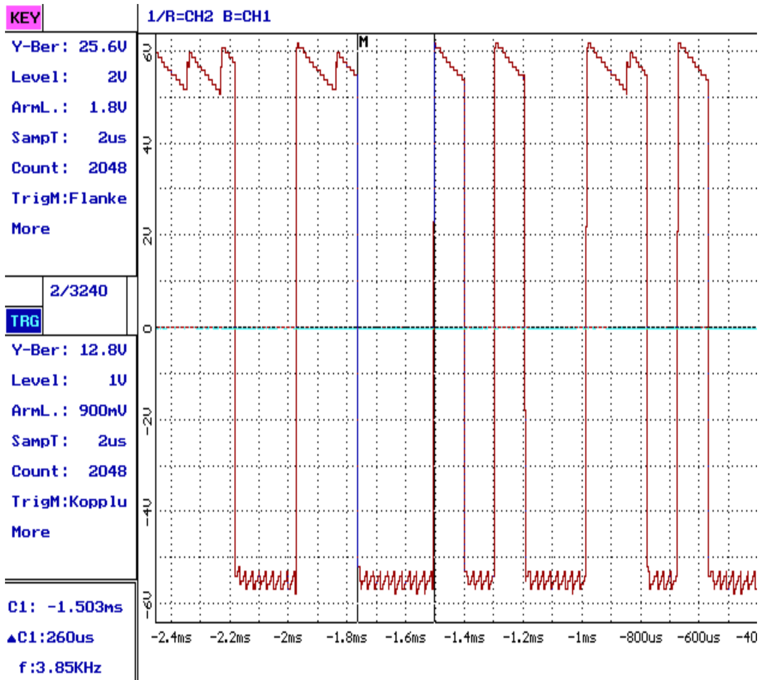


Abbildung 4.20: V24-Protokoll: Verzögerungszeit zwischen zwei Paketen

ermitteln, ist in Abbildung 4.20 der Zeitraum über zwei Stopbits und das Gap vermessen. Wird für die Übertragung eines Bits die theoretische Zeit von $T = 1/9600s$ angesetzt, so verbleibt eine Zeit von 51 μs für die Latenz. Eine Hochrechnung mit 1/9600 s pro Bit und 51 μs für das Handshake bestätigt sich in Abbildung 4.17. Der berechnete Wert ergibt sich zu 208,9 Hz und der gemessene beträgt 209,1 Hz. Das entspricht einer Abweichung von 0,1%, die durchaus von Messungenauigkeiten (Ablesefehler, Messverfahren) herrühren kann.

4.5 Ergebnisse

In Kapitel 4 wurde das *Entwurfskonzept*, ein thematischer Schwerpunkt dieser Arbeit behandelt. Anhand von zwei Anwendungsfällen, der Schnittstelle *Task und Kanal* und der *Adaption einer Task*, wurde der Anwendungsbereich des Entwurfskonzepts vorgestellt. Das Entwurfskonzept definiert zunächst einen *Interface-Block (IFB)* und gliedert diesen in drei wesentliche Komponenten:

- die Kontrolleinheit (Control Unit),
- den Sequenzgenerator (Sequence Generator)
- und den Protokollgenerator (Protocol Generator)

Der Sequenzgenerator, wie auch der Protokollgenerator, werden als *Generatorstufen* bezeichnet. Zusammen mit der Kontrolleinheit stellen die Generatorstufen die notwendige Funktionalität bereit, um unterschiedliche Typen von Schnittstellen systematisch zu implementieren. Das hier entwickelte Entwurfskonzept bietet die Möglichkeit der Erweiterbarkeit und Adaptierbarkeit. Die Darstellung des Entwurfskonzepts erfolgte mit Hilfe der in Kapitel 3 beschriebenen UML-Modellierung. Daraus ergeben sich auch die Grundlagen für eine Codegenerierung im Zusammenhang mit dem automatisierten Entwurf.

Im zweiten Abschnitt dieses Kapitels wurde das Entwurfskonzept erfolgreich auf die V24-Schnittstelle angewendet und hat sich in allen geforderten Aspekten bewährt. Das Beispiel verdeutlicht, dass das Entwurfskonzept ausgehend von der Spezifikation der seriellen Schnittstelle zu einer Implementierung führt. Die Modellierung durch UML-Diagramme hat sich somit auch im praktischen Einsatz bestätigt. Die im Rahmen des Demonstrators implementierte V24-Schnittstelle dient zur Steuerung der Roboter und wurde einer Echtzeitmessung unterzogen. Die Diagramme bestätigen im Rahmen der Messgenauigkeit die Einhaltung der Zeitrestriktionen. Die als Beispiel behandelte serielle Schnittstelle ist Bestandteil des komplexen Gesamtdemonstrators. Dieser wird im folgenden Kapitel ausführlich erläutert.

5 Demonstrator

Zur Illustration und Validierung der entwickelten Konzepte wurde ein Demonstrator definiert und realisiert. Dessen Aufbau und Funktionsweise wird zunächst ausführlich beschrieben. Im darauf folgenden Abschnitt sind die erarbeiteten Konzepte zum Entwurf von Echtzeitschnittstellen am Demonstrator angewandt worden und verdeutlichen damit die Anwendbarkeit und die Praxisrelevanz der Ergebnisse dieser Arbeit.

5.1 Bedeutung einer Fallstudie

Eine Fallstudie kombiniert die Vorteile eines praktischen Entwurfs mit denen einer rein theoretischen Betrachtung. Sie dient der Veranschaulichung der theoretischen Ideen am Demonstrator und verleiht ihnen praktische Relevanz. Weiterhin liefert sie Hinweise auf Schwachstellen des theoretischen Modells bezüglich Struktur und Umsetzung. Ein Beispiel dafür könnte der Entwurf eines Modells in VHDL sein, der in der Simulation durchaus korrekte Werte liefert, doch nach der Synthese nicht das gewünschte Verhalten aufweist. Dies gibt einen Hinweis darauf, dass das Modell noch unzureichend ist, z. B. Simulationszeiten vernachlässigt wurden, oder dass die praktische Umsetzung dieses Modells in dieser Form nicht fehlerfrei ist. Auf der anderen Seite ist ein praktischer Entwurf ohne grundlegendes theoretisches Modell häufig oberflächlich und kann nur sehr beschränkte Komplexität annehmen. Oft sind Aufgabenstellungen einfach durch mathematische Modelle beschreibbar, berechenbar und beweisbar. Bezogen auf den Demonstrator, kann man z. B. das Problem der Kollisionsvermeidung der interagierenden Roboter durch ein Vektor- oder Voxelmodell darstellen und berechnen.

5.2 Gesamtszenario

Der Demonstrator stellt ein Szenario dar (vgl. Abbildung 5.1), in dem zwei Roboter (R1 und R2) in einen Arbeitsbereich integriert sind. Roboter R1 wird manuell gesteuert, Roboter R2 dagegen fährt eine Standard-Routine ab, (vgl. Abbildung 5.2). Dazu wird ein digitaler Joystick als Steuereinheit verwendet. R2 soll eine untergeordnete Priorität haben und muss daher den Bewegungsablauf von R1 erkennen und eine Kollisionsvermeidung unter Anwendung einer bestimmten Ausweichstrategie durchführen. Unter dem Blickwinkel der harten Echtzeit betrachtet, macht dies eine Echtzeitkommunikation zwischen den Robotern erforderlich. Zusätzlich sind noch 2 Drehteller und ein Förderband als Zusatzhardware in der Szene angeordnet, deren Verhalten vorgegeben ist.

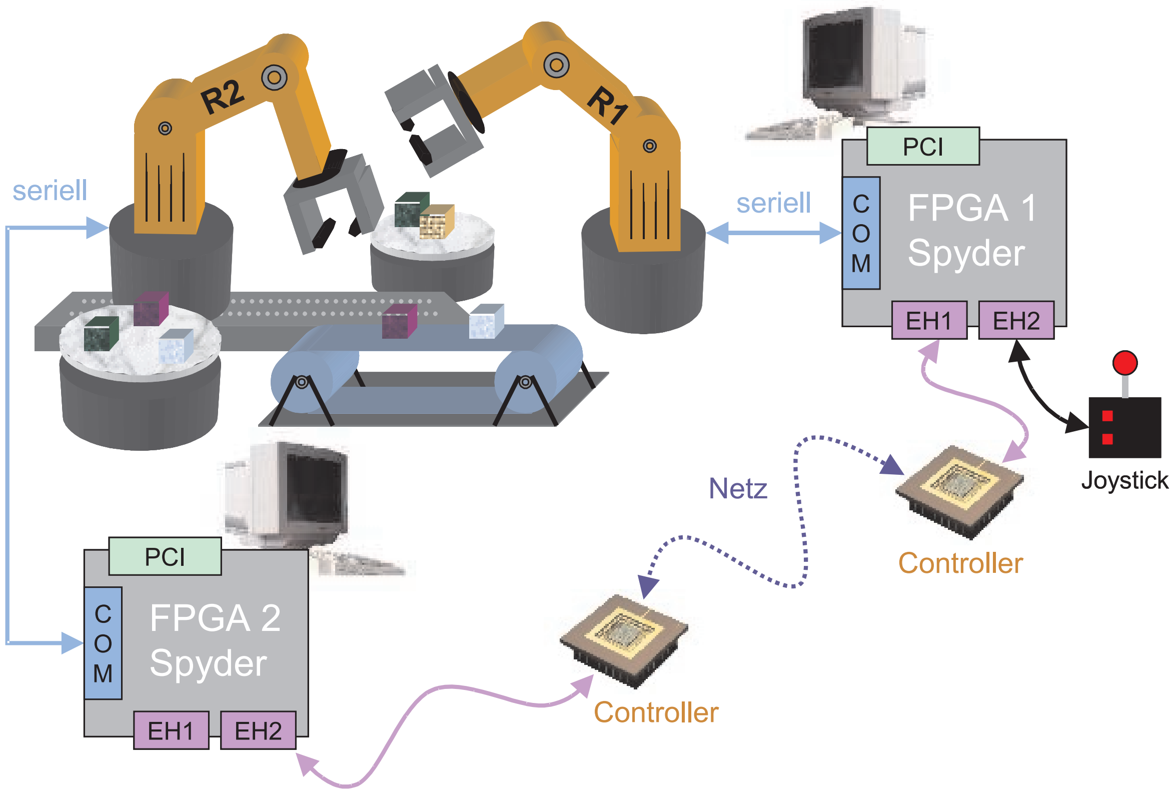


Abbildung 5.1: Gesamtszenario

Um den Demonstrator mit Echtzeitanforderungen in Verbindung zu bringen, kann man sich das Szenario in einen sicherheitskritischen, vielleicht sogar lebensfeindlichen Bereich verlagert vorstellen, z. B. ein Atomkraftwerk. Dort führt eine Maschine routinemäßig einprogrammierte Aufgaben aus, was dem Verhalten von Roboter R2 entspricht. Da der Roboter keine Sensoren zur Erkennung von Hindernissen hat, ist ein statisches Umfeld die Voraussetzung für den sicheren Einsatz von R2. Um es an dieser Stelle einfach zu halten, hat der Programmierer von R2 eine exakte Karte der Umwelt vorliegen und wird das Programm von R2 so gestalten, dass Kollisionen mit allen Fixkörpern der Umwelt vermieden werden. D. h. R2 wird weder das Förderband, den Drehteller, den Tisch auf dem er steht, noch sich selber tangieren.

Geschieht dann etwas unvorhergesehenes, so muss ein Mensch flexibel eingreifen. Der Aufenthalt im Arbeitsbereich des Roboters R2 ist für einen Menschen zu gefährlich und darf ohne dessen Deaktivierung nicht betreten werden. Deshalb ist ein manuell zu bedienender Roboter R1 installiert worden. Da R2 weiterhin seinen Job ausführt, die Roboter aber keine Sensoren haben, kann es zu ungewollten Zusammenstößen mit R1, der unter den beiden Robotern den Vorrang hat, kommen. Anhand der Bewegungsdaten, die sich aus der Motoransteuerung ergeben, muss auf die räumliche Lage der Roboter geschlossen werden, was durch eine Positionsberechnung erfolgt. Der Algorithmus zur

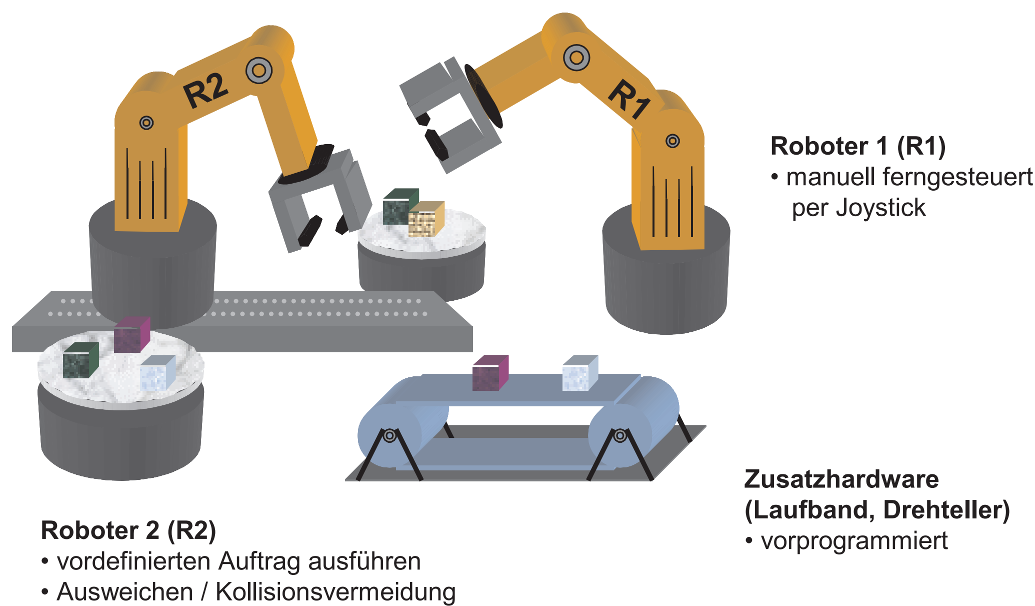


Abbildung 5.2: Aufgaben der Roboter

Kollisionsvermeidung nutzt diese Werte, um die Ausweichdaten für R2 zu errechnen. Eine Voraussetzung, um eine Kollision sicher zu vermeiden besteht darin, dass die Antwortzeiten klein genug sind, damit die Ausweichdaten für R2 rechtzeitig zur Verfügung stehen. Das impliziert eine ausreichende und vorhersagbare Datenübertragung für alle Tasks, Kanäle und Schnittstellen.

5.3 Kollisionsvermeidung

Eine Kollisionsvermeidung setzt sich im wesentlichen aus zwei Komponenten zusammen: Der Erkennung einer vorauszusehenden Kollision und der darauf aufbauenden Ausweichstrategie. Das Ziel der Kollisionsvermeidung ist es, jegliche Berührung der beiden Roboter zu unterbinden. Die „Qualität“ dieses Vorgangs kann in unterschiedliche Stufen eingeteilt werden. Einfache Algorithmen benötigen nur wenige Positionsdaten, arbeiten dafür relativ unflexibel und erlauben nur eine grobe Annäherung der beiden Roboter. Bessere Verfahren berechnen die exakte Position der beiden Roboter mit Hilfe der Vektorrechnung oder ggf. einer Voxeldarstellung (vgl. Abbildung 5.3). Damit lassen sich hochauflösende Modelle erstellen, auf deren Grundlage die Bewegungsdaten der Kollisionsvermeidung bestimmt werden können. Zu beachten ist, dass solche Verfahren größere Rechenzeiten und Systemresourcen benötigen. Die darauf aufsetzende Ausweichstrategie legt fest, nach welchem Verfahren R2 sich von R1 fortbewegt, wenn eine Kollision bevorsteht, um gleichzeitig noch den vorgegebenen Auftrag weiter auszuführen.

Ein weiterer Punkt der hier relevant ist, bezeichnet die „Kollision eines Roboters mit

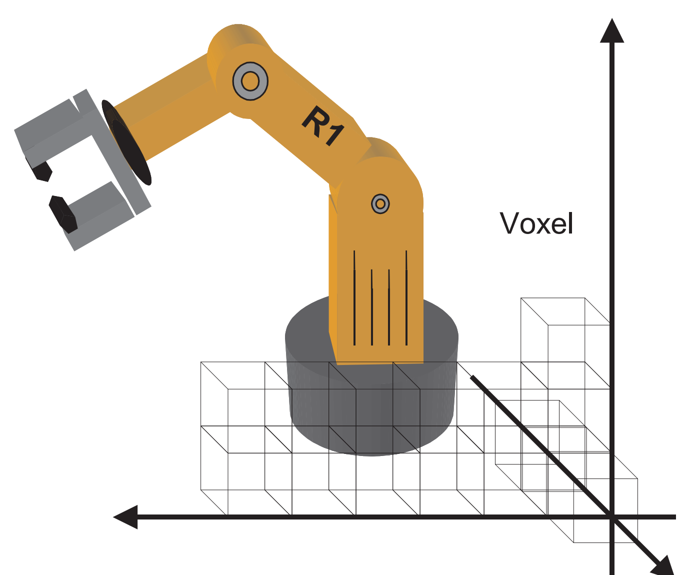


Abbildung 5.3: Voxelmodell zur Kollisionsvermeidung

sich selbst“. Zu einer solchen Selbstkollision kann es durch den Bediener von R1 oder einer fehlerhaften Programmierung von R2 kommen. Das ist schnell passiert, falls der Roboter bestimmte Punkte im Raum anfahren soll, die im Einzelnen erlaubt sind, allerdings ungültige Fahrwege des Roboters benutzt werden. Um das Problem zu lösen, kann eine Routine für den Roboter erstellt werden, die alle Bewegungen mitprotokolliert und und auf vorgegebene Schranken hin überwacht. Falls ein Roboterarm seine Grenzwerte überschreitet, sorgt die Selbstkollisionsvermeidung dafür, dass die Bewegung solange modifiziert wird, bis keine Kollision mehr bevorsteht. Um dabei *Deadlocks* auszuschließen, müssen gegebenenfalls auch andere Roboterglieder, die nicht direkt von der Selbstkollision betroffen sind, zwischenzeitlich bewegt werden.

5.4 HW-Aufbau des Demonstrators

Um ein eingehendes Verständnis für den Demonstrator zu vermitteln, werden zunächst die charakteristischen Merkmale der Roboter erläutert. Wie bereits erwähnt, besitzen die Roboter keine Sensoren. Durch die Gelenke verfügt jeder Roboter über sechs Freiheitsgrade, wobei R2 noch zusätzlich über die *Slidebase* bewegt werden kann. Die Robotermotoren sind Gleichstrommotoren, die von der *Steuereinheit* die entsprechenden Encoderschritte übermittelt bekommen. Dazu sind die Motoren mit Encodern ausgestattet, die eine schrittgenaue Ansteuerung ermöglichen. In der Steuereinheit befindet sich ein Mikrocontroller, der die Verwaltung der Encoderschritte übernimmt. Die Kommunikation zwischen Rechner und Steuereinheit erfolgt durch ein handshake-freies V24-

Protokoll. Es werden ASCII-Zeichen zur Codierung der Befehlssequenzen benutzt. Ein Auszug der in dieser Arbeit benötigten Befehle ist in Abb.5.1 gegeben. Einzelheiten des mechanischen Aufbaus sind in [Rob82, Mag92] beschrieben.

Tabelle 5.1: Befehlssequenzen des Roboters

ASCII-Sequenz	Bedeutung
aM± xxx<cr><lf>	Befehl: Fahre Motor *a* um *xxx* Schritte
B	Befehl: Alle Motoren anhalten
T	Abfrage: Sind alle Motoren fertig? (Rest-ECS=0)
aQ	Abfrage: Rest-Encoderschritte für Motor *a* ausgeben
aL	Abfrage: Ist Mikroschalter von Motor *a* gedrückt?

Weiterhin sind im Demonstrator zwei FPGAs enthalten, die als Rapid-Prototyping-Plattformen dienen, um die synthetisierten HW-Komponenten zu realisieren. Die FPGAs sind die zentralen Knotenpunkte des Systems, über die die gesamte Kommunikation verläuft. Zum Einsatz kommen zwei identische Spyder-Boards bestückt mit FPGAs der Firma Xilinx, mit einer I / O-Funktionalität von zwei Erweiterungssteckern (Extension-header) mit je 192 Pins und einer PCI-Bus Schnittstelle. Die weiteren Betrachtungen konzentrieren sich auf die im System enthaltenen Schnittstellen und deren Funktionalität, die im Abschnitt 5.5 ausführlich untersucht werden.

5.5 Echtzeitschnittstellen im Demonstrator

An dieser Stelle werden die im Demonstrator enthaltenen Schnittstellen vorgestellt und analysiert. Da der Demonstrator als HW / SW-Implementierung auf Basis einer FPGA-Prototyping Plattform realisiert ist, wird zur Ansteuerung von R1 wird FPGA1 eingesetzt, entsprechend wird R2 von FPGA2 gesteuert. Die Kollisionsvermeidung wird als Software auf einem Host-Prozessor ausgeführt und setzt den echtzeitfähigen Austausch von Positionsdaten der Roboter zwischen den FPGAs voraus. Daraus resultieren im Demonstrator vier echtzeitfähige Schnittstellentypen:

- FPGA1 – Joystick: HW-Schnittstelle
- FPGA1/2 – Roboter1/2: serielle Schnittstelle (V24)
- FPGA2 – Hostrechner: PCI-Bus
- FPGA1 – FPGA2: Echtzeitkanal für TTP

In den nachfolgenden Teilkapiteln wird zunächst die Bedeutung der vorgestellten Schnittstellen innerhalb des Demonstrators veranschaulicht. Die Implementierung und ihre interne Realisierung unter dem Gesichtspunkt des Entwurfskonzepts ist in Abschnitt 5.6 näher beschrieben.

5.5.1 FPGA1 – Joystick (Joystickschnittstelle)

Ein Joystick ist ein manuelles Eingabegerät, mit dem gleichzeitig mehrere Freiheitsgrade gesteuert werden können. Es gibt prinzipiell zwei verschiedene Typen von Joysticks. Zum einen die digitialen und zum anderen die analogen Geräte. Wie der Name bereits impliziert kann ein digitialer Joystick lediglich ein / aus-Informationen liefern, ein analoger Joystick dagegen hat einen beinahe kontinuierlichen Übergang von 0 zu 1. Im Demonstrator wird ein digitaler Joystick verwendet, da er direkt an das FPGA, welches nur digitale Daten verarbeiten kann, angeschlossen werden kann.

Analoger Joystick

Zuerst ging die Überlegung, einen Joystick in den Demonstrator zu integrieren dahin, einen analogen Joystick zu verwenden. Das hätte dem Anwender großen Komfort geboten und ermöglicht, den Roboter mit verschiedenen Geschwindigkeiten zu bewegen. Der Aufbau eines analogen Joysticks (vgl. Abbildung 5.4) ist dagegen aufwendiger als der eines digitalen (vgl. Abbildung 5.6).

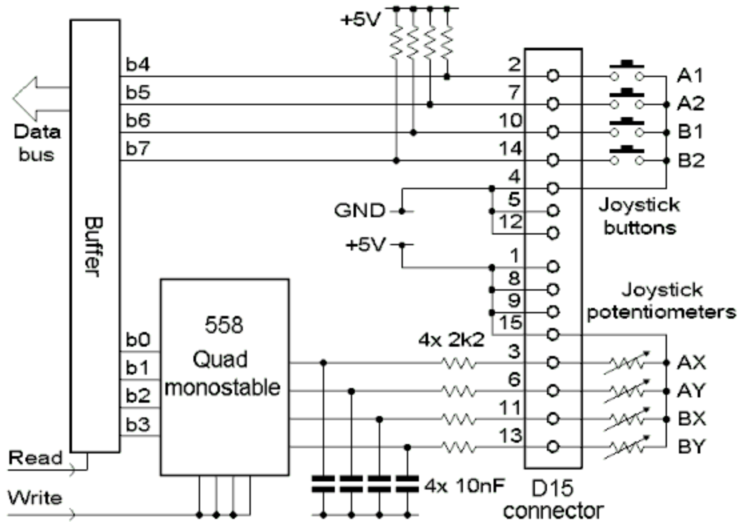


Abbildung 5.4: Aufbau eines analogen Joysticks [Eng98]

Die Schnittstelle des Joysticks (vgl. Abbildung 5.5) ist hier angegeben für zwei analog gesteuerte Freiheitsgrade, X1 und Y1, und zwei digitale Knöpfe, Switch1 und Switch2. RX und RY sind Drehpotentiometer (0...100KΩ), die an die X- bzw. Y-Achse des Joysticks gekoppelt sind und so einen stufenlosen Wertebereich für X und Y ermöglichen. Da ein PC nur digitale Daten verarbeiten kann, wird ein Zählverfahren zur A / D (Analog / Digital) -Umsetzung angewendet. Dabei wird ein Zähler (Baustein: 558 Quad monostable) gestartet, der die Zeit misst, bis sich die vorgeschalteten Kondensatoren auf einen festgelegten Wert aufgeladen haben. Je nach aktueller Einstellung des Potentiometers RX bzw. RY erfolgt dies langsamer oder schneller, so dass den Ladezeiten direkt

eine Joystickstellung zugeordnet werden kann. Die so ermittelten Werte sind quantisiert und nicht mehr wertekontinuierlich, erreichen aber bzgl. der menschlichen Motorik eine ausreichende Auflösung.

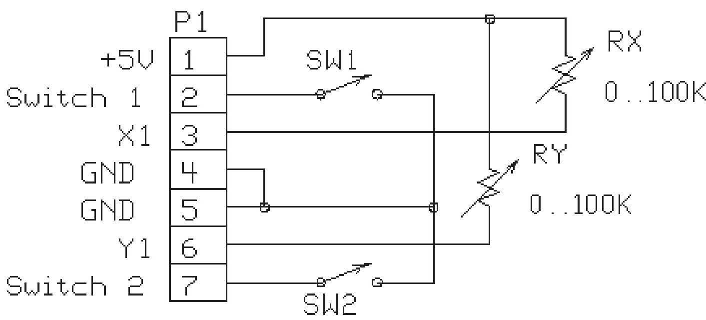


Abbildung 5.5: Schnittstelle eines analogen Joysticks [Eng98]

Die Joystick-Schnittstelle befindet sich normalerweise im Gameport des PCs. Da der Joystick direkt an das FPGA angeschlossen werden sollte, hätte man eine A / D-Umsetzerkarte mit ähnlicher Funktionalität, wie in Abbildung 5.4 beschrieben, entwerfen und vorschalten müssen.

Digitaler Joystick

Ein digitaler Joystick beinhaltet nur diskrete Schalter, die entweder geöffnet oder geschlossen sind. Der Joystick dieses Demonstrators ist eine Spezialanfertigung und verfügt über drei Freiheitsgrade, die X-, Y-, Z-Achse, denen jeweils zwei Richtungen zugeordnet sind:

- X: links ↔ rechts
- Y: vor ↔ zurück
- Z: auf ↔ ab

Der Schaltplan diese Joysticks ist in Abbildung 5.6 dargestellt. Ein solcher Aufbau ermöglicht das Steuern von maximal drei Motoren zur gleichen Zeit. Aufgrund der Taster ist jedoch nur eine einheitliche Fahrgeschwindigkeit für jeden Arm des Roboters möglich. Zur Verarbeitung der Joystickdaten wird eine Task im FPGA eingesetzt, die die für den Roboter benötigten Werte aus den Signalleitungen zusammenstellt. Anschließend werden die so erzeugten Werte an den Sequenzgenerator weitergereicht und über die serielle Schnittstelle an den Roboter übertragen.

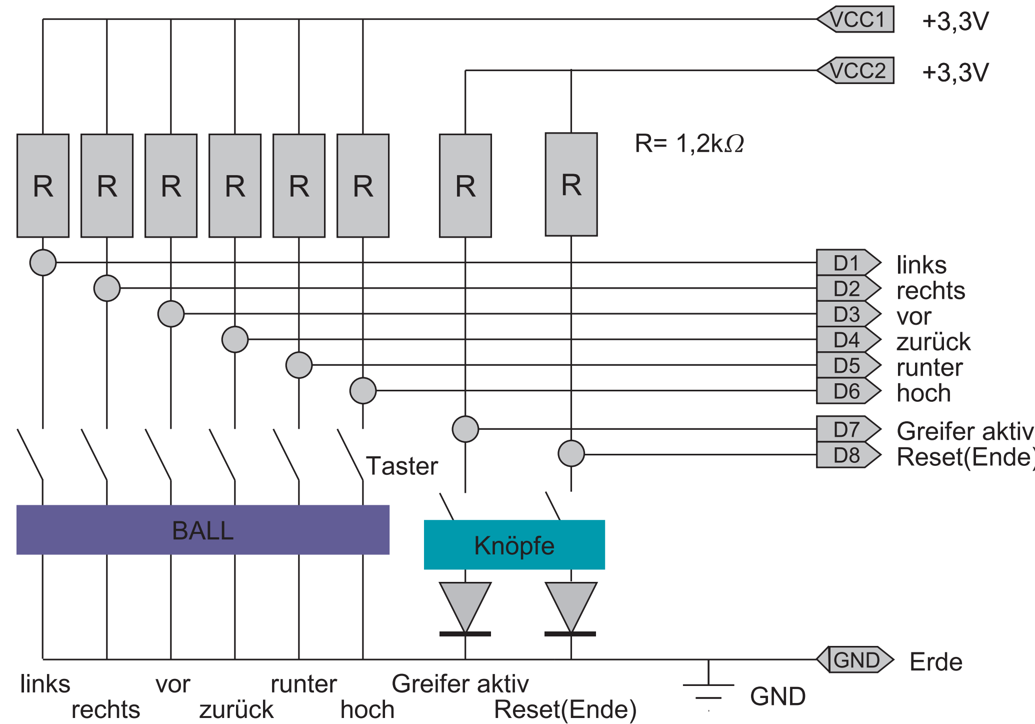


Abbildung 5.6: Aufbau eines digitalen Joysticks

5.5.2 FPGA1/2 – Roboter1/2 (Serielle Schnittstelle)

Nach der Beschreibung der Implementierung der V24-Schnittstelle im Kapitel 4.4 kann nun auch die Verwendung der beschriebenen Komponenten innerhalb des Demonstrators angesprochen werden. Im Demonstrator wird die serielle Schnittstelle in zwei Anwendungen eingesetzt:

- Kommunikation: FPGA (Rechner) ↔ Steuereinheit (Roboter)
- Komponente im TTP/A: FPGA ↔ FPGA

Die logische und die implementierte Verbindung zwischen FPGA und Steuereinheit ist in Abbildung 5.7 illustriert. Auf diese Weise werden die Steuerdaten an den Roboter übermittelt und Statusmeldungen der Motoren und Schalter zurückgeleitet. Die im FPGA aufbereiteten Joystickdaten werden dazu von IFB auf den Extensionheader gegeben und über den Pegelumsetzer an die Steuereinheit weitergeleitet. Die Umsetzerkarte ist in Abbildung 5.7 nicht vorhanden, da sie funktional ein Bestandteil des Protokollgenerators im IFB ist (vgl. Kapitel 4.4).

Auf die Funktion der seriellen Schnittstelle als Komponente von TTP/A wird im Abschnitt 4.4 (Implementierung von TTP) näher eingegangen.

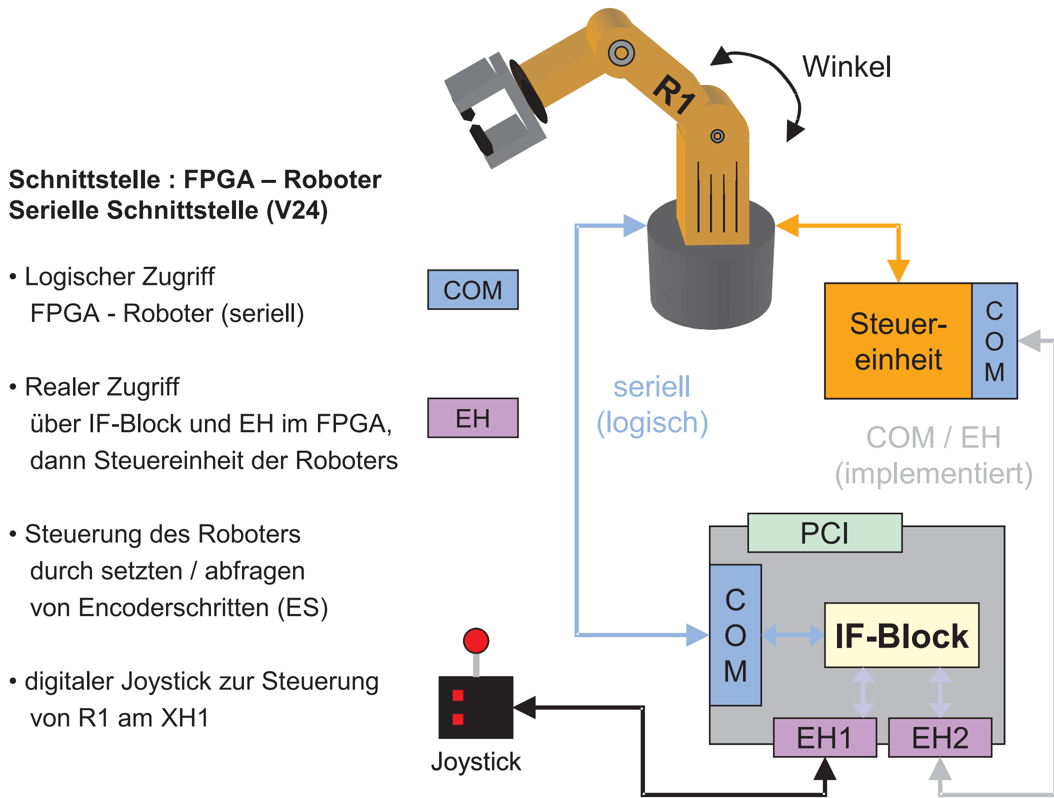


Abbildung 5.7: Kommunikation: FPGA-Roboter

5.5.3 FPGA2 – Hostrechner (PCI-Schnittstelle)

Eine Ausnahme zum reinen HW-Entwurf in dieser Arbeit bildet die Kollisionsvermeidung der Roboter, die allerdings nur theoretisch betrachtet wird und nicht implementiert wurde. Die Realisierung eines Algorithmus zur Kollisionsvermeidung in HW setzt im Allgemeinen besondere HW-Komponenten, wie z. B. Floating-Point-Einheiten voraus, die im FPGA nicht vorhanden sind. Deshalb erscheint eine SW-Lösung sinnvoll, die eine weitere HW / SW-Schnittstelle in das Design integriert. Der Algorithmus wird auf der Host-CPU des Rechners, in den FPGA2 eingesetzt ist, ausgeführt. Als Schnittstelle bietet sich der PCI-Bus des Rechners an, da das FPGA über einen solchen Anschluss verfügt. Die Kommunikation zwischen dem Rechner und dem FPGA über den PCI-BUS wird in Abbildung 5.8 veranschaulicht.

Gemäß Abbildung 5.3 sind zwei alternative Möglichkeiten für eine Kollisionsvermeidung möglich: Entweder werden die Motordaten von R1, der manuell gesteuert wird, schon auf dem FPGA1 in Positionsdaten umgerechnet und diese übertragen, oder man beschränkt sich darauf, direkt die Motordaten (Encoderschritte) zu transferieren. Variante Eins nutzt den Vorteil der parallelen Datenverarbeitung und verteilt die Rechenlast auf unterschiedliche HW. Die so berechneten Positionsdaten des Roboters sind jedoch wesentlich größer als die reine Information der Motorstellung. Diesen Vorteil nutzt die zwei-

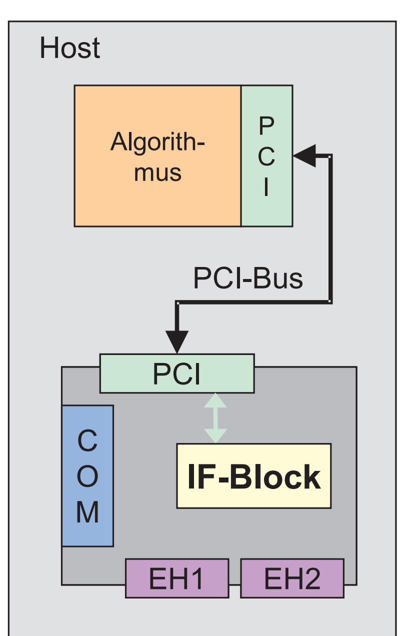


Abbildung 5.8: Kommunikation: FPGA-Host

te Variante, die zur Übertragung der Encoderschritte eine geringere Bandbreite benötigt, dafür eine leistungsstarke Recheneinheit erfordert, die unter Einhaltung der Zeitrestriktionen beide Roboter auswertet. Die Kollisionsvermeidung für das FPGA2, wie sie im zweiten Fall beschrieben ist, wird in Abbildung 5.9 illustriert. *Observer* protokollieren die Motorbewegungen mit und aktualisieren die entsprechenden Werte im DPRAM. Die Motordaten von R1 müssen dabei zuerst mit Hilfe von TTP/A in das FPGA2 transferiert werden (vgl. Abbildung 5.12). Der Algorithmus im Host greift regelmäßig auf diese Daten zu (*Polling*) und berechnet die Ausweichdaten für R2, falls eine Kollision bevorsteht. Eine Kommunikation über den PCI-Bus muss immer von der Host-CPU initiiert werden, da dem FPGA die Berechtigung fehlt, Anfragen an den PCI-Bus zu richten.

Eine Realisierung der Kollisionsvermeidung, die über den PCI-Bus mit der Host-CPU kommuniziert, bringt jedoch ein besonderes Problem mit sich: Der Zugriff über den PCI-Bus ist nur in Computern mit einem echtzeitfähigen Betriebssystem (RTOS – *R*eal-*T*ime *O*perating *S*ystem) als deterministisch zu betrachten. Ein Beispiel hierfür wäre *RT-Linux*. In der Literatur findet man unter anderem folgende Bücher [BY96, Epp97, Yod97, Rub98] und Internetseiten [RtL01, Wil01] zum Thema RT-Linux. Nur so kann gewährleistet werden, dass die SW-Tasks auch deterministisch ausgeführt werden und entsprechenden PCI-Bus-Zugriff erhalten.

5.5.4 FPGA1 – FPGA2 (TTP)

Um die Positionsdaten des Roboters R1 für die Kollisionsvermeidung immer rechtzeitig in FPGA2 zur Verfügung zu haben, ist zwischen den beiden FPGAs eine Echtzeitkom-

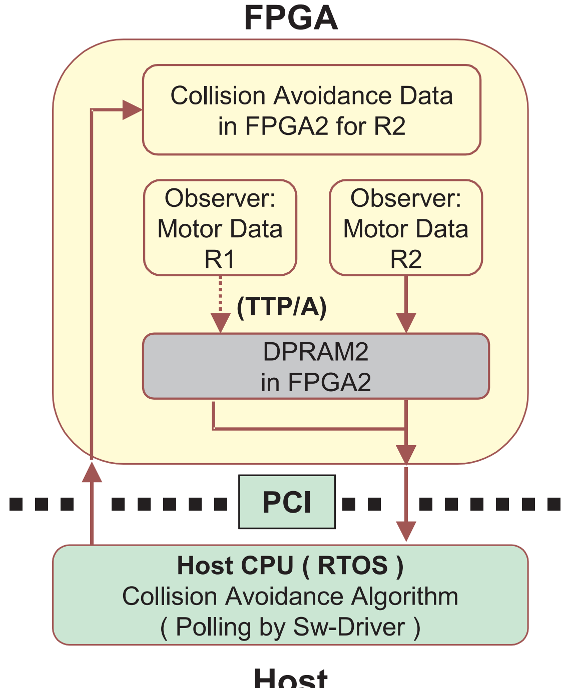


Abbildung 5.9: Die PCI-Schnittstelle: FPGA-Host

munikation notwendig (vgl. Abbildung 5.10). Das ausgewählte Verfahren beruht auf den Time Triggered Protocols (TTP) und ist eine für dieses Problem angepasste Variante des TTP/A.

Die wichtigen Elemente der Übertragung von Echtzeitdaten mit TTP sind in Abbildung 5.10 dargestellt. Hervorzuheben sind der TTP-kompatible Kanal mit einem TDMA-Zugriffsverfahren und die jeweiligen TTP-Bus-Controller, die Bestandteil des IFB sind (vgl. Abbildung 5.10 und 5.11). Es ist möglich das Kabel direkt an das FPGA anzuschließen, weil die Kommunikation mit dem Extensionheader (EH) direkt vom IFB verwaltet wird. Da keine weiteren HW-Komponenten wie z. B. externe Bus-Controller in dieses Schema einbezogen sind, kann das vorgestellte Modellierungskonzept vollständig und hinreichend angewendet werden. Weiterhin werden die Berechnungen der Ablaufplanung des Systems (vgl. Kapitel 5.6.3) auf diese Weise vereinfacht, da nur noch Laufzeiten des Kanals und der internen Struktur des FPGAs berücksichtigt werden müssen. Eine aufwendige Recherche von Daten über Zusatzhardware entfällt so vollständig.

TTP

Für die Kommunikation zwischen den FPGAs wird eine modifizierte Variante von TTP/A [Kop01]) eingesetzt. Die zu übertragenden Signale sind die Motordaten von Roboter1, zu deren Gunsten hier die Entscheidung ausgefallen ist (vgl. Kapitel 5.3). Wie

Schnittstelle : FPGA – FPGA

- Kommunikation über ein Echtzeitprotokoll (TTP)
- Harte Echtzeit (Determinismus)
- Controller in IFB implementiert
- Echtzeitkanal mit TDMA

Task 1
TTP (IFB)
Controller

Task 2
TTP (IFB)
Controller

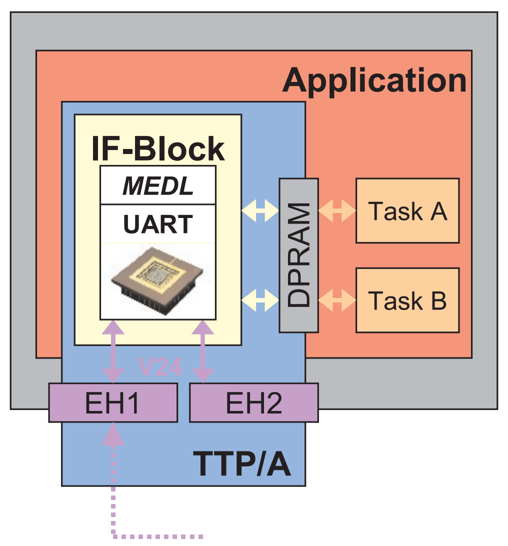
Medium
TTP-Kanal

PCI
COM
IF-Block
Task 1
EH1
EH2

TTP-Kanal

PCI
COM
IF-Block
Task 2
EH1
EH2

Abbildung 5.10: Kommunikation: FPGA-FPGA



Application

- alle Tasks mit genau einem IF-Block (IFB)

Einsatz von TTP

- **T**ime **T**riggered **P**rotocol
- Fehlertolerantes, echtzeitfähiges Protokoll auf Basis der TT-Architecture (TTA)
- Kommunikation mit den Tasks
- Datentransfer: FPGA1 – FPGA2 über EH mittels Echtzeitkanal
- Controller in IFB steuert Buszugriff
- MEDL (message descriptor list) beinhaltet statisch festgelegtes Zeitverhalten

Abbildung 5.11: TTP/A im Demonstrator

Abbildung 5.11 demonstriert, erstreckt sich das TTP von der Applikation bis zum Kanal. Darin enthalten ist die Schnittstelle zur Task (*DPRAM*), der IFB, in dem der TTP-Controller implementiert ist und die EH des FPGAs.

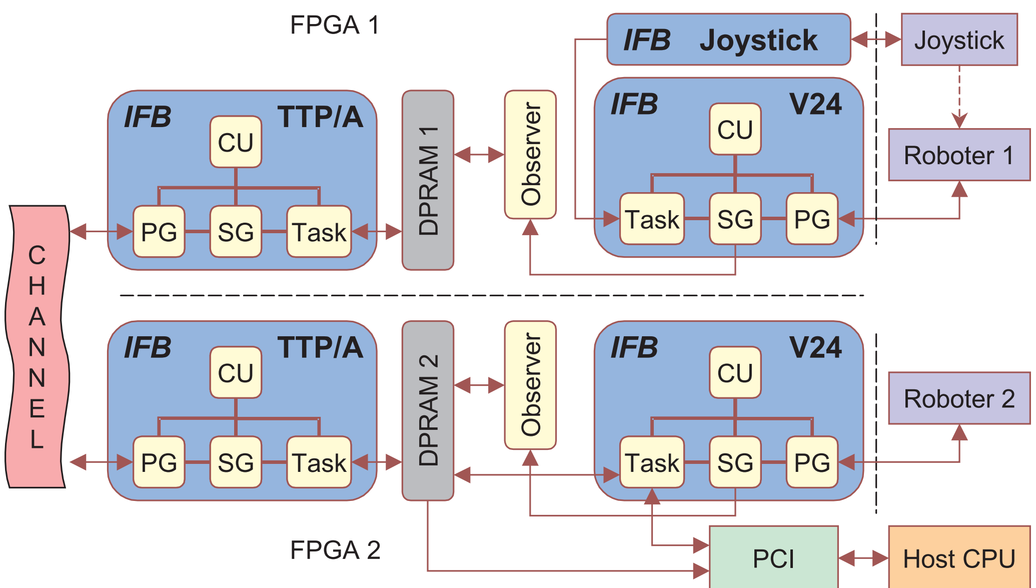


Abbildung 5.12: Interaktion der Echtzeitschnittstellen

Eine umfassende Übersicht der vier vorgestellten Echtzeitschnittstellen des Demonstrators ist in der Grafik 5.12 abgebildet. Es sind insgesamt fünf IFBs notwendig, um die bisher realisierte Funktionalität zu ermöglichen. Der erste IFB stellt die Schnittstelle zum Joystick dar. Die nächsten beiden realisieren je eine V24-Schnittstelle, die zur Steuerung von R1 bzw. R2 dient. Die modellierten Tasks in den V24-IFBs repräsentieren zum einen die Verwaltung der Joystickdaten in FPGA1 und zum anderen den vorgegebenen Job in Verbindung mit der Kollisionsvermeidung in FPGA2. Jedes FPGA besitzt weiterhin die Task *Observer*, die alle Bewegungsdaten mitprotokolliert, die zum Roboter versandt werden. Die Encoderschritte für jeden Motor werden vom *Observer* im DPRAM verwaltet, indem die aktuellen mit den transferierten Encoderschritte verrechnet werden. Die Motordaten von R1, die als Encoderschritte im DPRAM vorliegen, werden durch zwei identische IFBs von FPGA1 nach FPGA2 übertragen. Dabei ist jeweils ein IFB auf einem der FPGAs als TTP/A-Block implementiert. Der verwendete Kanal ist Twisted-Pair-Kabel, das an die Extensionheader angeschlossen ist. Die Motordaten der beiden Roboter, die nun im DPRAM von FPGA2 verfügbar sind, werden unter Anwendung eines Handshakes regelmäßig von der Host-CPU abgerufen und ausgewertet. Im Fall einer vorausberechneten Kollision werden die Ausweichdaten für R2 an die Task im V24-IFB von FPGA2 weitergeleitet, die ansonsten nur Leerrahmen erhält. Erkennt die Task den Eingang von Ausweichdaten, so wird augenblicklich vom vorgegebenen Job auf die Kollisionsvermeidung umgeschaltet. Der Algorithmus der Host-CPU wird nun solange weitere Ausweichdaten erzeugen, bis die Konfliktsituation vorüber ist. Dann kann der Algorithmus den Roboter an den Ausgangspunkt der Kollision oder an den aus der aktuellen Sicht optimalsten Punkt des Jobs fahren und die Kontrolle wieder an die Task im IFB abtreten. Dieser Vorgang ist abhängig von der eingesetzten Ausweichstrategie.

Eine genaue Beschreibung der Implementierung erfolgt im nächsten Abschnitt für folgende Schnittstellen:

- Joystick,
- V24-Schnittstelle und
- TTP/A.

5.6 Implementierung

Den Abschluss des Demonstrator-Kapitels bildet der Abschnitt Implementierung. Dabei wird auf drei echtzeitfähige Schnittstellen näher eingegangen. Abschließend folgt ein Resümee, in dem die erzielten Ergebnisse kurz dargestellt und evaluiert werden.

5.6.1 Joystick

Für jeden der sechs Freiheitsgrade des Joysticks wird eine der acht Leitungen der Joystickschnittstelle zur Verfügung gestellt (vgl. Abbildung 5.6). Die verbleibenden zwei Datenleitungen sind durch je einen Taster belegt. Zusätzlich sind noch 13 weitere Taster vorhanden, deren Signale durch Kombinationen von sonst unmöglichen Joystick-Zuständen, wie z. B. $links \wedge rechts \wedge vor \wedge hoch$, auf den acht Datenleitungen kodiert sind. Der Joystick benötigt eine Versorgungsspannung (VCC), die er über zwei getrennte Kabel bezieht, wobei *VCC1* die Richtungsschalter und *VCC2* die Taster mit Spannung versorgt. Die Signalleitungen liefern im Ruhezustand eine logische Eins, wird der zugehörige Knopf gedrückt, so wird das Potential der Leitungen auf logisch Null gezogen. Dieses Verfahren wird als *low-aktiv* bezeichnet.

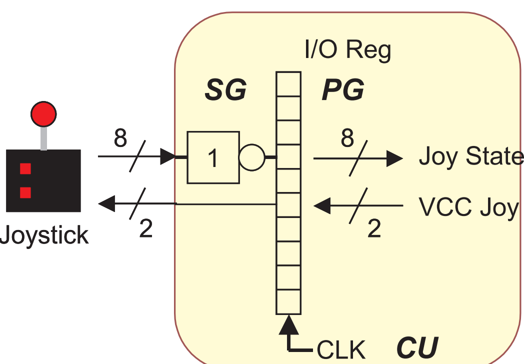


Abbildung 5.13: Schnittstelle des digitaler Joysticks

Die Schnittstelle FPGA-Joystick ist ausschließlich in HW implementiert, die hauptsächlich aus einem getakteten 10 Bit-Register besteht, das im FPGA realisiert ist (vgl. Abbildung 5.13). Zwei Leitungen sind Ausgänge und dienen der Spannungsversorgung des

Joysticks (vgl. Abbildung 5.6). Die restlichen Acht sind als Datenleitungen Input für das FPGA bzw. die Task, die die Joystickdaten verarbeitet. Sie werden regelmäßig mit dem Takt des Registers aktualisiert. Die Geschwindigkeit des Takts bestimmt die Aktualisierungsrate und kann variiert werden, darf allerdings nicht höher als die Laufzeiten der Joysticksignale sein. Die Abtastrate eines Joysticks fällt jedoch nicht in diesen zeitkritischen Bereich.

Die Joystickschnittstelle ist ein Beispiel für den Einsatz eines IFB, in dem die vorgestellte Modellierung aus CU, SG und PG nur schwer zu erkennen ist. Doch dabei muss man sich vor Augen führen, dass Schnittstellen sowohl in ihrem Aufbau als auch in ihrer Komplexität in verschiedenen Formen existieren. Dennoch ist es möglich diese Schnittstelle mit dem beschriebenen Entwurfskonzept abzudecken: Der Sequenzgenerator übernimmt in diesem IFB lediglich die Aufgabe, die Datenleitungen des Joysticks zu invertieren, da im FPGA eine positive Logik implementiert ist. Der Joystick dagegen verfügt über eine *low-aktive* Logik. Eine Steuerung des Sequenzgenerator durch die Kontrolleinheit ist nicht notwendig. Der Protokollgenerator beinhaltet nur das I / O-Register, das die Daten für die auswertende Task bereitstellt. Von der CU erhält der Protokollgenerator den Takt, der bei steigender Flanke das Register aktualisiert. Die Kontrolleinheit hat dabei lediglich die Funktion, die gewünschte Taktrate für den Protokollgenerator aus dem Systemtakt zu generieren.

5.6.2 Serielle Schnittstelle

Der Aufbau und die Implementierung der V24-Schnittstelle ist bereits in Kapitel 4.4 unter dem Gesichtspunkt des Entwurfskonzepts ausführlich dargestellt.

5.6.3 TTP

Im Internet sind mittlerweile verschiedene Realisierungen von TTP/A vorgestellt worden []. Diesen Realisierungen ist gemeinsam, dass die in [Kop01] vorgegebene Spezifikation als *Full-Custom*-Entwurf umsetzen. Die in dieser Arbeit vorgestellte Realisierung hingegen basiert auf einem neuartigen Entwurfskonzept (vgl. Kapitel 4.1). Damit wird am Beispiel des TTP/A-Protokolls [Kop01] gezeigt, dass komplexe Schnittstellenprotokolle umgesetzt werden können.

Implementierung des TTP/A-Protokolls durch den TTP-IFB

Das vollständige TTP/A-System im Rahmen des Demonstrators ist zu komplex, um auf Source-Code-Ebene vorgestellt zu werden. Deshalb wird anhand von Abbildung 5.14 die Implementierung erläutert. Der *TTP-IFB* basiert auf drei Hauptkomponenten: Die Kontrolleinheit, den Sequenzgenerator und den Protokollgenerator (vgl. Abbildung 5.14).

Der *TTP-IFB* verbindet eine Menge von Tasks mit den Übertragungsmedium. Die zu übertragenden Daten der Task befinden sich im DPRAM, bzw. die empfangenen sollen

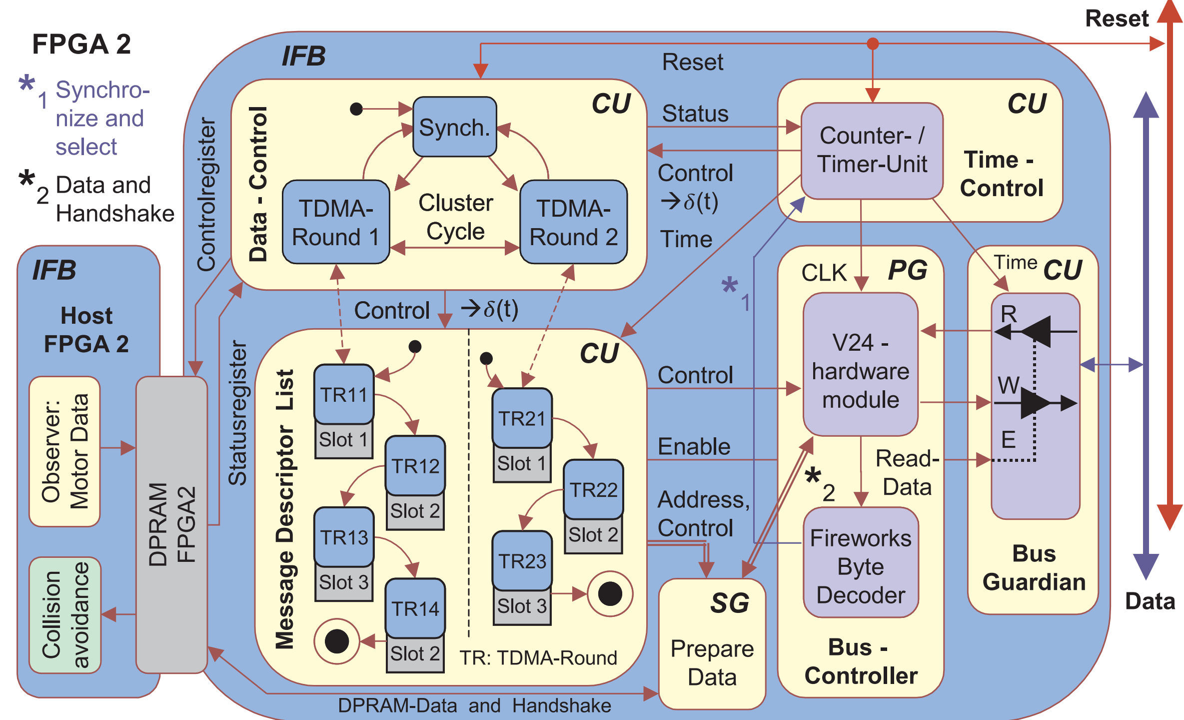


Abbildung 5.14: Implementierung des TTP/A durch den IFB

dort abgelegt werden. Der Kanal verfügt über zwei Leitungen: Eine Datenleitung und eine Resetleitung, mit der ein globales System-Reset durchgeführt werden kann.

Die Kontrolleinheit übernimmt im TTP-IFB eine ganze Reihe von Aufgaben, wie die des *Bus-Guardian* (BG), der *Timer / Counter*-Einheit und der Kontrollfunktion (*Data-Control*) in Zusammenhang mit der Message Descriptor List (MEDL). Der BG ist eine Einheit, die den physikalischen Buszugriff über *Tri-State*-Treiber ermöglicht. Die zugehörigen *Enable*-Leitungen werden von der MEDL in der Kontrolleinheit gesteuert. Um Fehlertoleranz im Zeitbereich zu ermöglichen, gibt die *Timer / Counter*-Einheit Referenzwerte an den BG, der bei Abweichungen den Bus-Zugriff unterbricht bis der Konten neu synchronisiert wurde. Die *Timer / Counter*-Einheit liefert weiterhin das *Clock*-Signal für den Protokollgenerator (CLK) und die Zeitbasis für die MEDL sowie die Vorgabe der aktuellen TDMA-Runde. Um verschiedenen TDMA-Runden zu beschreiben, enthält die Kontrolleinheit (CU) eine Komponente mit der Bezeichnung *Data-Control*, die in Form eines kommunizierenden Automaten implementiert ist. Eine TDMA-Runde (CU-Modus) wird durch eine feste Folge von Zuständen in der MEDL realisiert. Die Auswahl einer TDMA-Runde geschieht durch die Aktivierung des zugehörigen Zustandes im *Data-Control*-Automat (vgl. Moduswechsel im Sequenz- und Protokollgenerator in Kapitel 4.1). Ein Zustand der MEDL beinhaltet genau die MEDL-Einträge, die in dem Zeitraum aktuell sind, wenn der Zustand aktiv ist. Die Zustandsübergänge innerhalb der MEDL sind daher an feste Zeitpunkte gebunden, die in der Ablaufplanung des Systems

festgehalten sind und durch die *Timer / Counter*-Einheit vorgegeben werden. Die Informationen aus der MEDL werden zur Steuerung der Enable-Leitungen des Bus-Guardians sowie des Protokollgenerators eingesetzt. Weiterhin enthält die MEDL die I / O-Adressen für den Zugriff auf den DPRAM, die vom Sequenzgenerator benötigt werden.

Die Aufgabe des Sequenzgenerators beschränkt sich bei TTP/A auf die Datenkommunikation mit dem DPRAM und dem Protokollgenerator. Im Sequenzgenerator werden die Daten byteweise für den Protokollgenerator vorbereitet, die von dort übertragen werden. Auf der anderen Seite übernimmt der Sequenzgenerator das Weiterleiten von empfangenen Daten an das DPRAM.

Der Protokollgenerator übermittelt die vom Sequenzgenerator bereitgestellten Daten seriell auf dem Datenbus und ist durch einen HW-UART-Baustein realisiert, wie er im Abschnitt *serielle Schnittstelle* vorgestellt wird. Darin ist beschrieben, wie ein UART-Baustein durch einen V24-IFB implementiert werden kann. Die serielle Übertragung, die durch die Einträge in der MEDL gesteuert wird, ist echtzeitfähig. Durch Messungen ergeben sich Diagramme, die vergleichbar mit denen der V24-Schnittstelle aus Abbildung 4.17 sind. Der Zugriff auf das Medium erfolgt über den BG der, wie ober beschrieben, die Einhaltung der Zugriffszeiten überwacht. Um die Synchronisation in dem TTP/A-System zu ermöglichen ist im Protokollgenerator ein *Fireworks-Byte Decoder* enthalten, der die empfangenen Daten des UART auf ein Fireworks-Byte hin überprüft. Wird ein solches Byte erkannt, erfolgt die sofortige Meldung an die *Timer / Counter*-Einheit. Ausgehend von dem Zeitpunkt der Erkennung T_{RecFw} können alle Komponenten des TTP-IFB durch die *Timer / Counter*-Einheit innerhalb des Zeitraums T_{Init} zum globalen Synchronisationszeitpunkt T_{Synch} resynchronisiert werden (vgl. Abbildung 5.15).

Zeitberechnung im TTP/A-Protokoll

Im Modellierungskapitel wurde beschrieben, dass die Zeitvorgaben des IFB in der Task, dem Kanal und der Zielplattform modelliert sind. Das setzt voraus, das in der Modellierung des Kanals eine Menge vorgegebener Protokolle enthalten sind, die den Anforderungen der Task entsprechen. Um mehr Flexibilität zu bieten kann auch eine andere Lösung in Betracht gezogen werden: Aus den Anforderungen der Task wird ein *Full-Custom*-Protokoll berechnet. Die Einschränkung auf vorgegebene Standardprotokolle kann somit aufgehoben werden, allerdings müssen sich die Anforderungen des berechneten Protokolls immer noch der vom Kanal bereitgestellten Bandbreite unterordnen. Eine Vorgehensweise der Berechnung eines solchen *Full-Custom*-Protokolls wird am Beispiel des *TTP-Cluster-Cycles* (vgl. Abbildung 5.15) repräsentativ für andere zeitbehaftete Systeme demonstriert.

Der Vorgang der Ablaufplanung in Verbindung mit der statische Kanalzuteilung legt bereits während des Entwurfs eines Systems alle Zeiten für die Knoten fest. Der Buszugriff erfolgt über ein TDMA-Verfahren. Einen Zyklus, in dem jedem enthaltenen Knoten des Systems eine festgelegte Sendezeit zugeteilt ist, bezeichnet man als TDMA-Runde. Die Folge aller unterschiedlichen TDMA-Runden ergibt schließlich den Cluster-Cycle. Eine genaue Beschreibung des Sachverhalts findet sich im Abschnitt 2.2.7.

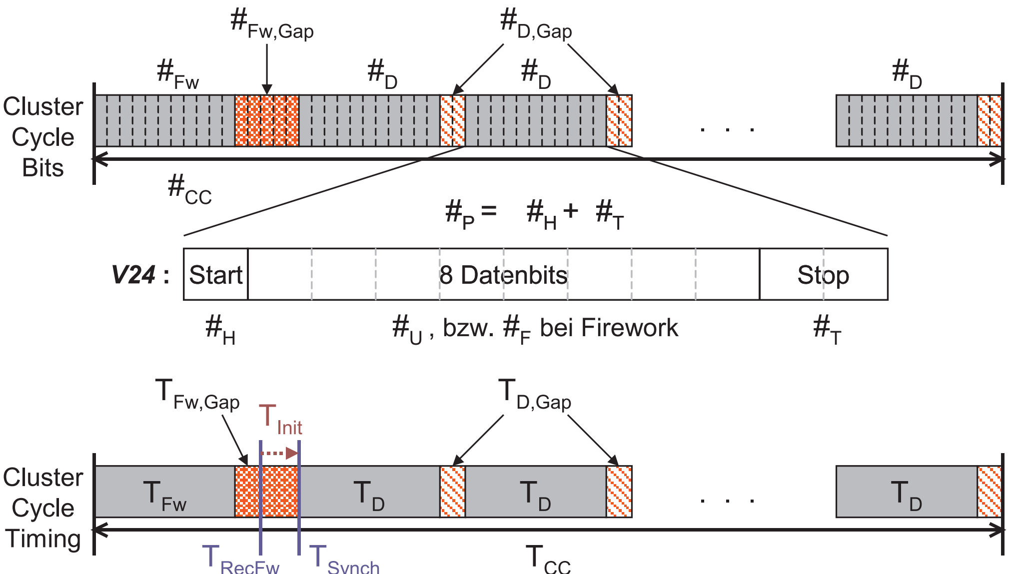


Abbildung 5.15: Berechnung des *Cluster Cycles*

Die Ausgangswerte für die Berechnung des Protokolls sind aus drei Quellen zu entnehmen. Diese Werte sind für den Rechenvorgang als konstant, da unabänderlich, anzunehmen. Die Bandbreite des Kanals liefert die obere Grenzfrequenz $f_{Ch,max}$ und damit die maximale Übertragungsrate des Protokolls. Durch Zeitrestriktionen der Task wird der Aufbau des Protokolls bestimmt, wie z. B. der Zeitabstand für die Synchronisation der Knoten. Daraus ergibt sich die Länge einer *TDMA-Runde*, die als T_{TR} bezeichnet ist. Zuletzt werden noch die Rahmenbedingungen der Zielhardware einbezogen. In diesem Beispiel gibt das FPGA z. B. den maximale Systemtakt $f_{FPGA,max}$ vor. Da im Demonstrator nur eine TDMA-Runde vorhanden ist, entspricht deren Wiederholung bereits dem Cluster-Cycle, d. h. $T_{CC} = T_{TR}$. Abbildung 5.15 (oberer Teil) veranschaulicht den Aufbau dieses Cluster-Cycles strukturiert nach übertragenen Bits. Die einzuhaltenden Zeiten sind in der unteren Hälfte der Abbildung 5.15 der Strukturierung nach Bits zugeordnet.

Im Demonstrator sind diese Konstanten wie folgt festgelegt:

- $f_{Ch,max} = 2\ MHz$ (Übertragungsrate des Kanals gemäß Vorgabe)
- $T_{CC} = T_{TR} = \frac{1}{8kHz} = 125\ \mu$s (Synchronisation mit 8 kHz: Vergleich zu FireWire)
- $f_{FPGA,max} = 40\ MHz$ (PCI-Bus Takt als Clock im FPGA)

Der Zusammenhang zwischen Frequenz und Sendezeit lautet allgemein: $T = \frac{1}{f}$. Daraus resultieren die zwei im TTP-IFB beinhalteten Taktraten:

- $T_{Ch,min} = \frac{1}{f_{Ch,max}} = 500\ ns$ (Sendetakt für Datentransfer)

- $CLK = \frac{1}{f_{FPGA,max}} = 25\ ns$ (Verarbeitungstakt im FPGA)

Aufbauend auf diesen Daten kann nun die Berechnung der des Protokolls beginnen. Es gilt folgende Gleichungen für die Bitanzahl in einem Cluster-Cycle ($\#_{CC}$):

$$\#_{CC} = \frac{T_{CC}}{T_{Ch,min}} = \frac{125.000\ ns}{500\ ns} = 250\ Bits$$

Durch Umformung und Einsetzen lässt sich diese Gleichung folgendermaßen notieren:

$$\begin{aligned}
T_{CC} &= \#_{CC} \cdot T_{Ch,min} \\
&= \#_{CC} \cdot \frac{T_{Ch,min}}{CLK} \cdot CLK \\
mit &\quad \#_{Counter} = \frac{T_{Ch,min}}{CLK} \quad gilt: \\
&= \#_{CC} \cdot \#_{Counter} \cdot CLK \\
&= 250 \cdot 20 \cdot CLK
\end{aligned}$$

Als Ergebnis hat sich herausgestellt, dass sich die Zeit für den Cluster-Cycle T_{CC} als Produkt der Paketgröße ($\#_{CC}$), des Systemtakts der Zielplattform (CLK) und $\#_{Counter}$ beschreiben lässt. Es ist darauf zu achten, dass der Faktor $\#_{CC}$ (Anzahl der Bits im Cluster-Cycle) ganzzahlig sein muss, da nur vollständige Bits übermittelt werden können. Ebenso ist es von großer Bedeutung, dass das Verhältnis $\#_{Counter} = \frac{T_{Ch,min}}{CLK}$ ganzzahlig ist, da die Zielplattform dann Vielfache des Systemtakts CLK abzählen kann, um $T_{Ch,min}$ zu berechnen.

Nachdem diese Werte festgelegt sind, muss eine *Partitionierung* des Cluster-Cycles in die einzelnen Pakete und die dazwischen befindlichen Zeitlücken (Gap) durchgeführt werden. Dazu wird die Definition des Aufbaus eines Cycle-Clusters wie folgt formuliert (vgl. Abbildung 5.15):

$$\begin{aligned}
\#_{CC} &= (\#_H + \#_F + \#_T) + \#_{Fw,Gap} + n(\#_H + \#_U + \#_T) + n \cdot \#_{D,Gap} \\
mit &\quad \#_P = \#_H + \#_T \quad gilt: \\
\#_{CC} &= (\#_P + \#_F) + \#_{Fw,Gap} + n(\#_P + \#_U + \#_{D,Gap}) \\
mit &\quad \#_U = \#_F \quad gilt: \\
\#_{CC} &= (n+1) \cdot (\#_P + \#_U + \#_{D,Gap}) + (\#_{Fw,Gap} - \#_{D,Gap}) \\
\#_{CC} &= (18+1) \cdot (3+8+2) + (5-2) = 19 \cdot 13 + 3 = 250\ Bits
\end{aligned}$$

Die Rechnung bestätigt die schon zu Beginn berechnete Gesamtsumme der Bits eines Cluster-Cycles. Ein Maß, an dem die Güte der Partitionierung geprüft werden kann, ist die Effektivität $\#_{Eff}$ des Protokolls, die folgendermaßen festgelegt ist:

$$\begin{aligned}
\#_{Eff} &= \frac{Information}{Information + Redundanz} \\
\#_{Eff} &= \frac{n \cdot \#_U}{(n+1) \cdot (\#_P + \#_U + \#_{D,Gap}) + (\#_{Fw,Gap} - \#_{D,Gap}})
\end{aligned}$$

Für eine große Anzahl an Paketen kann die Effektivität $\#_{Eff,\infty}$ durch folgende Nährung abschätzt werden:

$$\begin{aligned}\#_{Eff,\infty} &= \frac{n \cdot \#_U}{(n+1)\cdot(\#_P + \#_U + \#_{D,Gap}) + (\#_{Fw,Gap} - \#_{D,Gap}}) \\ &\leq \frac{n \cdot \#_U}{(n+1)\cdot(\#_P + \#_U + \#_{D,Gap})} \\ &\leq \frac{\#_U}{\#_P + \#_U + \#_{D,Gap}} \\ &= 1 - \frac{1}{1 + \frac{\#_U}{\#_P + \#_{D,Gap}}}\end{aligned}$$

Die Redundanz, die den Nutzdaten durch das Protokoll hinzugefügt wurde, wird auch als *Overhead* bezeichnet und findet sich in der Gleichung im folgenden Trem wieder:

$$Overhead_\infty = \frac{1}{1 + \frac{\#_U}{\#_P + \#_{D,Gap}}}$$

Tabelle 5.2: Eine mögliche Partitionierung des Cluster-Clycles

Beschreibung	CC		Paket		Summe	Zeit/Paket
Fireworks	1	·	11	=	11	5,5 µs
Datenpaket	18	·	11	=	198	5,5 µs
Fireworks-Gap	1	·	5	=	5	2,5 µs
Daten-Gap	18	·	2	=	36	1,0 µs
Gesamt					250	125 µs

Der Overhead ist immer kleiner eins und sinkt stetig bei einem steigenden Verhältnis von Nutzdaten zu Protokolldaten und Verzögerungszeiten. In Tabelle 5.2 ist eine Beispiel einer solchen Partitionierung angegeben. Die maximal damit zu erreichende Effektivität für sehr viele Datenpakete beträgt:

$$\begin{aligned}\#_{Eff,\infty} &= 1 - \frac{1}{1 + \frac{\#_U}{\#_P + \#_{D,Gap}}} \\ &= 1 - \frac{1}{1 + \frac{8}{3+2}} \\ &= 1 - \frac{5}{13} = \frac{8}{13} \\ &\hat{=}\ 61,5\%\end{aligned}$$

Die tatsächlich erreichte Effektivität erhält man durch Einsetzen der Parameter in die vollständige Formel:

$$\#_{Eff} = \frac{n \cdot \#_U}{(n+1)\cdot(\#_P + \#_U + \#_{D,Gap}) + (\#_{Fw,Gap} - \#_{D,Gap}})$$

$$= \frac{18 \cdot 8}{(18+1) \cdot (3+8+2) + (5-2)}) = \frac{144}{250}$$
$$\triangleq 57,6\%$$

Die *bezogenen Effektivität* dieser Partitionierung beträgt damit:

$$\#_{Eff,bez} = \frac{\#_{Eff}}{\#_{Eff,\infty}} = \frac{0,576}{0,615} = 0,937$$
$$\triangleq 93,7\%$$

Das bedeutet, bei einer Paketanzahl von 18 Datenpaketen und einem Firework-Paket, wird eine Rate von 93,7 % der, mit diesem Protokoll möglichen Effektivität erzielt. Die Angaben über die Effektivität einer Partitionierung helfen zu beurteilen wie stark sich der Anteil des Overhead im Protokoll auswirkt. Das gibt gleichzeitig an, welcher Prozentanteil der Bandbreite noch für die Nutzdaten verbleibt.

Um die Ausgangswerte der Berechnung und der Partitionierung des Cluster-Cycles zu bestimmen, ist eine Auswertung des Zeitaspekts der Modellierung für die Task, den Kanal und die Zielplattform erforderlich. Die Modellierung des Zeitaspekts für eine Datenpaket ist in Abbildung 5.16 durch ein Sequenzdiagramm dargestellt. Ausgehend von der Klasse *Frame* im Objekt *Paket* ist der Aufbau eines Pakets aus Header, UseData und Trailer veranschaulicht. Die Definition der Zeiten sind durch die Zeitrestriktionen in der Farbe blau modelliert.

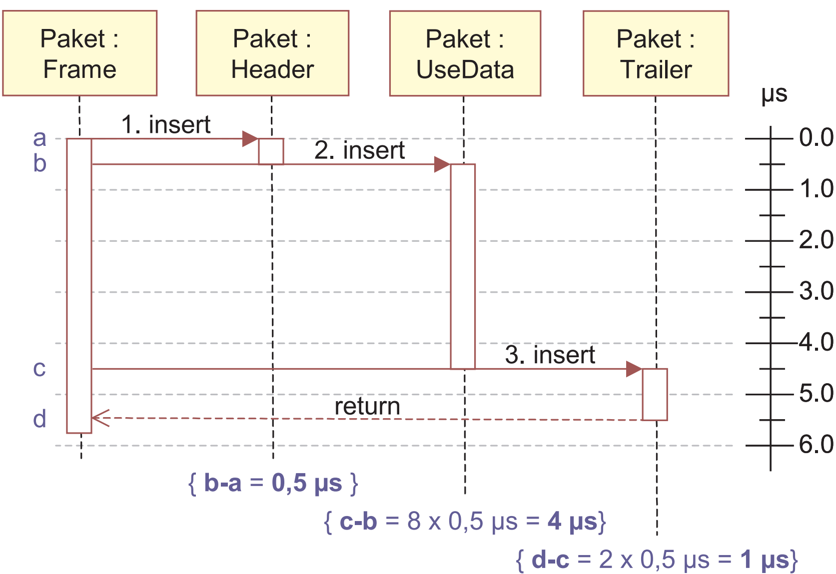


Abbildung 5.16: Zeitbehaftete Modellierung: Das Paket

Diese Sequenzdiagramme sind auch im Kontext des automatischen Entwurfs von Bedeutung. Aus ihnen können Informationen über Aufbau und Zeitverhalten des Protokolls

gewonnen werden. Auf den ersten Blick erscheint das redundant mit den Angaben des Kanals über die Bandbreite und die unterstützten Protokolle. Dort werden ebenfalls Informationen über Aufbau und Zeitverhalten von Protokollen beschrieben. Der Unterschied besteht darin, dass der Kanal diese Angaben nur für vordefinierte Protokolle bereitstellt. Sollte jedoch, wie in diesem Fall ein *Full-Custom*-Entwurf des Protokolls erfolgen, der eine Berechnung aus den Vorgaben der Task notwendig macht, sind die Sequenzdiagramme notwendig.

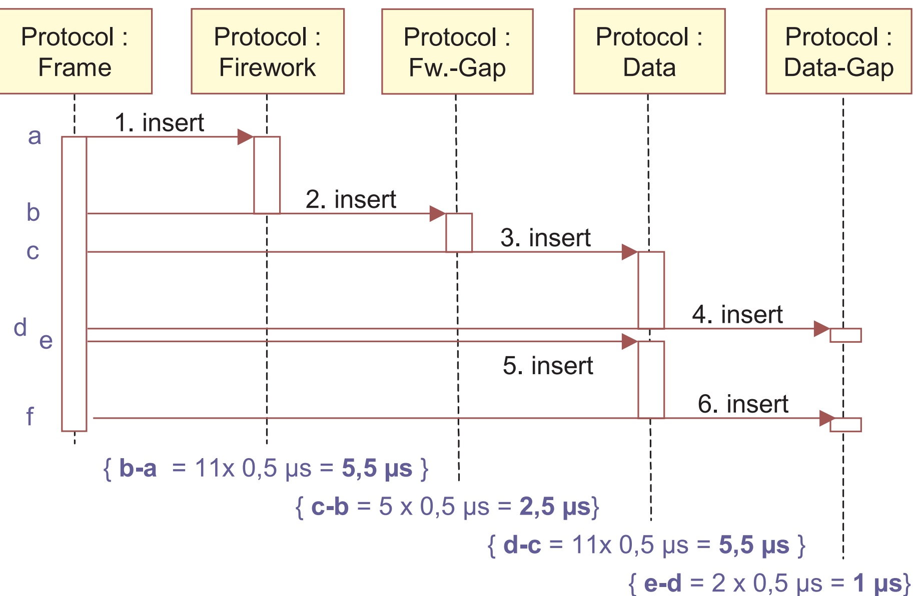


Abbildung 5.17: Zeitbehaftete Modellierung: Das Protokoll

Abbildung 5.17 beschreibt den übergeordneten Kontext zu Abbildung 5.16. Dort ist das Protokoll dargestellt, das bei TTP in Form eines *Cluster-Cycles* einen Sendezyklus beschreibt. Es beginnt mit dem Fireworks-Byte, gefolgt von der „Synchronisations-Lücke“, in der die einzelnen Knoten ihre Synchronisation durchführen können. Darauf werden dann die Datenpakete eingefügt, denen jeweils eine kurze Arbitrierungszeit folgt. So können mit Hilfe von Sequenzdiagrammen die zeitlichen Vorgaben für die Ablaufplanung des TTP/A-Systems modelliert werden. Ebenso kann durch Sequenzdiagramme die Vorgabe des Buszugriffs der beteiligten Knoten erfolgen (ohne Abbildung). Dazu müssen alle Knoten mit Buszugriff im Diagramm eingetragen und jeweils zu den gewünschten Zeitpunkten als lesend oder schreibend markiert werden.

Der Abschnitt *Zeitberechnung im TTP/A-Protokoll* hat somit gezeigt, dass alle notwendigen Beschreibungsmittel zur Verfügung stehen, um den Aspekt der Echtzeit zu beschreiben. Ebenso können auch erforderliche Berechnungen in Zusammenhang mit der Modellierung in Form von Sequenzdiagrammen vorgenommen werden.

5.7 Ergebnisse

In diesem Kapitel ist das Szenario des in dieser Arbeit entwickelten Demonstrators ausführlich beschrieben worden. Ebenso wurde die Funktionalität und der Aufbau der Komponenten erläutert. Die Intention des Demonstrators war die Validierung des Entwurfskonzepts. Der Aufbau des Szenarios ist daher so gewählt worden, dass verschiedenen Schnittstellen mit Echtzeitanforderungen im Demonstrator enthalten sind. Die vier Wichtigsten davon sind im einzelnen vorgestellt und ihre Funktionsweise erklärt worden.

Die Anwendung des Modellierungskonzepts hat gezeigt, dass alle relevanten Aspekte dieser betrachteten Schnittstellen abgedeckt werden können:

- Verschiedene Schnittstellen mit sehr unterschiedlichen Eigenschaften sind durch das Konzept abgedeckt und können gemäß der Kontrolleinheit und der beiden Generatorstufen Sequenz- und Protokollgenerator modelliert und implementiert werden.
- Die Modellierung durch Klassendiagramme und Automaten hat gezeigt, dass eine generelle Abbildung auf eine Implementierung möglich ist.
- Durch den Einsatz von Sequenzdiagrammen ist die adäquate Beschreibung des Zeitaspekts möglich.
- Das betrachtete System darf über mehreren Tasks und Kanäle verfügen.
- Die Realisierung des Entwurfskonzepts durch den IFB funktioniert für verschiedene Schnittstellen.
- Das Konzept ist in sich abgeschlossen im Gegensatz zu traditionellen Ansätzen, die nur Teilaspekte des Entwurfsprozesses betrachten.
- Das Modellierungskonzept besitzt durch den Einsatz von UML eine einfache Erweiterbarkeit des Modells für weitere Anforderungen der Tasks oder des Kanals.

Weiterhin hat der Demonstrator „am Beispiel interagierender Roboter“ veranschaulicht, dass das Thema der Arbeit „Entwurf von Echtzeitschnittstellen“ in der Praxis relevant ist, und in vielen Bereichen der Technik eine Anwendung findet.

Im Rahmen des Demonstrator wurde ein vollständiger Entwurfsablauf durchgeführt, bis hin zur Implementierung der Komponenten. Ein wichtiger Punkt dabei war die Beachtung der Synthesefähigkeit, sowohl im Bereich der Modellierung wie auch für die Implementierung. Ein Beispiel dazu ist die V24-Schnittstelle, an der das exemplarisch im Kapitel 4 demonstriert worden ist. Das so gewonnene Verfahren schafft die Basis für Vergleichsmessungen im Bereich des Schnittstellenentwurfs.

Zusammenfassend ergibt sich das Ergebnis, dass der Entwurf von unterschiedlichen Schnittstellen mit:

- nur einem einheitlichen,
- Ebenen-übergreifenden,
- leicht erweiterbaren und
- wohldefinierten

Konzept möglich ist. Anhand des Demonstrators ist daher nachgewiesen worden, dass das *Entwurfskonzept* den Aspekt der *Modellierung* von Echtzeitschnittstellen in allen geforderten Bereichen abdeckt. Ebenso wurden erfolgversprechende Ansätze für den automatisierten Entwurf vorgestellt. Damit hat sich herausgestellt, dass die Gliederung des Entwurfs in Modellierung und automatisierten Entwurf, wie sie in Abbildung 5.18 symbolisiert ist und in Kapitel 1 vorgestellt wurde, erfolgreich zu einem anwendbaren Entwurfskonzept geführt hat.

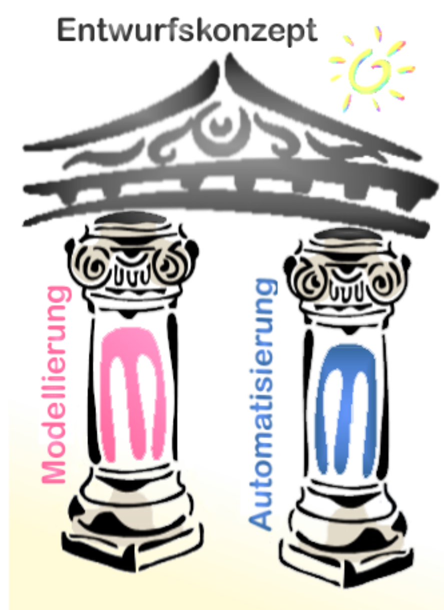


Abbildung 5.18: Schnittstellenentwurf als Säulenmodell illustriert

6 Zusammenfassung und Ausblick

Dieses Kapitel schließt diese Arbeit mit einer Zusammenfassung der bearbeiteten Themen ab. Dabei werden die entscheidenden Konzepte noch einmal aufgeführt und daran erläutert, dass die Aufgabenstellung erfolgreich bearbeitet werden konnte. Im Anschluss wird ein Ausblick auf Weiterentwicklungsmöglichkeiten und Zukunftsperspektiven gegeben.

6.1 Zusammenfassung der Arbeit

Ziel dieser Arbeit war die Entwicklung eines neuartigen Konzepts um den Entwurfsprozess von Schnittstellen unter Echtzeitbedingungen zu erleichtern. Die Betrachtung konzentrierte sich dabei auf zwei Schwerpunkte: Modellierung und automatisierten Entwurf. Der Aspekt Modellierung ist dabei besonders intensiv betrachtet worden, da er als Hilfsmittel zur Modellierung des Entwurfskonzepts dient. Darauf bauen auch die Grundlagen des automatischen Entwurfs auf.

Die vorliegende Arbeit ist in drei Hauptteile gegliedert. Im ersten Teil (vgl. Kapitel 1) wurden die Grundlagen zur Bewertung von Schnittstellen strukturiert zusammengestellt. Dabei ist großer Wert auf Detailtreue und Vergleichbarkeit für ein breites Spektrum an Schnittstellen gelegt worden. Insbesondere wurden Übertragungsmedien, - Protokolle und Echtzeitanforderungen analysiert.

Im zweiten Teil wurde auf dieser Basis das Modellierungskonzept vorgestellt. UML-Techniken, d. h. UseCase Diagramme, Klassendiagramm, Statecharts, Sequenzdiagramme und Aktivitätendiagramme wurden für die Beschreibung dedizierter Aspekte von Schnittstellen angewandt. Das Modellierungskonzept schließt die Anforderungsspezifikation sowie die verschiedenen Entwurfsebenen bis hin zur konkreten Datenkodierung ein. Dabei wurden auch die Echtzeitaspekte berücksichtigt. Es ist damit in dieser Arbeit ein adäquates Konzept zur Modellierung von Echtzeitschnittstellen vorgestellt worden. Darauf aufbauend wurde das Entwurfskonzept eingeführt und konform zur Modellierung eine Entwurfsstruktur vorgestellt, die durch den Interface-Block (IFB) realisiert wird. Der IFB untergliedert sich in Kontrolleinheit, Sequenzgenerator und Protokollgenerator und wurde auf Basis von endlichen Automaten implementiert. Dieses Konzept ist auf die serielle Schnittstelle sowie die Implementierung eines TTP-Systems angewandt worden.

Der Demonstrator, aufgebaut aus zwei kooperierenden Robotern mit dem Ziel der Kollisionsvermeidung, illustriert die Modellierung und das Entwurfskonzept. Die Implementierung des Demonstrators bildete den praktischen Teil dieser Arbeit und lieferte die exemplarische Validierung der Modellierung und des Entwurfskonzepts. Das Roboter-

Szenario wurde im letzten Teil der Arbeit (vgl. Kapitel 5) vorgestellt. Dazu sind die im Demonstrator enthaltenen Echtzeitschnittstellen vorgestellt und analysiert worden. Insbesondere wurde auch der Echtzeitaspekte im Rahmen der Anwendungen behandelt und erfolgreich in das Entwurfskonzept integriert.

6.2 Ausblick auf Modellierung und Entwurf

Es hat sich gezeigt, dass die Idee eines einheitlichen Entwurfskonzepts umgesetzt werden konnte. Auch die Beschreibung der Konzepte durch eine Modellierung ist erfolgreich, dennoch verbleiben einige Punkte, deren genauere Betrachtung Gegenstand weiterer Untersuchungen sein könnte:

- Bisher sind zwei Anwendungsgebiete des Entwurfskonzepts vorgestellt worden: Zum einen kann der IFB zwischen Task und Kanal eingesetzt werden und zum anderen bei der Adaption einer Task an ein bestehendes System. Es existiert aber noch eine weitere Kombinationsmöglichkeit, auf die der IFB angewendet werden kann: Eine Real-Time-Bridge ist die Schnittstelle zwischen zwei verschieden Kanälen, die miteinander verbunden werden müssen. Dazu sind Protokolltransformationen, die sich sowohl auf die Daten als auch auf den physikalischen Bereich auswirken können, notwendig. Damit realisiert der IFB dann die Schnittstelle zwischen Kanal1 und Kanal2.
- Die vollständigen Automatisierung der Codegenerierung erfordert die exakte Definition der Modellierung. Dabei muss beachtet werden, das verschiedenen Zielarchitekturen unterschiedliche Ansprüche an die Synthese und somit auch die Codegenerierung stellen. Die notwendigen Informationen müssen dazu in den Diagrammen der Modellierung enthalten sein. Eine Weiterentwicklung der Modellierung und der Codegenerierung führt schließlich zum vollständig automatisierten Entwurf.
- Als weitere Komponente kann die Modellierung der Zielplattform in den IFB integriert werden. Das erscheint insofern sinnvoll, da das Ziel des automatisierten Entwurfs im Bereich der eingebetteten Systeme die Abbildung des Modells auf diese Zielplattform ist. Damit wären dann auch alle für die Synthese notwendigen Informationen im Modell des IFB zusammengefasst.

6.3 Ausblick für den Demonstrator

Im Umfeld des Demonstrators sind noch weitere Forschungsarbeiten im Rahmen von Projektgruppen, Studien- und Diplomarbeiten denkbar. Insbesondere sind die folgenden Themen geeignet, die in der Arbeit angesprochenen wurden, deren vollständige Ausarbeitung aber aus Zeitgründen nicht möglich war:

- Aufstellen eines vollständigen mathematischen Robotermodells, das für eine Positionsberechnung mit Integerarithmetik geeignet ist. Die Zielsetzung beruht auf der echtzeitfähigen Berechnung und Auswertung des Modells bei der sich anschließenden Kollisionsvermeidung. In diesen Kontext fällt auch die Implementierung einer intelligenten Ausweichstrategie. Dabei ist zu beachten, dass ,obwohl sich die Roboter möglichst nahe kommen sollen, der Sicherheitsaspekt nicht verletzt werden darf.

- Das Time Triggered Protocol ist im Allgemeinen für verteilte und fehlertolerante Systeme entworfen worden. In dieser Arbeit ist der Aspekt Fehlertoleranz nur im Zeitbereich berücksichtigt worden. TTP kann noch um verschiedene Instanzen von FTUs erweitert werden, um so durch Redundanz von Komponenten zusätzliche Ausfallsicherheit zu garantieren.

- Die Ansteuerung des Roboters durch einen analogen Joystick würde dem Bediener einen größeren Komfort bieten und unterschiedliche Fahrgeschwindigkeiten für Roboterglieder ermöglichen.

- Der bisher implementierte Sourcecode in VHDL befindet sich im Stadium der Anpassung an das Entwurfskonzept unter dem Aspekt des entwickelten Modellierungskonzepts. Eine Optimierung des VHDL-Codes hätte eine verbesserte Ressourcennutzung und ein einfacheres Laufzeitverhalten zur Folge, was für die angestrebte Automatisierung wichtige Erkenntnisse liefern würde.

Literaturverzeichnis

[Ada01] Adaptec. *FireWire / IEEE 1394 Backgrounder.* http://www.adaptec.com/-worldwide/product/markeditorial.html, 2001.

[Alt01] Altera, San Jose, CA. *Altera Data Book*, 2001.

[And99] Don Anderson. *FireWire System Architecture.* Addison Wesley, Mindshare Inc., second edition, 1999.

[App01] Apple. *FireWire / IEEE 1394 – Enwicklung und Anwendung, Herstellerfirma.* http://www.apple.de, 2001.

[Bec01] Prof. Peter E. Beckmann. *http://nld.physik.uni-mainz.de/determinismus.htm.* Arbeitsgruppe Nichtlineare Dynamik Institut für Physik, Fachbereich Physik, Johannes Gutenberg-Universität Mainz / Germany, Okt 2001.

[Ber00] Udo Bertholdt. *http://www.elektron-bbs.de/elektronik/index.htm.* Elektron BBS, Dezember 2000.

[Bur01] Alan Burns. *Real time systems and programming languages.* Addison-Wesley, Harlow [u.a.], 3 edition, 2001. Ada 95, real time Java and real time POSIX / Alan Burns and Andy.

[But00] Giorgio C. Buttazzo. *Hard real time computing systems : predictable scheduling algorithms and applications.* Kluwer, Boston [u.a.], 3 edition, 2000. (Kluwer international series in engineering and computer science : Real-time systems).

[BY96] Barabanov and Yodaiken. Real-Time Linux. *Linux Journal*, Mar 1996.

[CC94] R. Creasy and R. Coirault. *VHDL Modelling Guidelines.* ESA – European Space Agency, Noordwijk, The Netherlands, 1994. European space research and technology centre.

[DEC01] DECOMSYS. *Dependable Computer Systems – Your FlexRay Devolop Partner.* http://www.decomsys.com, 2001.

[Dem97] Klaus Dembowski. *Computerschnittstellen und Bussysteme.* Hüthing Verlag, Heidelberg, first edition, 1997.

[Dou98] Bruce Powel Douglas. *Real-time UML : developing efficient objects for embedded systems.* Addison-Wesley, Reading, Massachusetts [u.a.], first edition, 1998. The Addison-Wesley object technology series.

[Dou00] Bruce Powel Douglas. *Doing hard time : developing real-time systems with UML, objects, frameworks and patterns.* Addison-Wesley, Reading, Massachusetts [u.a.], third edition, 2000. The Addison-Wesley object technology series.

[Eng98] Tomi Engdal. *http://www.epanorama.net/documents/joystick/index.html.* www.ePanorama.net, Okt 8996-1998.

[Eng01] Gregor Engels. TDSE 1, Techniken des Softwareentwurfs. *Vorlesung, Universität Paderborn, Universität der Informationsgesellschaft*, WS 2000/2001.

[Epp97] Jerry Epplin. Linux as an Embedded Operating System. *http://www.embedded.com/97/fe39710.htm*, Oct 1997.

[ESP98] ESPRIT OMI Project 23396. *TTA: Time-Triggered Architecture.* http://www.vmars.tuwien.ac.at/projects/tta/index.html, 1996 - 1998.

[Fis01] Karsten Fischer. *Universal Serial Bus, Schnittstellen in Multimedia Computern.* Seminar, Uni Paderborn, Warburger Straße 100, 33098 Paderborn, April 2001.

[Fla98] Rony G. Flatscher. *http://www.wu-wien.ac.at/glossar.* Abteilung für Wirtschaftsinformatik, März 1998.

[Fle01] FlexRay. *The Communication System for advanced automotive control applications.* http://www.flexray-group.com, 2001.

[For00] USB Implementers Forum. USB-IF: Universal Serial Bus Specification. 2000.

[Gom00] Hassan Gomaa. *Designing concurrent, distributed, and real-time applications with UML.* Addison-Wesley, Boston [u.a.], 2000. The Addison-Wesley object technology series.

[Har96] Wolfram Hardt. *HW/SW-Codesign auf Basis von C-Programmen unter Performanz-Gesichtspunkten.* PhD thesis, University of Paderborn, Shaker Verlag, Aachen, 1996.

[Har00] Wolfram Hardt. Integration von verzögerungzeit-invarianz in den entwurf eingebetteter systeme, 2000.

[Har01] Info's und mehr Hardwareecke. *FAQ zu Schnittstellen, Hardware FAQ.* http://www.hardwareecke.de/specials/rs232.html, Oktober 2001.

[HLM00] W. Hardt, T. Lehmann, and Visarius M. Towards a design methodology capturing interface synthesis. *Universität Paderborn, Computer Science Department*, 2000.

[HP94] John L. Hennessy and David A. Patterson. *Computer organization and design : the hardware software interface*. Morgan Kaufmann, San Francisco, Calif., 1994. With a contrib. by James R. Larus.

[HT00] W. Hardt and J. Teich. HW / SW-Codesign. *Vorlesung, Universität Paderborn, Universität der Informationsgesellschaft*, SS 2000.

[HT01] Wolfram Hardt and Jürgen Teich. HW/SW-Codesign, Hardware/Software-Codesign. *Vorlesung, Universität Paderborn, Universität der Informationsgesellschaft*, WS 2000/2001.

[HV01] W. Hardt and M. Visarius. *IPQ-Project*. University of Paderborn, IPL, Paderborn, Germany, 2001.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman Inc., Reading, Massachusetts [u.a.], first edition, 1999. The Addison-Wesley object technology series.

[Kam96] K.-D. Kammeyer. *Nachrichtenübertragung*. Teubner, Stuttgart, second edition, 1996.

[Kel99] Hans J. Kelm. *USB - Universal Serial Bus*. Franzis Verlag, 1. edition, 1999.

[Kle00] Bernd u. Lisa Kleinjohann. Eingebettete Systeme. *Vorlesung, Universität Paderborn, Universität der Informationsgesellschaft*, WS 1999/2000, SS 2000.

[Kop95] Nossal R. Kopez, Herrmann. *The Cluster Compiler – A Tool for the Design of Time-Triggered Real-Time Systems*. LA Jolla, Clifornia, June 1995. Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems.

[Kop01] Herrmann Kopez. *Design Principles for Distributed Embedded Applications*. Kluwer Academic Publ., Boston [u.a.], fourth edition, 2001.

[Lap97] Phillip A. Laplante. *Real time systems design and analysis : an engineer's handbook*. IEEE Press [u.a.], Piscataway, NJ, 2 edition, 1997. Design and applications of real-time systems.

[Lük99] H.D. Lüke. *Signalübertragung*. Springer Verlag, Berlin Heidelberg, seventh edition, 1999.

[LSU89] Roger Lipsett, Carl Schaefer, and Cary Ussery. *Hardware Description and Design*. Kluwer Academic Publishers, Boston/Dordrecht/London, first edition, 1989.

[Mag92] A. Mag. Entwicklung eines Versuches für ein Prozesstechnikpraktikum, der Grundlagen, eine spezielle Benutzeroberfläche, ein Teach-In und eine Tutorial-Shell für den Roboter enthalten soll. *Universität Paderborn, Computer Science Department*, 1992.

[Mäd99] A Mäder. *Guide to HDL Coding Styles for Synthesis.* Universität Hamburg – Fachbereich Informatik, Hambug, Germany, 1999. Arbeitsbereich Technische Grundlagen der Informatik.

[Mäd01a] A Mäder. *VHDL Kurzbeschreibung.* Universität Hamburg – Fachbereich Informatik, Hambug, Germany, 2001. Arbeitsbereich Technische Grundlagen der Informatik.

[Mäd01b] A Mäder. *VHDL-Synthese für das Werkzeug SYNOPSYS Design Analyzer.* Universität Hamburg – Fachbereich Informatik, Hambug, Germany, 2001. Arbeitsbereich Technische Grundlagen der Informatik.

[Mik94] T. Miki. The potential of photonic networks. IEEE Column Magazine Band 32, Dez. 1994.

[ML92] Stanley Mazor and Patricia Langstraat. *A Guide to VHDL.* Kluwer Academic Publishers, Berlin Heidelberg, first edition, 1992.

[Oes01] Bernd Oestereich. *Die UML Kurzreferenz für die Praxis : kurz, bündig, ballastfrei.* Oldenbourg, München [u.a.], 2001.

[Pro95] J.B. Proakis. *Digital Communications, Band 1.* McGraw-Hill, third edition, 1995.

[Ram01] Franz Rammig. RTOS, Real Time Operating Systems. *Vorlesung, Universität Paderborn, Universität der Informationsgesellschaft*, WS 2000/2001, SS 2001.

[Ric97] Frank Richter. *http://www.tu-chemnitz.de/urz/netz-kurs/node9.html.* MET, März 1997.

[Rob82] Eshed Robotec. *SCORBOT ER-III, Users's Manual.* Eshed Robotec, Tel-Aviv, Israel, 1982.

[RtL01] RtLinux. *Webpage zu RtLinux.* http://www.rtlinux.org, Okt 2001.

[Rub98] Alessandro Rubini. *Linux-Gerätetreibe.* O'Reilly, Cambridge [u.a.], 1. edition, 1998.

[Sch97] Georg Schnurer. Die Schatten kommen – Erstkontakt mit USB. page 292ff, Februar 1997.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real time object oriented modeling.* Wiley, New York [u.a.], 1994. Wiley professional computing.

[Sho85] Robyn E. Shotwell. *The Ethernet Sourcebook.* New York: North-Holland, 1985.

[Smi00] Douglas J. Smith. *HDL Chip Design.* Doone Publications, Madison, AL, USA, seventh edition, 2000. A Practical Guide For Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog.

[Syn98] Synopsys, Inc., Mountain View, CA. *VHDL Design Compiler (tm) Manual*, 1998-08 edition, 1998.

[Tan96] Andrew S. Tanenbaum. *Computernetzwerke*. Prentice Hall, Upper Saddle River, USA, 3rd edition, 1996.

[Tar91] Katie Tarnay. *Protokoll Specification and Testing*, volume 1. Plenum Press, New York and London, New York, U.S.A. and Budapest, Hungary, 1991. Central Research Institute for Physics in Budapest, Hungary.

[TH95] Klaus Ten Hagen. *Abstrakte Modellierung digitaler Schaltungen*. Springer-Verlag, Boston/Dordrecht/London, first edition, 1995.

[Tin95] K. Tindell. *Analysis of Hard Real-Time Communications*, volume 9. Realtime Systems, 1995. pp. 147-171.

[TTT01] TTTech. *Time-Triggered Technology*. http://www.tttech.com, 2001.

[Ver93] P. Verissimo. *Real-Time Communication* . Addison Wesley – ACM Press, Reading, Mass., 1993. In: S. Mullender (Ed.).

[Wil01] Phil Wilshire. *Papers about RtLinux*. http://www.realtimelinux.org, Okt 2001.

[Xil01] Xilinx Corporation. *Xilinx Field Programmable Gate Array Data Book*, 2001.

[Yod97] Victor Yodaiken. The RT-Linux approach to hard real-time. *Whitepaper, http://www.rtlinux.org/documents/papers/whitepaper.html*, 1997.

Abkürzungsverzeichnis

ACK Acknowledge

ALF Automatisches Line-Feed

API Application Programmer Interface

BG Bus-Guardian

CD Carrier Detect

CLB Complex Logic Block

CNI Communication Network Interface

CSR Control Status Register

CTS Clear To Send

CRC Cyclic Redundancy Checksum

CU Control Unit

DEE Datenendeinrichtung

DPRAM Dual Ported Random-Access Memory

DSR Data Set Ready

DÜE Datenübertragungseinrichtung

DTR Data Terminal Ready

E Empfänger

EIA Electronic Industry Association

ES Eingebettetes System

FPGA Field-Programmable Gate Array

FS Full-Speed

FSM Finite State Machine

FTU Fault Tolerant Unit

FW FireWire, IEEE 1398

GND Ground

GUI Graphical User Interface

GByte/s Gigabyte pro Sekunde

GPS Global Positioning System

HS High-Speed

HW Hardware

IEEE Institute of Electrical and Electronics Engineers

IFB Interface-Block

IP Intellectual Property

IR Infrarot

IRM Isochronous Resource Manager

LAN Local Area Network

LS Low-Speed

LSB Least Significant Bit

LWL Lichtwellenleiter

MByte/s Megabyte pro Sekunde

MEDL Message Descriptor List

MFM Modified Frequency Modulation

Modem Modulator / Demodulator

MSB Most Significant Bit

MSC Message Sequence Chart

NBW Non Blocking Write

NRZ Non Return to Zero

NRZI Non Return to Zero Invert

OSI/ISO Open Systems Interconnection / International Standards Organization

PE Paper Empty

PG Protocol Generator

PID Packet Identifier

RAM Random Access Memory

RDI Receive Data Interrupt

RI Ring Indikator

ROM Read Only Memory

RTOS Real Time Operating System

RTS Request To Send

RxD, RD Read Data

S Sender

SAP Service Access Points

SDL Specification and Description Language

SG Sequence Generator

SLCT Select

SOF Start-Of-Frame

SRU Smallest Replaceable Unit

SW Software

TCP/IP Transmission Control Protocol / Internet Protocol

TDG Timing Dependency Graph

TDMA Time Division Multiple Access

TP Twisted-Pair

TTA Time Triggered Architecture

TTL Transistor Transistor Logic

TTP Time Triggered Protocol

TxD, TD Transmit Data

UART Universal Asynchronous Receiver

UML Unified Modelling Language

USB Universal Serial Bus

UTP Unshielded Twisted-Pair

FPGA-basierte Fail-safe-Schnittstellen für eingebettete Systeme

Dipl.-Inf. Marcel Flade

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 automatisch Fail-Save Daten für die Komponenten mit harter Echtzeit zu generieren. Die Neuheit dieses Ansatzes besteht darin, das Fail-Save Verhalten nicht erst in der Endkomponente sondern bereits in der verbindenden Schnittstelle und damit transparent für die Komponente zu erzeugen.

Inhaltsverzeichnis

1 Einführung **149**
1.1 Aufgabenstellung . . . 149
1.2 Gliederung der Arbeit . . . 149

2 Grundlagen **151**
2.1 Das Modell des Interfaceblock (IFB) . . . 151
2.1.1 Die Kontrolleinheit (CU) . . . 152
2.1.2 Die Protokollhandler (PH) . . . 153
2.1.3 Der Sequenzhandler (SH) . . . 153
2.1.4 Der Modus . . . 153
2.2 Definition des Protokollbegriffs . . . 154
2.3 FPGA . . . 156
2.3.1 Konfigurierbare Logikblöcke . . . 157
2.3.2 IO-Blöcke . . . 157
2.3.3 Verbindungsstruktur . . . 158

3 Fail-safe-Verhalten **161**
3.1 Einordnung und Bedeutung . . . 161
3.2 Arten von Fehlern bei Schnittstellen . . . 163
3.3 Möglichkeiten der Fehlererkennung . . . 163
3.3.1 Parität . . . 164
3.3.2 Rechteckcode . . . 164
3.3.3 Hammingcodes . . . 165
3.3.4 Zyklische Codes (CRC) . . . 165
3.3.5 Frame-Check . . . 165
3.3.6 ACK-Fehler Erkennung . . . 166
3.3.7 Monitoring . . . 166
3.3.8 Packet Identifier Check . . . 166
3.3.9 Message Descriptor List (MEDL) . . . 166
3.4 Fehlererkennungsmechanismen verschiedener Schnittstellen . . . 167
3.4.1 RS-232 . . . 167
3.4.2 RS-485 Schnittstelle . . . 167
3.4.3 Enhanced Parallel Port (EPP) . . . 167
3.4.4 USB . . . 168

3.4.5 Firewire . . . 169
3.4.6 TTP/C . . . 170
3.4.7 TTP/A . . . 172
3.4.8 LVDS . . . 173
3.4.9 Controller Area Network (CAN) . . . 173
3.4.10 Ethernet . . . 174
3.5 Integration von Fail-safe-Verhalten in Schnittstellen . . . 175
3.6 Demonstrator . . . 179
3.6.1 Bedeutung . . . 179
3.6.2 Aufbau und Funktionsweise . . . 179
3.6.3 Implementierungskonzepte . . . 180
3.6.4 Implementierung . . . 182
3.7 Zusammenfassung und Ausblick . . . 192

Literaturverzeichnis **193**

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 1.2. Kapitel enthält die Aufgabenstellung und einen kurzen Überblick über die Gliederung der Arbeit. Kapitel 2.1.4 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 3.6 stellt die Implementierung eines Demonstrators vor, in welchem die in dieser Arbeit vorgestellten Konzepte und Ergebnisse kombiniert und validiert werden. Das 3.7. 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.

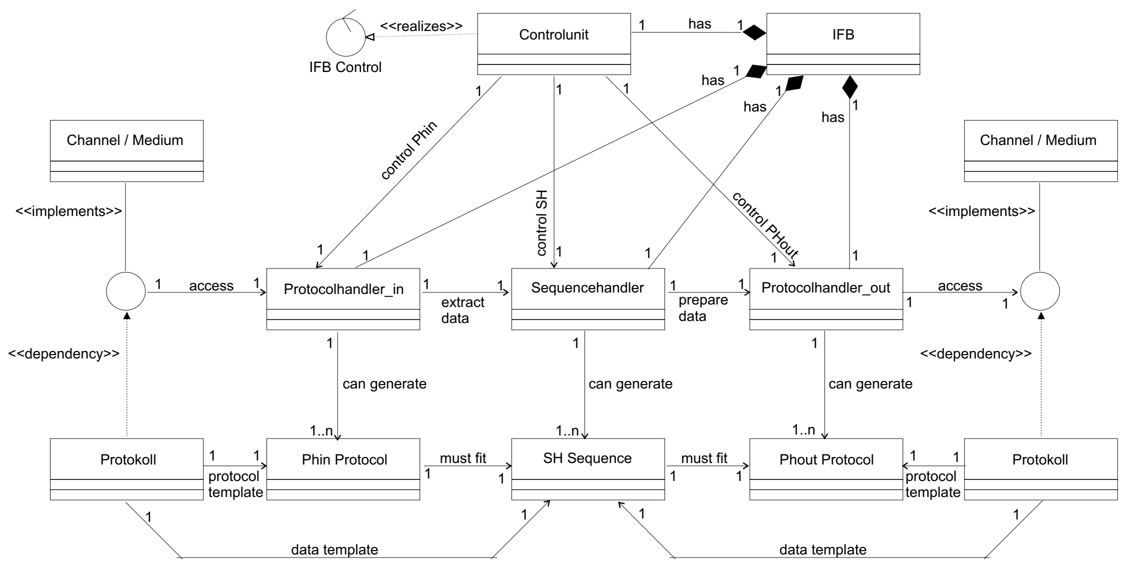


Abbildung 2.1: Klassendiagramm des Interfaceblocks

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.

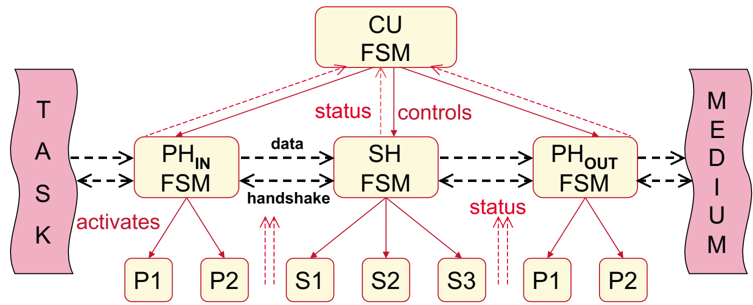


Abbildung 2.2: Makrostruktur des Interfaceblocks

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 Protokoll-

handler und des Sequenzhandlers. Dazu bestimmt 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 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

Abbildung 2.3: Das ISO-OSI-Referenzmodell

Schicht weiter.

Für die reale Kommunikation kann auch ein Interfaceblock (vgl. Abschnitt ??) 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 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

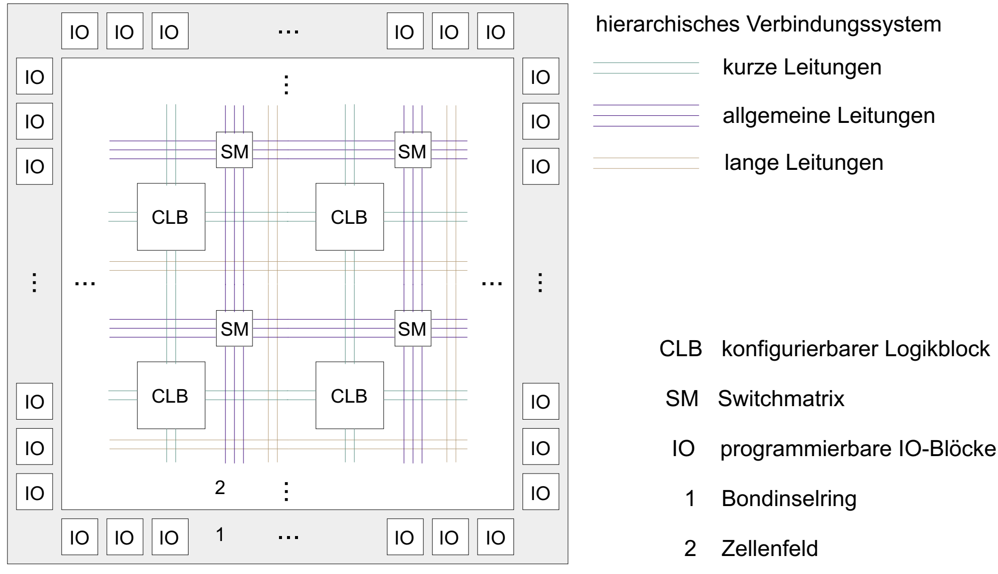


Abbildung 2.4: Der allgemeine Aufbau einer FPGA

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 3.6 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.

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.

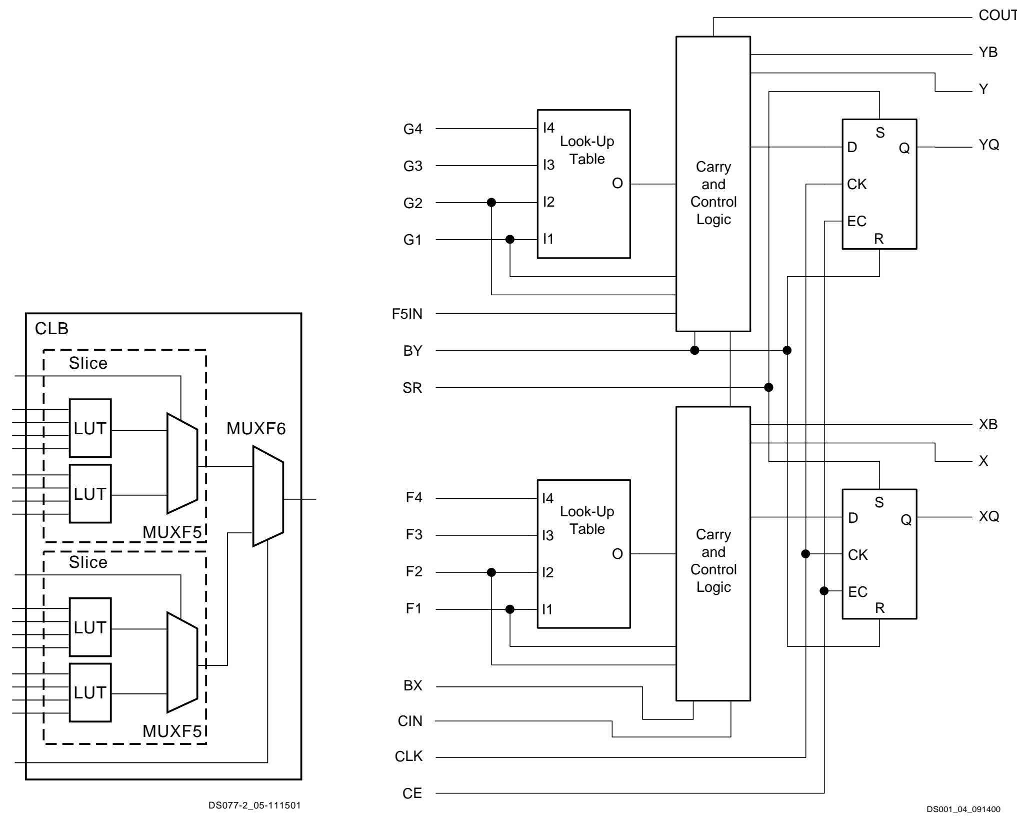


Abbildung 2.5: Konfigurierbarer Logikblock einer Xilinx Spartan 2E FPGA, Quelle [Xil03a]

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 be-

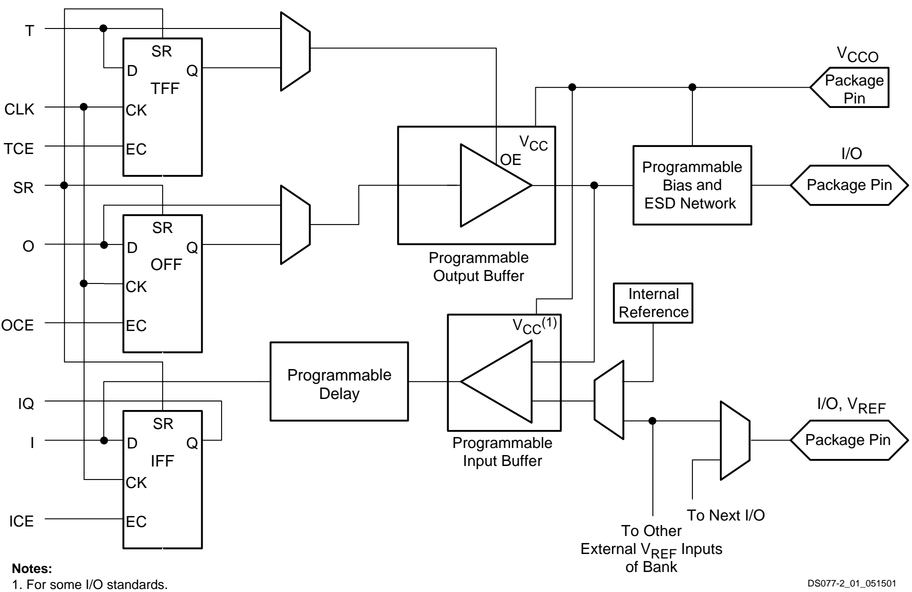


Abbildung 2.6: IO-Block einer Xilinx Spartan 2E FPGA, Quelle [Xil03a]

nachbarten 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 **??** 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 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

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]

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 Erweiterung 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.

Fehler / Mechanismus	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

3.3.1 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.

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 $2 \cdot n$ 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.

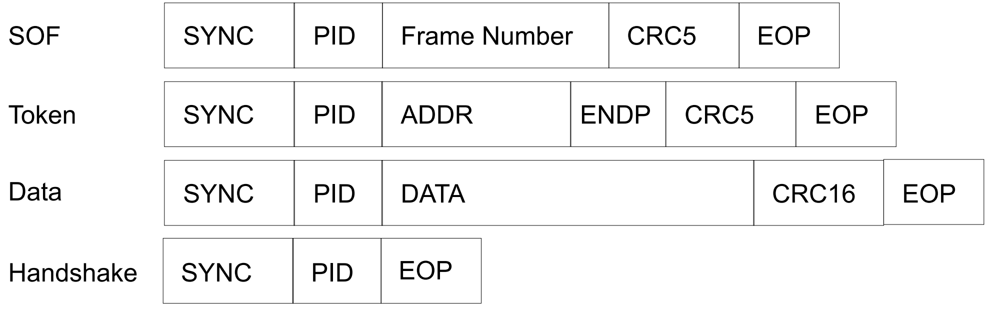


Abbildung 3.1: Aufbau der USB-Pakete

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.

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

PID 0	PID 1	PID 2	PID 3	$\overline{\text{PID 0}}$	$\overline{\text{PID 1}}$	$\overline{\text{PID 2}}$	$\overline{\text{PID 3}}$

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

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.

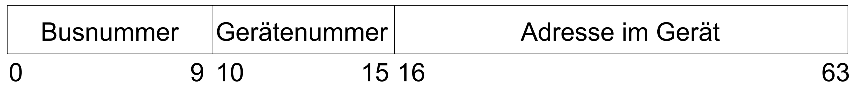


Abbildung 3.3: Aufbau der Adresse von Firewire

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. An-

schließend 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).

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

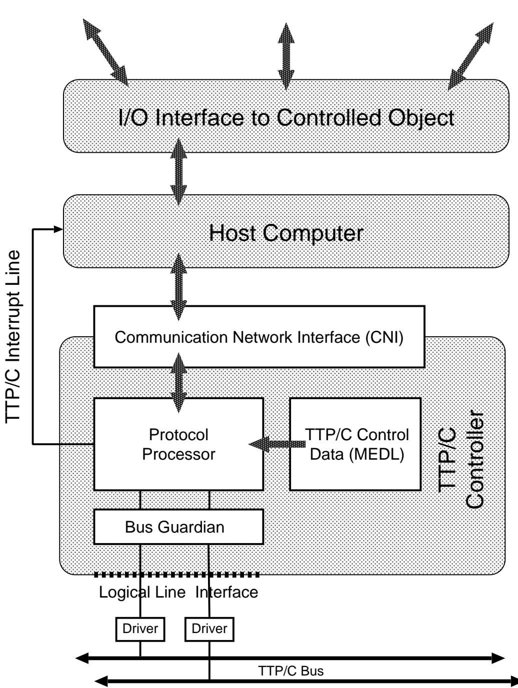


Abbildung 3.4: Aufbau einer TTP/C Node, Quelle [AG02]

Zeit	Adresse	Attribute 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]

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.

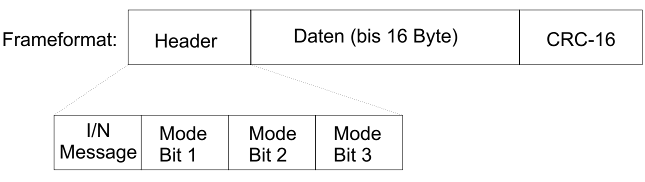


Abbildung 3.6: TTP/C Frameformat, Quelle [Kop97]

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

CRC-Berechnung beim Sender:

Header	Datenfeld	C-State des Sender	CRC

Nachricht auf dem Bus:

Header	Datenfeld	CRC

CRC-Berechnung beim Empfänger:

Header	Datenfeld	C-State des Empfängers	CRC

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

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 Nachricht, 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

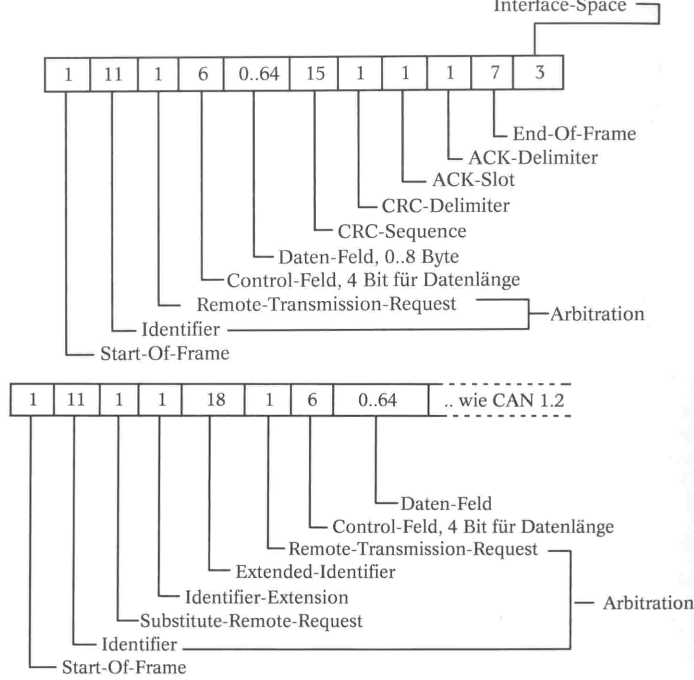


Abbildung 3.8: Datenrahmen für CAN 1.0 und 2.0, Quelle [Dem93]

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 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.

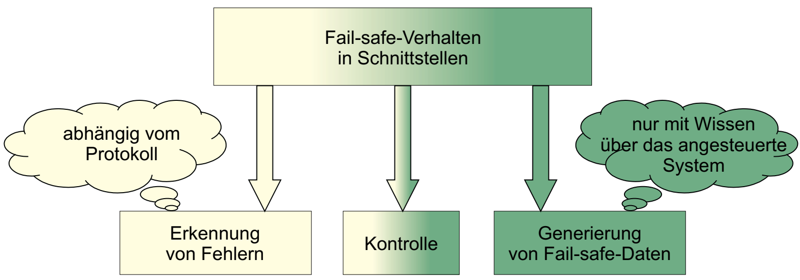


Abbildung 3.9: Aufgaben zur Erfüllung von Fail-safe-Verhalten in Schnittstellen

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 **??** 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.

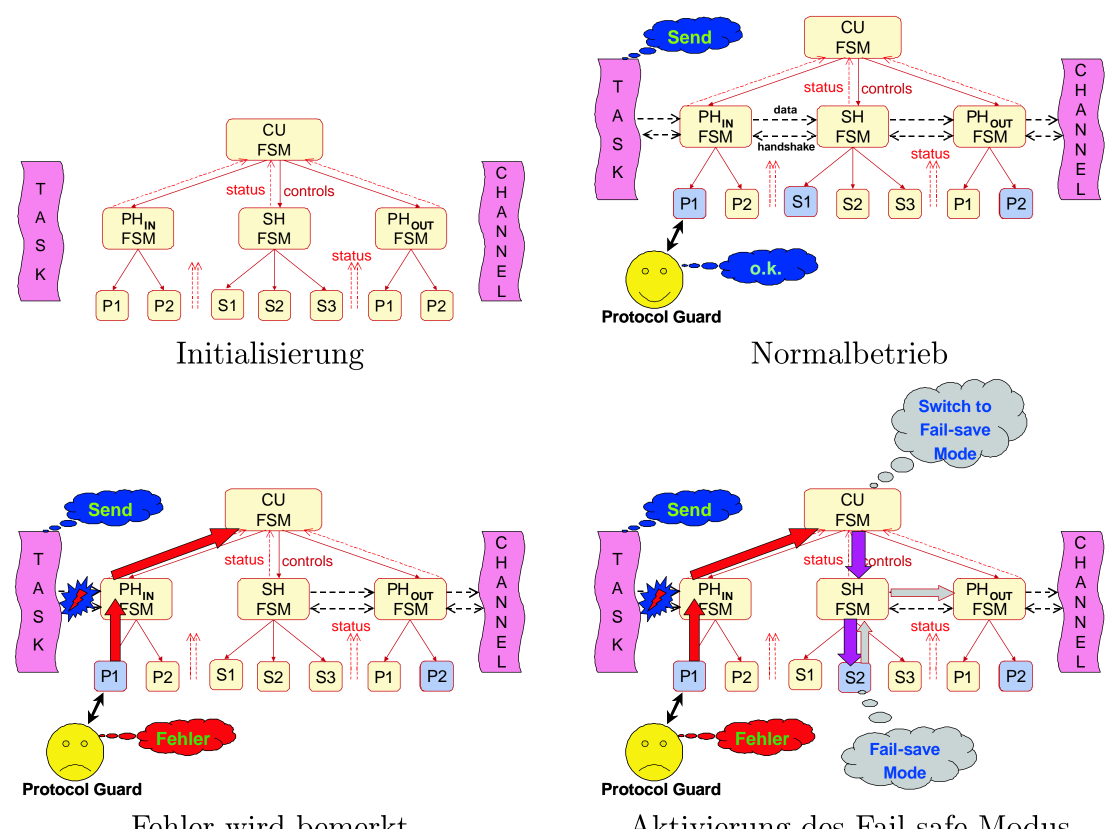


Abbildung 3.10: Fail-safe-Verhalten in einem Interfaceblock

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 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. Zusätzlich 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.

3.6 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.

3.6.1 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.

3.6.2 Aufbau und Funktionsweise

Der Demonstrator (Abbildung 3.11) 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 3.12 veranschaulicht die Freiheitsgrade des Roboters und stellt dar, durch welche Daten diese gesteuert werden können.

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.

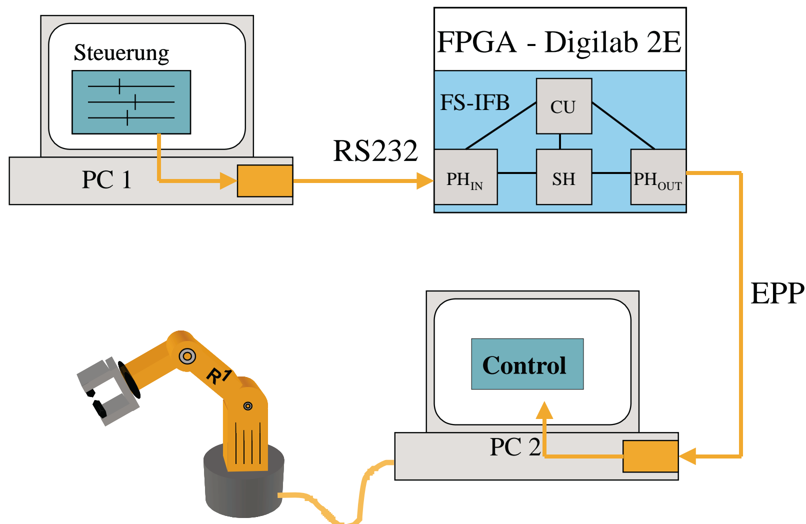


Abbildung 3.11: Der Aufbau des Demonstrators

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

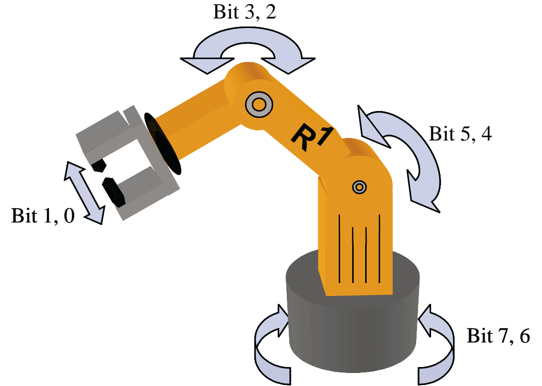
- PH_{in} im FPGA, Empfang von seriellen Daten über RS-232-Schnittstelle
- PH_{out} im FPGA, Senden von parallelen Daten über parallele Schnittstelle mit EPP-Protokoll
- Normalmodus im FPGA, Parallelisierung der seriellen Eingangsdaten
- Fail-safe-Modus im FPGA, Erzeugung des Nullvektors
- Kontrolleinheit 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.

3.6.3 Implementierungskonzepte

3.6.3.1 Modellierung von Fail-safe-Verhalten

Zur Beschreibung von Fail-safe-Verhalten für konkrete Anwendungen benötigt man eine Modellierungssprache. Ein adäquates Beschreibungsmittel hierfür stellen endliche Auto-



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

Abbildung 3.12: Funktionen des Funktionen

maten (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.

3.6.3.2 Konstruktion des Interfaceblocks

Der Interfaceblock (IFB) aus Kapitel 2.1.4 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 3.13) notwendig. Dazu gehören die Fragen:

- Wie beschreibe ich die Muster?
- Welche Eingangssignale werden gebraucht?

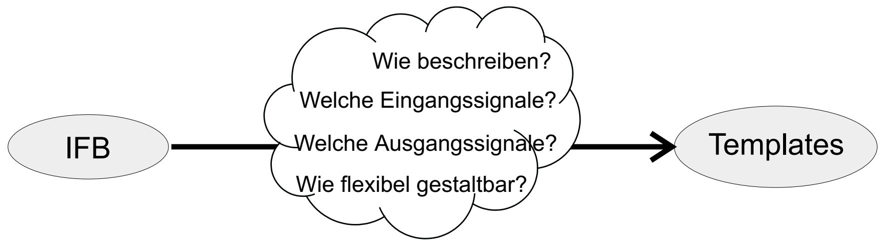


Abbildung 3.13: Vom Modell zum Template des IFB

- 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.

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.

3.6.4 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

3.6.4.1 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 (PH$_{in}$)**, den **Protokollhandler für ausgehende Signale (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.

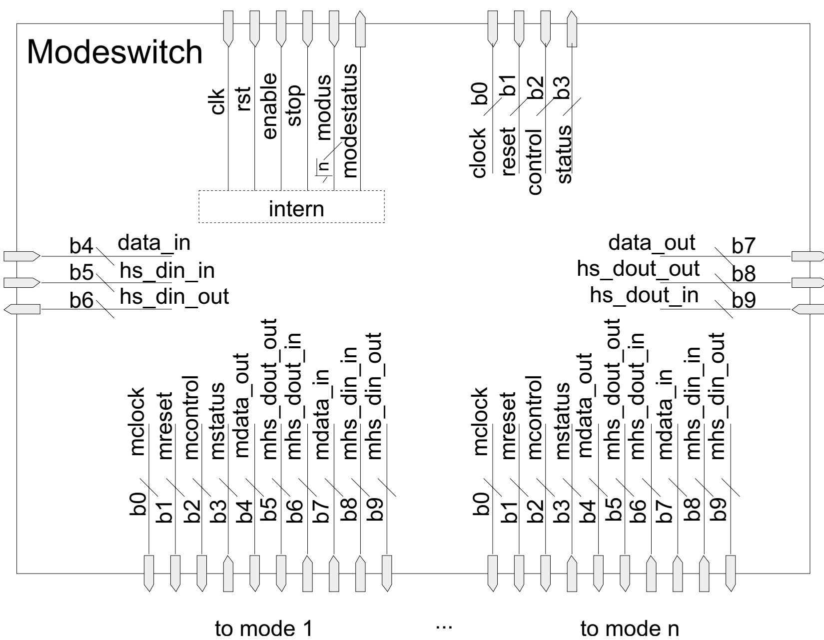


Abbildung 3.14: Aufbau des Modeswitch

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 3.14). 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.

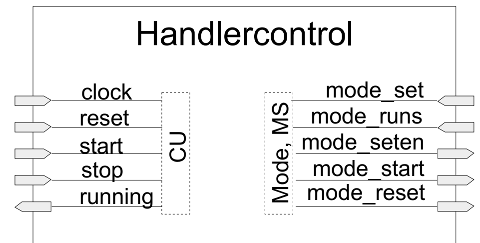


Abbildung 3.15: Ports der Handlercontrol

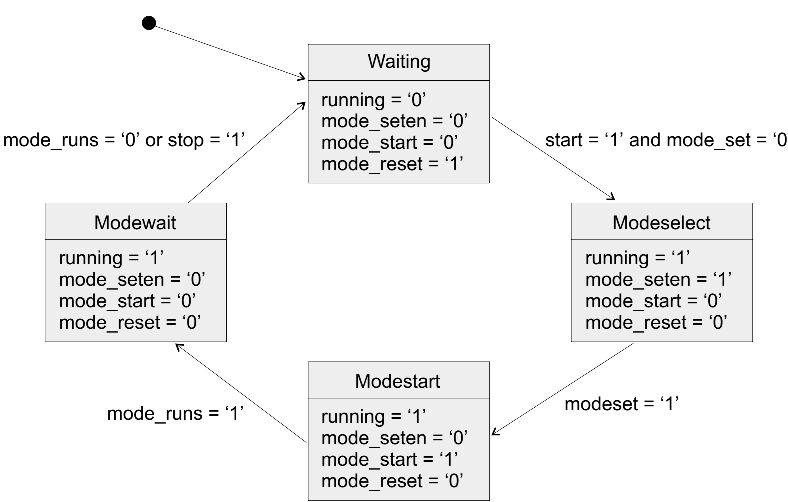


Abbildung 3.16: Automat zur Kontrolle der Handlerfunktionen

Um die Vorgänge im Handler zu steuern wurde die Komponente **Handlercontrol** entwickelt, deren Aufbau in Abbildung 3.15 veranschaulicht ist. Sie besteht aus einem Automaten (Abbildung 3.16), 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.

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

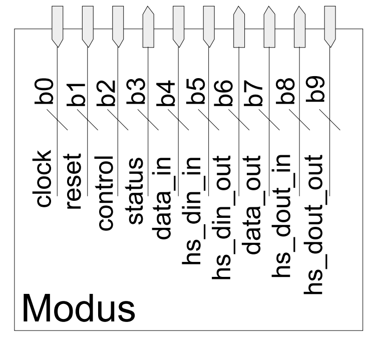


Abbildung 3.17: Ports des Modus

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.

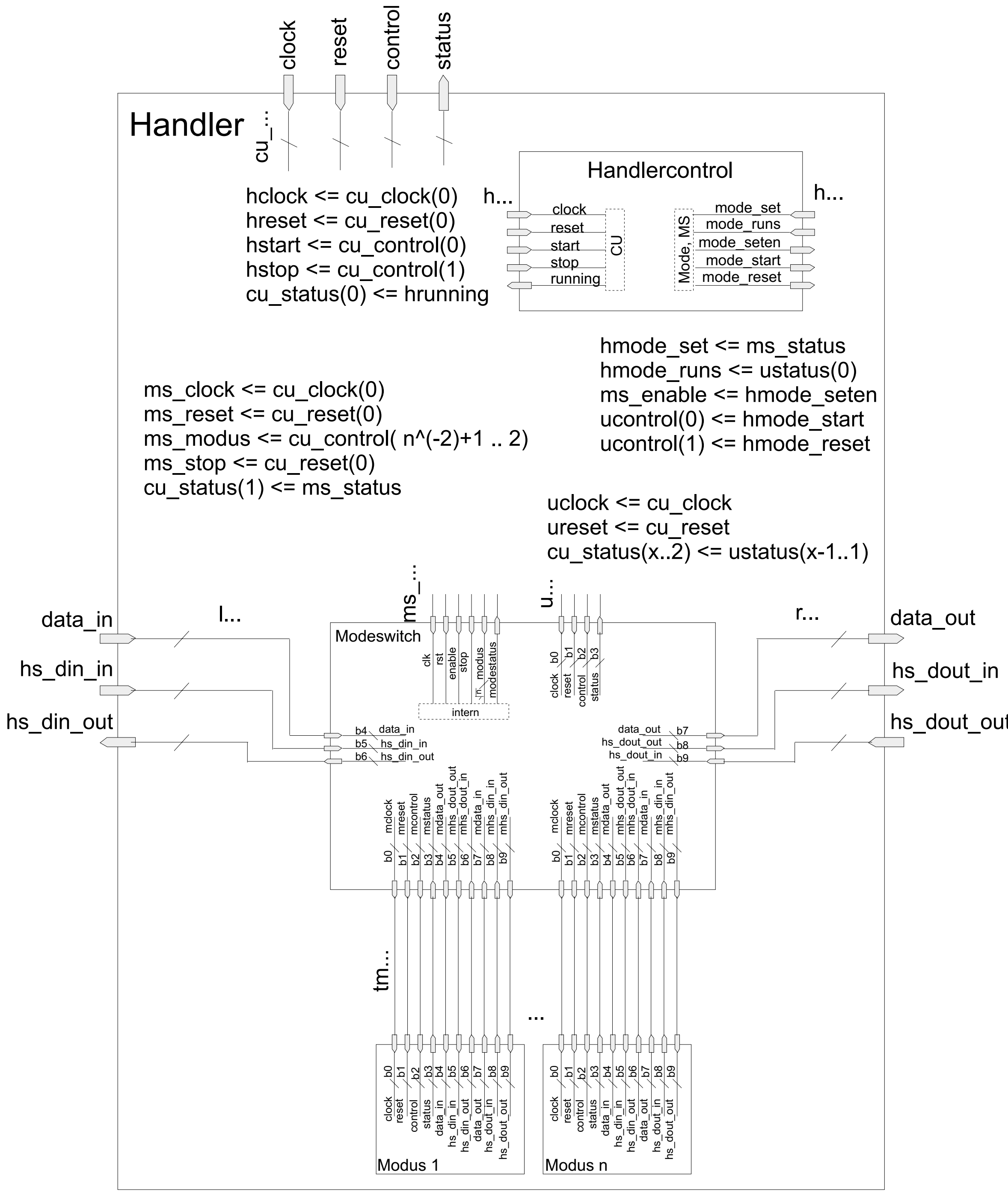


Abbildung 3.18: Aufbau des Handlers. Internen Signalen sind die Buchstaben mit drei Punkten vorangestellt (z.B. u... bedeutet uclock, ureset, ...). Auf die Verdrahtung im oberen Teil wurde aus Gründen der Übersichtlichkeit verzichtet und dafür die Abildungsvorschriften der Signale aufgeführt.

3.6.4.2 Schnittstellen und ihre Verbindung

EPP

Der Kommunikationsautomat für das EPP-Protokoll wurde anhand von Timingdiagrammen aus [Axe97] erstellt. Er ist in Abbildung 3.19 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 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

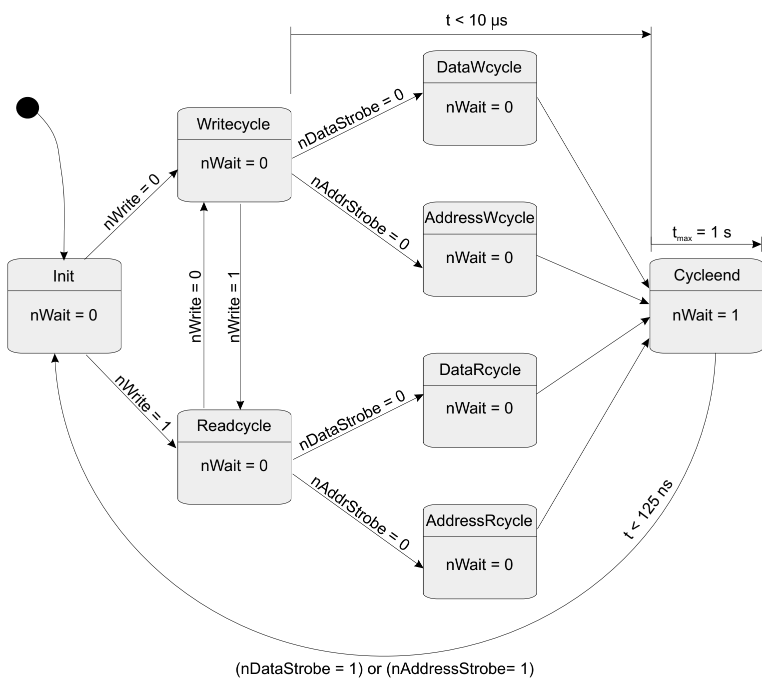


Abbildung 3.19: Kommunikationsautomat des EPP-Protokolls

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 3.20) 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 3.20 der 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.

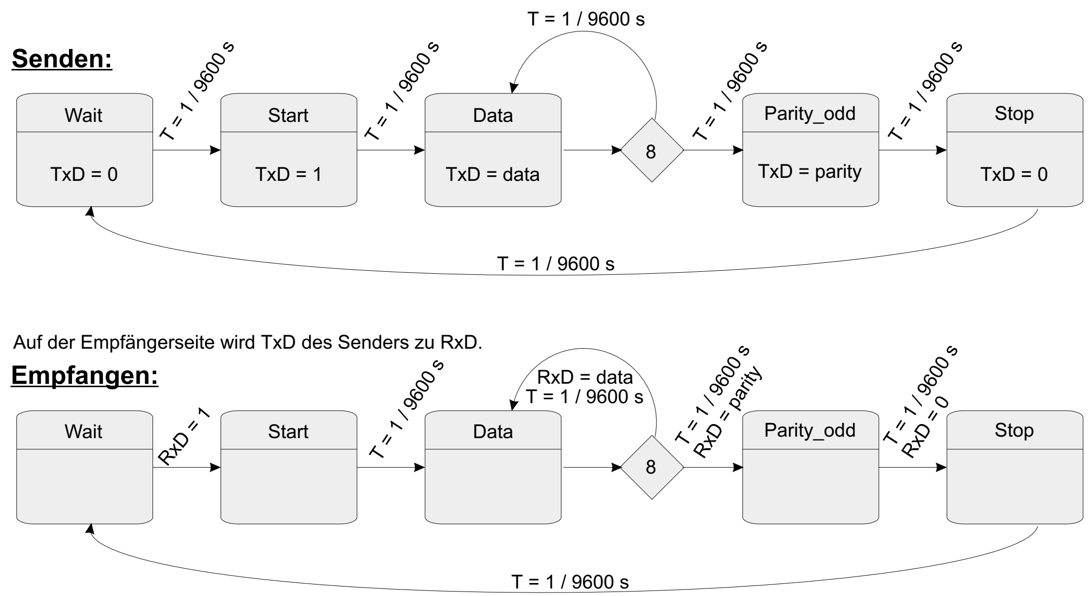


Abbildung 3.20: Kommunikationsautomat der RS-232-Schnittstelle

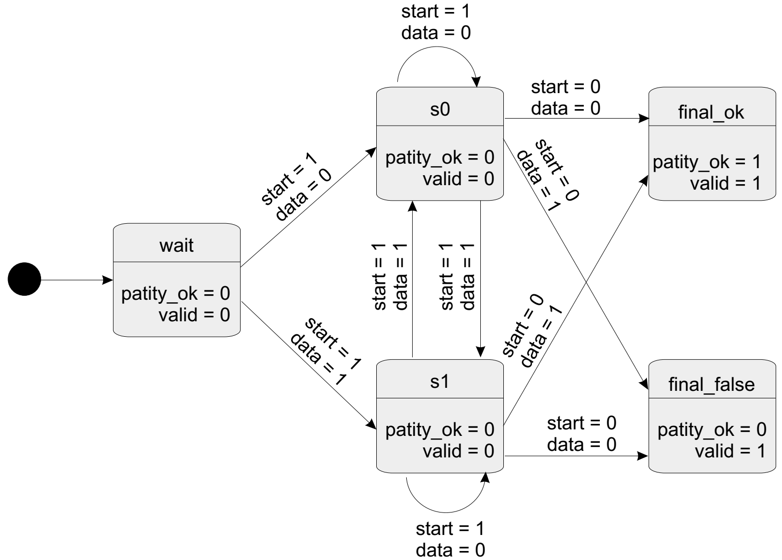


Abbildung 3.21: Automat zur Paritätsprüfung

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.

3.6.4.3 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 3.21 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 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 Zu-

stand 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 3.12).

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 3.22 zeigt diesen Automaten.

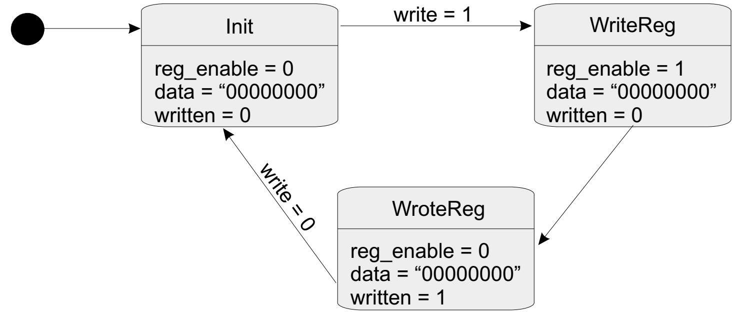


Abbildung 3.22: Automat für den Fail-safe-Datengenerator

Kontrolleinheit

Die Kontrolleinheit setzt sich aus drei Automaten (Abbildung 3.23, 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

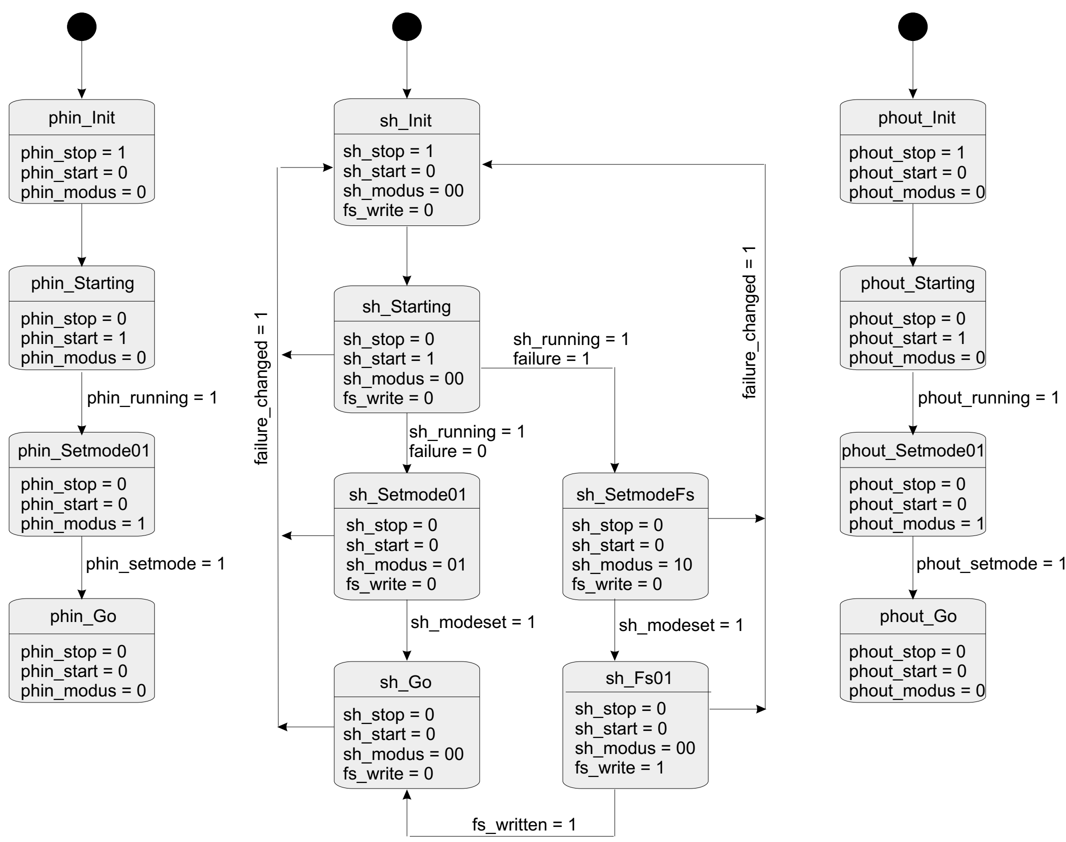


Abbildung 3.23: Automaten der Kontrolleinheit

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 3.23, mittlerer 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.

3.7 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 Hauptteil 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.

Literaturverzeichnis

[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.

Automatische Adaption von Hardware-Acceleratoren für Verhaltenssimulation

Dipl.-Inf. Marcel Flade

Der Entwurf von Hardware-Software-Systemen wurde bisher dadurch erschwert, dass eine Lücke im Entwurfsprozess zwischen der System-Level-Beschreibung und der Hardwarebeschreibung existierte. Um diese Lücke zu schließen, wurde SystemC entwickelt. Jedoch ist ein Nachteil dieser Systementwurfsprache, dass man bisher entwickelte Komponenten in VHDL oder Verilog, also so genannte Intellectual Properties nicht einfach in SystemC einbinden kann. Für die Simulation muss erst ein SystemC Modell der Komponente erstellt werden. Das bedeutet zusätzliche Entwicklungsarbeit und damit höhere Entwurfskosten. In diesem Beitrag wird ein Co-Simulationsansatz auf Systemebene vorgestellt. Dabei wird eine bereits synthesefähige VHDL-Komponente auf einem FPGA abgearbeitet und mit der Simulation der restlichen SystemC-Komponenten gekoppelt. Dies ermöglicht die Reduzierung des Entwurfsaufwands. Zur einfacheren und schnelleren Adaption des FPGA an die SystemC-Simulation, soll ein automatisiertes Verfahren entwickelt werden. Der Vorteil eines automatisierten Verfahrens besteht darin, Zeit und Kosten bei der Vorbereitung der Simulation zu sparen.

Inhaltsverzeichnis

1. Einführung **199**

1.1. Aufgabenstellung . . . 202
1.2. Gliederung der Arbeit . . . 203

2. Stand der Technik **205**

2.1. SystemC . . . 205
2.2. Intellectual Properties . . . 207
2.3. Simulation . . . 210
2.3.1. Simulation mit SystemC . . . 212
2.3.2. Hardwaresimulation mit ModelSim . . . 213
2.3.3. HW/SW-Cosimulation . . . 213
2.3.4. Hardware-Akzeleratoren . . . 217
2.4. Fazit . . . 221

3. Die Methode der Adaptierung **223**

3.1. Das Hardware/Software-Interface . . . 223
3.1.1. Anforderungsanalyse für das HW/SW-Interface . . . 226
3.2. Der Interfaceblock als HW/SW-Interface . . . 229
3.2.1. Das Modell des Interfaceblocks . . . 229
3.2.2. Analyse der Leistung des Interfaceblocks . . . 231
3.2.3. Erweiterung des Interfaceblocks . . . 232
3.3. Technische Umsetzung der Adaptierung durch einen Simulationsinterfaceblock . . . 236
3.3.1. Die hardwareseitige Implementierungsplattform . . . 236
3.3.2. Umsetzung des HW/SW-Interfaces . . . 237
3.3.3. Der PH_{SW} . . . 240
3.3.4. Der $PH_{HW\text{-}in}$ . . . 242
3.3.5. Der SH_{HW} . . . 244
3.3.6. Der $PH_{HW\text{-}out}$ . . . 245
3.3.7. Die Controlunit . . . 246
3.4. Anwendungsmöglichkeiten . . . 250

4. Der Simulationsinterfaceblock-Generator 253
4.1. Motivation . . . 253
4.2. Arbeitssweise . . . 254
4.2.1. Eingaben und Ausgaben . . . 254
4.2.2. Die internen Datenstrukturen . . . 254
4.2.3. Der Programmablauf . . . 260
4.3. Benutzung des Programms . . . 264
4.3.1. Programmstart und -aufbau . . . 264
4.3.2. Die Analyse der Quelldatei . . . 265
4.3.3. Festlegung von Signalparametern . . . 265
4.3.4. Weitere Parameter . . . 266
4.3.5. Die Generierung . . . 268
4.3.6. Überblick über die generierten Dateien . . . 269
4.4. Verbesserungsmöglichkeiten und Fazit . . . 273

5. Demonstrator 275
5.1. Bedeutung eines Demonstrators . . . 275
5.2. Aufbau und Funktionsweise . . . 275
5.3. Nachweis der Funktion . . . 278
5.4. Betrachtungen zur Simulationszeit . . . 281
5.4.1. Geschwindigkeit der parallelen Schnittstelle . . . 281
5.4.2. Vergleich der Simulationszeiten . . . 283
5.5. Schlussfolgerungen . . . 287

6. Zusammenfassung und Ausblick 289
6.1. Zusammenfassung der Arbeit . . . 289
6.2. Ausblick für den erweiterten Interfaceblock . . . 291
6.3. Ausblick für den Simulationsinterfaceblock-Generator . . . 292

A. Herleitung der Formel zur Berechnung der Taktfrequenz der IP-Emulation 293

B. Hinweise zur Nutzung von SystemC in Mircrosoft Visual C++ 6.0 295

Literaturverzeichnis 297

Abkürzungsverzeichnis 303

1. Einführung

Die Entwicklung eines digitalen technischen Systems unterliegt heutzutage hohen Anforderungen hinsichtlich Entwicklungszeit und Entwicklungskosten. Der Gewinn, den ein Produkt erzielt, fällt umso größer aus, je schneller es auf dem Markt verfügbar ist. Das Ziel des Entwicklungsprozesses liegt demzufolge darin, ein Produkt schnell und mit möglichst geringen Entwicklungskosten auf den Markt zu bringen.

Ein Schritt in diese Richtung stellt ein modulares Design dar. Es folgt dem Grundsatz, dass ein System aus verschiedenen Modulen aufgebaut ist (vgl. Abbildung 1.0.1). Die Entwicklung und der Test der einzelnen Module kann unabhängig voneinander erfolgen. Somit kann der Entwicklungsaufwand auf mehrere Personen oder Teams verteilt werden und die Entwicklungszeit reduziert sich durch die Parallelisierung der Entwurfsarbeit. Das Gesamtsystem ergibt sich aus der Zusammenführung und Kopplung der einzelnen Module.

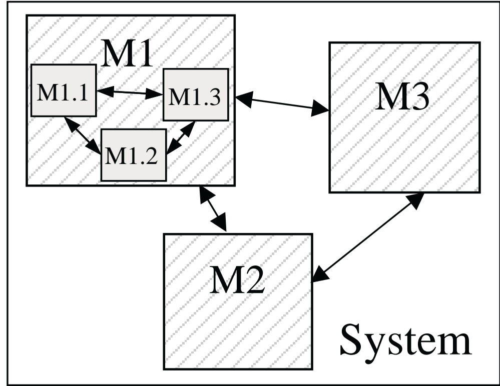


Abbildung 1.0.1.: Schema eines modularen Designs

Einen weiteren Vorteil dieses Konzeptes stellt die Wiederverwendung (Reuse) von bereits entwickelten Modulen in neuen Systemen dar. Diese Intellectual Properties (IPs, vgl. Abschnitt 2.2) verkürzen den Entwicklungsaufwand für neue Systeme erheblich, da diese Module nicht neu entwickelt werden müssen. Im Idealfall können sie ohne Änderungen in das Design eingebunden werden.

Ein aktuelles Problem mit dem die Entwickler konfrontiert sind, bildet eine Lücke im Entwurfsprozess. Abbildung 1.0.2 stellt diesen Sachverhalt dar. Die Entwicklung eines

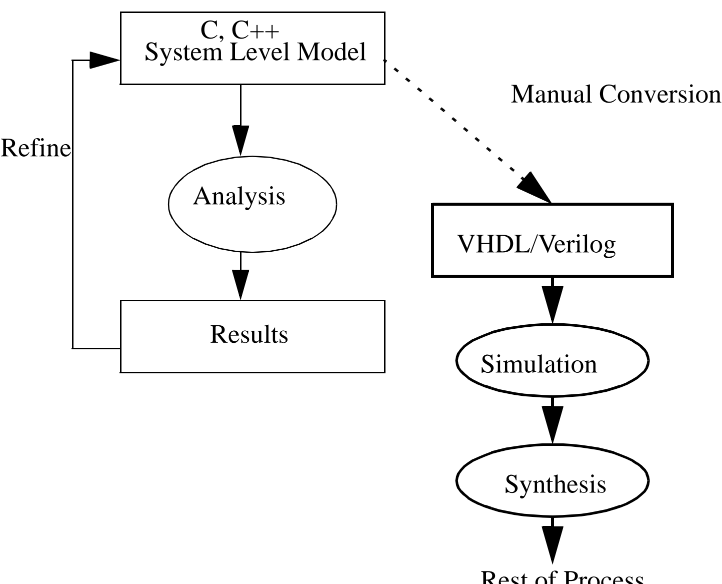


Abbildung 1.0.2.: Aktuelle Methode des Hardwareentwurfsprozesses, Quelle [Opeb]

Systems beginnt mit einer Spezifikation auf Systemebene. Die dafür verwendete Sprache ist meist C oder C++. Die Spezifikation wird schrittweise analysiert und verfeinert. Dabei durchläuft die Systementwicklung verschiedene Abstraktionsebenen des Entwurfsprozesses (vgl. Abbildung 1.0.3 und 1.0.4). Das Ergebnis eines erfolgreichen Entwurfsprozesses besteht in der Implementierung des Systems, die wie in der Spezifikation festgelegt arbeitet.

Ab einem Verfeinerungsgrad, der einer Verhaltensbeschreibung entspricht, reichen die bestehenden Beschreibungsmöglichkeiten von C/C++ nicht mehr aus und es muss auf eine Hardwarebeschreibungssprache, wie zum Beispiel VHDL oder Verilog, übergegangen werden. Dieser Schritt stellt beim aktuellen Entwurfsprozess noch ein Hindernis dar, da er vom Entwickler von Hand durchgeführt werden muss. Dieser Prozess beansprucht viel Zeit und ist fehleranfällig. Nach der Konvertierung muss das Design ausgiebigen Tests unterzogen werden, um sicherzustellen, dass die Konvertierung mit der Spezifikation übereinstimmt. Dadurch verlängert sich die Entwicklungszeit und teure Resourcen werden gebunden.

Wünschenswert ist nun ein Entwurfssystem, das es erlaubt, den Entwurf von Hardware über alle Entwurfsebenen modular und in einer einheitlichen Beschreibungssprache durchzuführen. Für den zunehmend wichtiger werdenden Entwurf von eingebetteten Systemen sollte das Entwurfssystem zudem die Unterstützung der dafür benötigten Entwurfsdimensionen (vgl. Abbildung 1.0.5) gewährleisten. Außerdem sollte es die Verwendung von Intellectual Properties auf jeder Verfeinerungsebene für den Entwurf und den Test des Systems unterstützen.

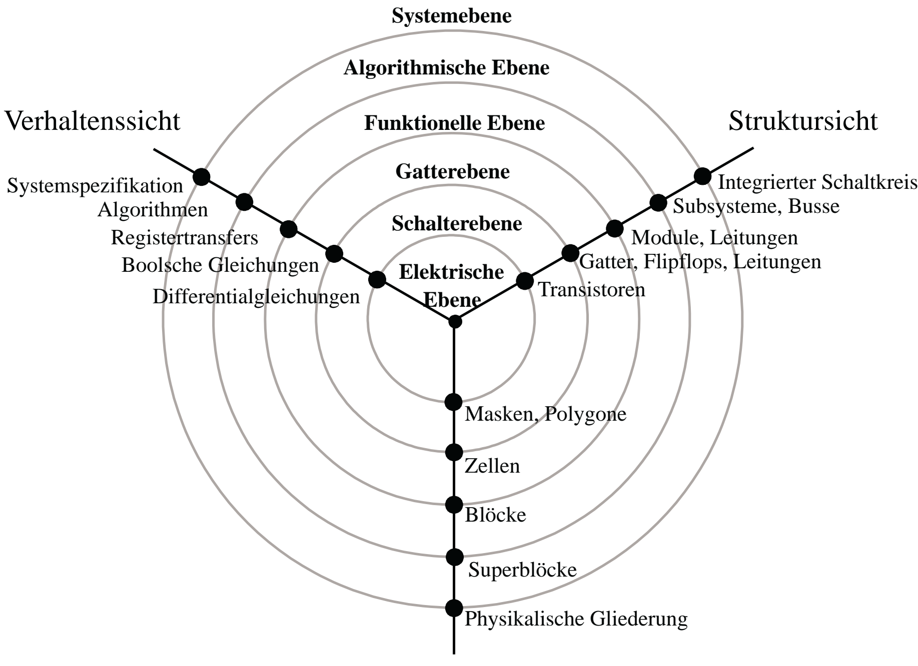


Abbildung 1.0.3.: Das Y-Diagramm nach Gajski. Es zeigt die verschiedenen Sichten und Abstraktionsebenen beim Hardwareentwurf.

Um diese Anforderungen zu erfüllen, wurde SystemC (Abschnitt 2.1) entwickelt. SystemC unterstützt den modularen Entwurfsprozess von der Systemspezifikation bis zur Register-Transfer-Ebene. Außerdem erlaubt es Hardware/Software-Codesign, was für die Entwicklung von HW/SW-Systemen vorteilhaft ist. Die Einbindung von sprachfremden Intellectual Properties ist als Netzliste möglich. Eine Netzliste ist eine synthetisierte Register-Transfer-Beschreibung. In höheren Abstraktionsebenen können diese Intellectual Properties nicht in ihrer Originalform verwendet werden. Zur Simulation eines Systems in diesen Abstraktionsebenen muss ein Modell der IP erstellt werden. Dieser Prozess ist fehleranfällig und kostet Zeit und Geld, was den Erfolg des Produktes weniger gut ausfallen lassen kann.

Die Umgehung der Erstellung eines Modells der Intellectual Property könnte den Entwurfsprozess weiter beschleunigen. Jedoch muss dazu eine Lösung gefunden werden, um die originalen IPs bereits in die Simulation höherer Abstraktionsebenen einzubinden. Die Entwicklung einer solchen Lösung ist Ziel der vorliegenden Arbeit.

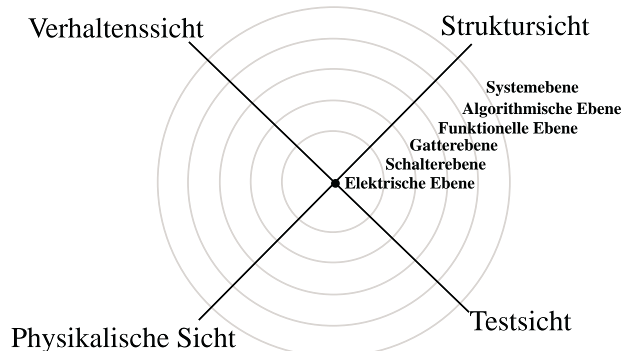


Abbildung 1.0.4.: Das X-Diagramm nach Rammig. Gegenüber dem Y-Diagramm nach Gajski bringt es die Testsicht zusätzlich in jede Entwurfsebene mit ein.

1.1. Aufgabenstellung

Der Entwurf von Hardware-Software-Systemen wurde bisher dadurch erschwert, dass eine Lücke im Entwurfsprozess zwischen der System-Level-Beschreibung und der Hardwarebeschreibung existierte. Um diese Lücke zu schließen, wurde SystemC entwickelt.

Jedoch ist ein Nachteil dieser Systementwurfsprache, dass man bisher entwickelte Komponenten in VHDL oder Verilog, so genannte Intellectual Properties, erst auf Netzlistenebene in SystemC einbinden kann. Für die Simulation von höheren Abstraktionsebenen muss erst ein SystemC Modell der Komponente erstellt werden. Das bedeutet zusätzliche

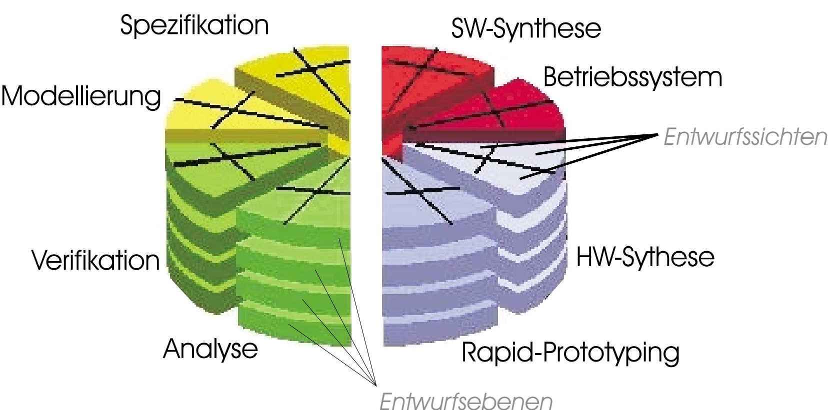


Abbildung 1.0.5.: Die P-Chart Entwurfsstrukturierung nach [Har02] für den Entwurf von HW/SW-Systemen

Entwicklungsarbeit und damit höhere Entwurfskosten.

In dieser Arbeit soll ein Co-Simulationsansatz auf Systemebene untersucht werden. Dabei soll die bereits synthesefähige VHDL-Komponente auf einem FPGA abgearbeitet werden und mit der Simulation der restlichen SystemC-Komponenten gekoppelt werden. Der Entwurfsaufwand reduziert sich und das Verhalten des Systems mit der realen Intellectual Property kann untersucht werden.

Zur einfacheren und schnelleren Adaptierung des FPGA an die SystemC-Simulation, soll ein automatisiertes Verfahren entwickelt werden. Der Vorteil eines automatisierten Verfahrens besteht darin, Zeit und Kosten bei der Vorbereitung der Simulation zu sparen.

1.2. Gliederung der Arbeit

Das Thema dieser Arbeit ist die **Automatische Adaption[1] von Hardware-Acceleratoren für Verhaltenssimulation**. Das erste Kapitel enthält eine kurze Einführung in die Thematik und die Aufgabenstellung. Kapitel 2 beschreibt den aktuellen Stand von Techniken, auf die diese Arbeit aufbaut oder die mit dem Thema verwandt sind. Mit der Methode der Adaptierung von Hardwareakzeleratoren an die SystemC Verhaltenssimulation beschäftigt sich Kapitel 3. Ein allgemeines Lösungskonzept wird darin vorgestellt und darauf aufbauend eine technische Umsetzung geboten. Kapitel 4 beschreibt die Automatisierung des vorgestellten Lösungskonzeptes durch ein, im Rahmen dieser Arbeit entwickeltes, Programm. Der Nachweis über die korrekte Funktion der erarbeiteten Lösungen wird in Kapitel 5 an einem Beispiel durchgeführt. Das 6. Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf Zukunftsperspektiven und Möglichkeiten der Weiterentwicklung.

[1] Für den englischen Begriff Adaption wird im weiteren Verlauf dieser Arbeit der deutsche Begriff Adaptierung verwendet.

2. Stand der Technik

Dieses Kapitel beschäftigt sich mit dem aktuellen Stand der Techniken, auf die diese Arbeit aufbaut. Zunächst erfolgt eine Beschreibung der noch jungen Systementwurfssprache SystemC. Der zweite Teil des Abschnittes definiert den Begriff der Intellectual Properties und umreißt deren Bedeutung. Anschließend wird auf die Rolle der Simulation bei der Hardwareentwicklung eingegangen und einige Beispiele für Simulationswerkzeuge, darunter auch Hardwareakzeleratoren vorgestellt. Das Ende des Kapitels faßt die hier gewonnen Erkenntnisse zusammen und arbeitet noch einmal kurz die Vor- und Nachteile aktueller Techniken heraus.

2.1. SystemC

SystemC™[1] ist eine standardisierte Systementwurfs- und Verifikationssprache. Sie wurde im September 1999 eingeführt und durch die OSCI (Open SystemC™ Initative) standardisiert. Die OSCI ist ein Konsortium größerer EDA-Firmen und IP-Providern. Mitglieder dieses Konsortiums sind zum Beispiel ARM Ltd., Cadence Design Systems Inc., Synopsys Inc., Motorola, Panasonic und andere. Mehr Informationen zu den Mitgliedern des Konsortiums befinden sich auf der Homepage von SystemC [Opea].

[1] SystemC ist ein eingetragenes Warenzeichen der Open SystemC™Initiative.

Abbildung 2.1.1.: Logo von SystemC™, Quelle [Opea]

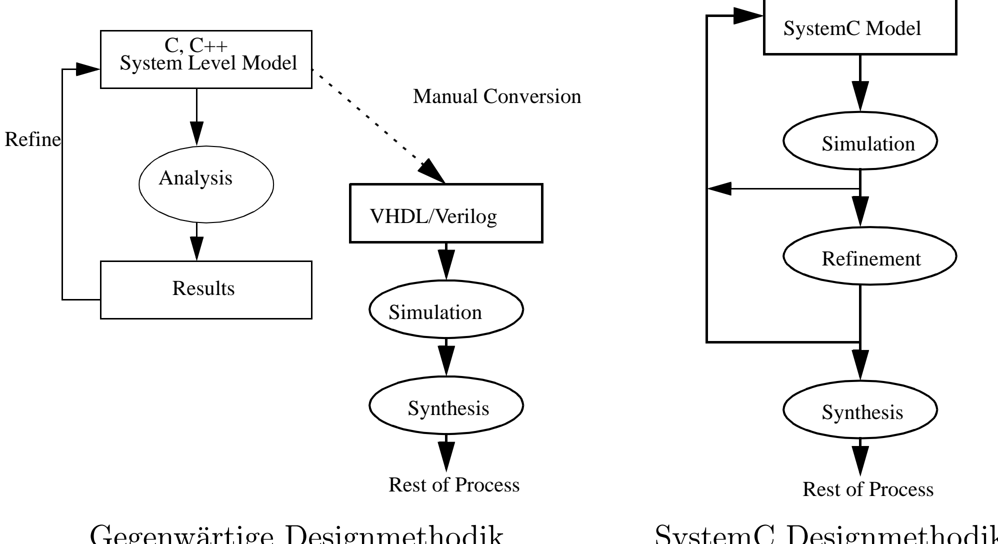


Abbildung 2.1.2.: Gegenwärtige und SystemC Designmethodik, Quelle [Opeb]

SystemC ist eine C++ Klassenbibliothek und nutzt somit die Möglichkeit, C++ durch Klassenbibliotheken zu erweitern ohne neue syntaktische Konstrukte hinzuzufügen. Der Entwickler kann somit die ihm vertraute Sprache C++ und die damit verbundenen Entwicklungswerkzeuge weiterhin verwenden. SystemC wird über die Open Source Lizenz [VA] vertrieben. Die Bibliothek kann somit kostenlos genutzt werden. Der Nutzer enthält mit der Bibliothek ein kostenloses Entwurfs- und Simulationspaket. Es enthält die notwendigen Konstrukte um Systemarchitekturen, inklusive Hardwaretiming, Nebenläufigkeit und reaktivem Verhalten, zu modellieren, die nicht in Standard C++ enthalten sind. Außerdem verfügt die Bibliothek bereits über einen Simulator mit dem die entworfenen Designs getestet werden können. Die Beschreibung des Simulators folgt in Abschnitt 2.3.1.

Mit SystemC kann ein Entwickler effektive zyklengenaue Modelle von Softwarealgorithmen, Hardwarearchitekturen und Schnittstellen eines System-on-Chip- (SoC) und System-Level-Designs beschreiben. Außerdem bietet es die Möglichkeit eine ausführbare Spezifikation zu erstellen. Auch Hardware/Software-Codesign, welches in heutigen Systemen eine immer größere Bedeutung erlangt, wird unterstützt.

SystemC unterstützt den Systementwurf von der Spezifikation bis zur Register-Transfer-Ebene. Bei bisher verwendeten Beschreibungsmöglichkeiten bestand eine Lücke im Entwurfsprozess. Zum Beispiel stehen in C/C++ keine geeigneten Beschreibungsmittel für Hardware zur Verfügung, da Konzepte, wie Nebenläufigkeit und ein Zeitmodell, fehlen. Deshalb war bisher eine manuelle Konvertierung von einem C/C++ Verhaltensmodell in eine, durch eine Hardwarebeschreibungssprache modellierte, Regsiter-Transfer-Beschreibung notwendig. Abbildung 2.1.2 stellt die bisher und auch gegenwärtig verwendete Designmethode und die Designmethodik von SystemC gegenüber.

Als junges Entwurfssystem gibt es allerdings noch Einschränkungen. So ist beim Softwaredesign die Einbindung von Echtzeitbetriebssystemen (RTOS) noch nicht möglich. Eine Verbesserung soll Version 3 von SystemC bringen.

Als Werkzeuge zur Synthese kommt der CoCentric® SystemC™ Compiler der Firma Synopsys® [Syn] zum Einsatz. Er erlaubt sowohl eine Synthese von der Register-Transfer-Ebene als auch von der Verhaltensebene. Der Compiler ist jedoch nicht in der kostenfreien Klassenbibliothek enthalten und muss gesondert gekauft werden. Mehr Informationen zum CoCentric® SystemC™ Compiler sind in [Syn03] zu finden.

2.2. Intellectual Properties

Gorden Moore formulierte 1965 das nach ihm benannte Mooresche Gesetz. Es besagt, dass sich die Anzahl der Transistoren auf einer gegebenen Fläche Silizium etwa alle 18 Monate verdoppelt. Abbildung 2.2.3 stellt diesen Sachverhalt grafisch dar. Dieses Gesetz hat bis heute noch Gültigkeit und wird nach Meinung von Experten auch noch 10 Jahre seine Gültigkeit behalten.

Für die Hardwareentwickler ergibt sich die Konsequenz, dass auf der gleichen Fläche Silizium eine immer größere Funktionalität untergebracht werden kann. Die Folge ist ein

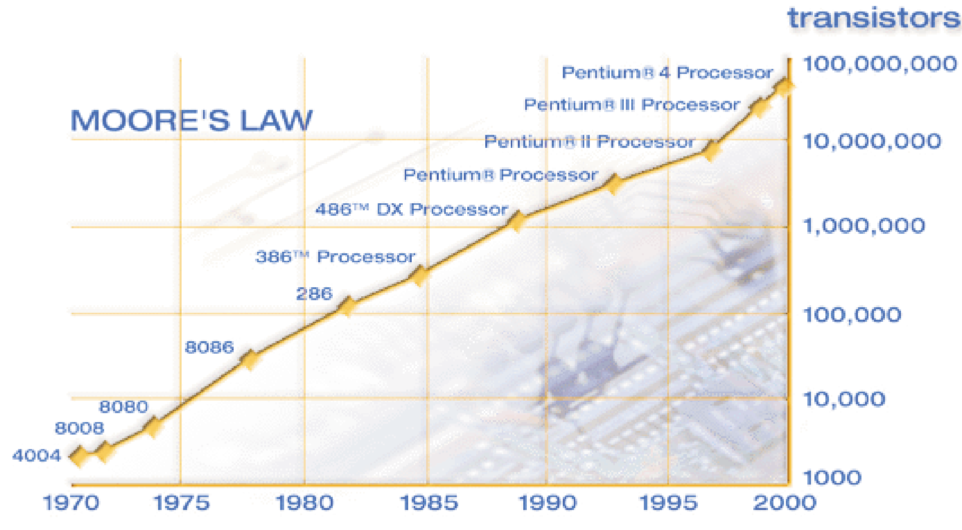


Abbildung 2.2.3.: Darstellung von Moores Gesetz, Quelle [Int]

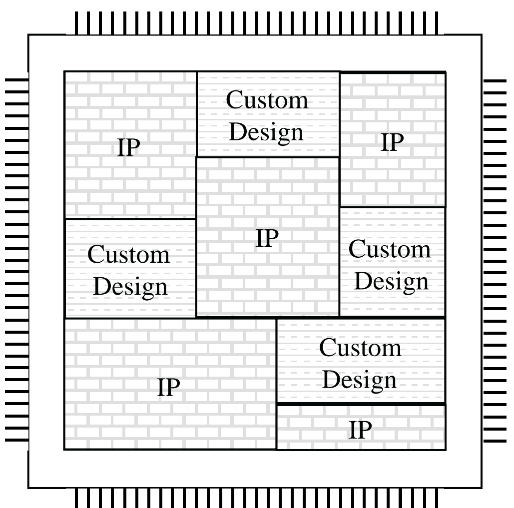


Abbildung 2.2.4.: Aufbau eines System-on-Chip aus IPs und selbstentwickelten Komponenten

System-on-Chip-Design, bei dem versucht wird, ein digitales System auf einem einzigen Chip zu implementieren.

Der Vorteil einer System-On-Chip-Lösung ist eine größere Zuverlässigkeit gegenüber Systemen aus mehreren Chips. Schwachstellen liegen bei diesen Systemen vor allem in der physischen Verdrahtung der Chips untereinander.

Der Nachteil eines System-on-Chip ist eine größere Spezialisierung der Funktion. Mit einer größeren Fülle an Funktionen sinkt das Anwendungsfeld der Chips. Doch je kleiner das Anwendungsfeld, desto weniger Stückzahlen werden vom Chip gebraucht. Jedoch müssen sich mit dem Verkauf des Chips auch die Entwurfskosten amortisieren. Damit sich der Entwurf von SoC-Lösungen auch weiterhin lohnt, müssen die Entwicklungskosten und die Time-to-Market[2] sinken.

Strategien zur Senkung der Entwicklungskosten und der Time-to-market stellen zum Einen eine weitere Entwurfsautomatisierung und zum Anderen die Wiederverwendung bereits entwickelter Komponenten dar. Die Wiederverwendung von Komponenten basiert auf der Nutzung von Intellectual Properties.

Intellectual Properties (dt. Geistiges Eigentum) ist nach der allgemeinen Definition jedes Produkt des menschlichen Intellekts, das einzigartig, neuartig und nicht offensichtlich ist. Dazu gehören literarische, künstlerische und wissenschaftliche Arbeiten, Leistungen von Künstlern, Erfindungen auf allen Gebieten der menschlichen Erkenntnis, wissenschaftliche Entdeckungen, industrielle Designs, eingetragene Waren- und Dienstleistungsmarken, Handelsnamen und Kennzeichnungen sowie alle weiteren Rechte aus intellektueller Tätigkeit in den Gebieten der Industrie, Wissenschaft, Literatur oder Kunst. [Wor, Mar02, Mic04]

[2]Time-to-market gibt die Zeit an, die von der Idee bis zur Marktreife eines Produktes vergeht.

Eine Intellectual Property (IP) im Sinne der Hardwareentwicklung stellt eine Hardwarekomponente dar, die bereits entwickelt wurde. Diese Komponenten werden in neue Designs neben selbst entwickelten Komponenten eingebunden (Abbildung 2.2.4), im Idealfall ohne daran Änderungen vornehmen zu müssen. Dadurch entfällt die Entwicklung dieser Teilsysteme im Entwurfsprozess und Entwicklungszeit wird eingespart.

Die IP kann in verschiedenen Beschreibungen vorliegen [Har04]. Man unterscheidet zwischen Hard-IPs und Soft-IPs. Hard-IPs besitzen ein vorgegebenes Layout und Timing. Sie haben fixe Schnittstellen. Die Optimierung der IPs ist in Bezug auf Geschwindigkeit, Siliziumfläche und Leistungsaufnahme getroffen. Sie sind leicht zu verifizieren, außerdem halbleiterhersteller- und technologieabhängig. Jedoch besitzen sie den Nachteil, dass sie unflexibel sind.

Soft-IPs basieren auf einer synthetisierbaren Beschreibung ausgehend von einer Hardwarebeschreibungssprache wie zum Beispiel VHDL oder Verilog. Eine Soft-IP ist vorcompiliert als Netzliste oder als Quellcode verfügbar. Sie sind halbleiterhersteller- und technologieunabhängig konzipiert. Soft-IPs sind flexibel und zum Teil parametrisierbar.

Das Konzept der IPs wird als wichtiger Schritt angesehen, um die Entwicklungszeiten von Hardware zu verringern. Vor allem für die System-on-Chip-Entwicklung stellt es ein Schlüsselkonzept dar.

Jedoch gibt es bei der IP-Nutzung noch einige Hindernisse. Zum Beispiel stellt sich die Frage nach einem einheitlichen Qualitätkriterium für IPs, um bei der Auswahl von IPs einheitliche Vergleichskriterien zu haben. Zu den Qualitätsmerkmalen einer IP zählen unter anderem der Umfang der Dokumentation, Skripte zur Logiksynthese, Applikationsbeispiele, Simulationsergebnisse, Testdaten und Verifikationsergebnisse.

Um bestehende Hindernisse bei der Nutzung und dem Handel von IPs zu bewältigen, wurde das Projekt IPQ [ipq] ins Leben gerufen. Die Zielsetzung des Projektes liegt darin, eine entscheidende Verbesserungen für die Qualitätssicherung bei der Anwendung und Entwicklung von IP-Modulen zu erzielen. Dazu gehören Spezifikationsmethoden, eine intelligente IP-Suche, Eingangschecks von IPs, Verfahren zur IP-Anpassung und Beiträge zur Standardisierung.

Das dabei auftretende Problem lag im Nichtvorhandensein eines einheitlichen Beschreibungsstandards über Firmengrenzen hinweg. Jede Firma benutzt ihre eigenen internen Standard und verzichtet auch ungern auf diese, da eine Umstellung mit hohen Kosten verbunden ist. Dadurch wird der IP-Handel erschwert, denn die IPs müssen nach dem Kauf manuell an den firmeninternen Standard angepasst werden.

Zur Lösung des Problems musste ein standardisiertes und von den Firmen akzeptiertes Austauschformat gefunden werden. Dieses Format musste außerdem die Möglichkeit bieten, eine firmeninterne Beschreibungen in das standardisierte Format zu konvertieren, aber auch den Weg vom standardisierten in firmeninterne Formate gewährleisten. Als Lösung wurde das IPQ-Format entwickelt. Es basiert auf XML-Schema. Außerdem fand

die Entwicklung zahlreicher Tools statt, die zum Beispiel das Suchen einer IP vereinfachen. Weitere Informationen zu diesem Thema bieten [ipq], [VLKH04], [VH04] und [VLH+03].

Mit Hilfe des IPQ-Formates und der dazu entwickelten Tools wird eine standardisierte Kommunikationsmöglichkeit von IP-Usern und IP-Serviceprovidern (vgl. Abbildung 2.2.5) geboten. Das System ist bereits einsatzfähig. Jedoch scheitert sein Einsatz durch teilweise ungeklärte rechtlichen Fragen zum IP-Handel und zur IP-Nutzung. Doch auch für dieses Problem wird sich eine Lösung finden und eine intensivere Nutzung von IPs beim Systementwurf wird Einzug halten. Damit lassen sich dann die Entwicklungszeit und -kosten weiter senken.

2.3. Simulation

Der Begriff Simulation wurde vom Verein Deutscher Ingenieure (VDI) in der Richtlinie 3633 [Ver96] wie folgt definiert: „Simulation ist ein Verfahren zur Nachbildung eines Systems mit seinen dynamischen Prozessen in einem experimentierbaren Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind.
Im weiteren Sinne wird unter Simulation das Vorbereiten, Durchführen und Auswerten gezielter Experimente mit einem Simulationsmodell verstanden.

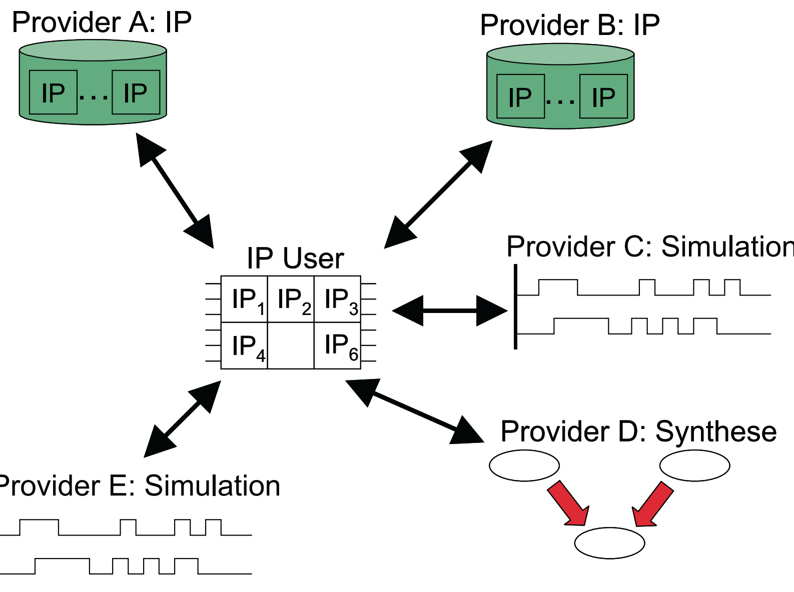


Abbildung 2.2.5.: IPQ soll eine standardisierte Kommunikation von IP-User und verschiedenen IP-Serviceprovidern erlauben. Quelle [VH04]

Entwurfsebene	Aufwand (relativ)
Verhalten	1
Register-Transfer	10
Gatter (Logik)	100
Schalterebene	1 000
Elektrische Ebene	10 000
Geometrie	100 000 - 1 000 000

Tabelle 2.3.1.: Simulationsaufwand von verschiedenen Entwurfsebenen, Quelle [Mül02]

Mit Hilfe der Simulation kann das zeitliche Ablaufverhalten komplexer Systeme untersucht werden.“

Die Simulation ist ein wichtiges Werkzeug beim Hardwareentwurf, um ein System während des Entwurfsprozesses zu testen und Fehler frühzeitig zu lokalisieren und zu beseitigen. Die Durchführung von Simulationen soll den Nachweis erbringen, dass der verfeinerte Entwurf noch mit der Spezifikation des Systems übereinstimmt.

Den Ausgangspunkt einer Hardwaresimulation bildet eine Beschreibung in einer Hardwarebeschreibungssprache (HDL, engl. Hardware Description Language). Die wichtigsten Sprachen sind VHDL, Verilog und SystemC. Auf Basis dieser Beschreibung wird von einem Simulationswerkzeug ein Simulationsmodell erstellt. Ein Simulationsmodell kann unterschiedliche Abstraktionsebenen eines Systems nachbilden. Dabei erhöht sich der Detailgrad des Simulationsmodells mit steigender Nähe zur Implementierung.

Mit steigendem Detailgrad nähert sich die Genauigkeit des Simulationsmodells stark dem realen System an. Durch die Berücksichtigung von Verzögerungszeiten von Gattern und Leitungen, sowie dem Übergang zu einer vierwertigen Logik, entsprechen die Ergebnisse der Simulation den real zu erwarteten Daten des zu implementierenden Systems.

Der hohe Detailgrad der Simulation wirkt sich jedoch stark auf die Simulationsgeschwindigkeit aus. Durch eine steigende Anzahl zu berücksichtigender Parameter in einem Simulationsschritt, steigt der Simulationsaufwand stark an und die Simulationsgeschwindigkeit verringert sich. Tabelle 2.3.1 stellt diesen Sachverhalt dar. Dauert eine Simulation auf der Verhaltensebene nur ein paar Minuten, kann die gleiche Simulation auf Gatterebene durchaus mehrere Stunden beanspruchen. Deshalb sollte bei der Vorbereitung einer Simulation darauf geachtet werden, welche Genauigkeit die Simulationsdaten zur Verifizierung der aktuellen Entwurfsebene besitzen müssen und dementsprechend das Simulationsmodell auszuwählen.

Es existieren eine Vielzahl von Simulatoren und Simulationsmethoden, die die Simulation unterschiedlicher Abstraktionsebenen beherrschen. Einige Simulatinsmethoden versuchen zudem, dem Zeitverlust, der durch genauere Modelle entsteht, mit verschiedenen Ansätzen entgegenzuwirken. Eine kleine Auswahl von Simulatoren und Simulationsmethoden sollen die folgenden Abschnitte vorstellen.

2.3.1. Simulation mit SystemC

SystemC ist eine C++ Klassenbibliothek in der auch ein Simulator vorhanden ist. Eine ausführliche Beschreibung von SystemC enthält Abschnitt 2.1.

Der Ablauf der Simulation ist in Abbildung 2.3.6 dargestellt. Den Ausgangspunkt stellt das in SystemC beschriebene Modell eines Systems dar. Die einzelnen Module des Modells und der Testbench werden von einem C++-Compiler übersetzt und mit dem Simulationskern aus der Bibliothek zusammengelinkt. Das Ergebnis ist ein ausführbares Programm, welches die Simulationsergebnisse erzeugt.

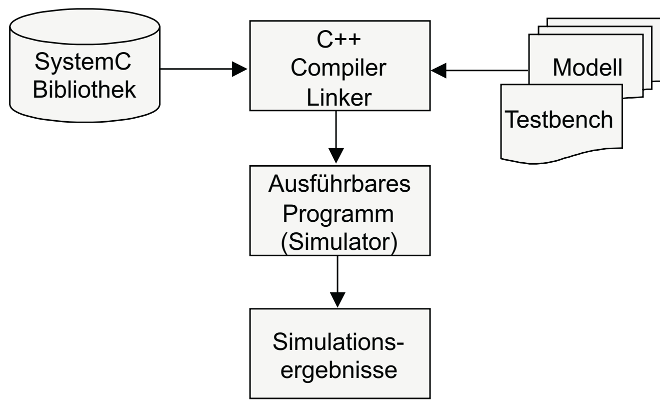


Abbildung 2.3.6.: Simulation eines Modells mit SystemC

Die SystemC-Simulation ist zyklusbasiert. Die Aktualisierung von Prozessen und Signalen erfolgt mit dem Übergang des Taktsignals. Die Folge ist, dass die Genauigkeit der Simulation auf eine Taktperiode beschränkt ist. Die SystemC-Bibliothek enthält zur Simulationssteuerung einen zyklusbasierten Scheduler. Er behandelt alle Ereignisse von Signalen und führt die Prozesse aus, deren Eingangssignale sich ändern. Die manuelle Simulationskontrolle ist beschränkt auf das Starten, Stoppen und schrittweises Ausführen der Simulation durch SystemC-Funktionen.

Die Ausgabe der Simulationsergebnisse kann in Tracefiles erfolgen. Dazu bietet die Bibliothek spezielle Funktionen, die es auch erlauben, die Tracefiles in verschiedenen Formaten zu generieren. Formate die unterstützt werden sind das Integrated Signal Data Base (ISDB) Format, Waveform Intermediate Format (WIF) und das Value Change Dump (VCD) Format. Eine andere Möglichkeit die Simulationsdaten auszugeben, besteht in der Nutzung von vorhandenen C++-Ausgabefunktionen wie *cout*.

Weitere Informationen zu SystemC und der Simulation damit finden sich im Benutzerhandbuch [Opeb] sowie in der funktionalen Spezifikation [Ope02].

2.3.2. Hardwaresimulation mit ModelSim

ModelSim® ist eine HDL-Simulationssoftware, die von der Firma Model Technology [Mod] entwickelt wurde. Model Technology ist eine betriebseigene Tochtergesellschaft von Mentor Graphics [Mena]. ModelSim unterstützt die Simulation von VHDL, Verilog und SystemC. Dabei können Simulationsmodelle von der Verhaltenssimulation über Netzlistenmodelle bis hin zur platzierten und trassierten Schaltung mit einem genauen Zeitmodell simuliert werden. Den Simulator gibt es für alle gängigen Betriebssysteme.

ModelSim®

Abbildung 2.3.7.: Logo von ModelSim®, Quelle [Mod]

Zum Simulator gehört ein Projektmanager zur Kontrolle der Simulation. Zusätzlich bietet er eine integrierte Debugumgebung, die es erlaubt, auf alle Signale des Designs zuzugreifen. Der Verlauf der einzelnen Signale wird mit einem Waveformbetrachter visualisiert und kann dadurch genau verfolgt werden. Ein Wizard und Templates erlauben dem Benutzer, Testbenches schnell zu erstellen. ModelSim benutzt als Simulationssprache eine Tcl basierte Skriptsprache. Dadurch können an den Simulator leicht andere Programme angebunden werden.

Die Simulation erfolgt auf die folgende Weise. Die Hardwarebeschreibung wird zunächst plattformunabhängig übersetzt. Anschließend erzeugt der Compiler daraus einen maschinenspezifischen Code für die ausführende Maschine, der zur Laufzeit optimiert wird. Die Ausführung des Codes erzeugt die Simulationsdaten des Modells. ModelSim zeichnet die Ausgaben des Codes auf und stellt sie zur Auswertung dem Nutzer zur Verfügung.

Von ModelSim existieren verschiedene Versionen. Die meisten Features stellt die LE Version zur Verfügung. Mehr Informationen zum Simulator bietet die Homepage von Model Technology [Mod] und das Handbuch zu ModelSim [Mod03].

2.3.3. HW/SW-Cosimulation

Hardware/Software-Codesign ist ein Schlüsselkonzept, um die Entwicklung moderner Hardware/Software-Systeme zu vereinfachen und zu beschleunigen. Anstatt Hardware und Software getrennt zu entwickeln und beide Bereiche im Anschluss mühevoll zusammenzufügen, erlaubt Hardware/Software-Codesign die Entwicklung beider Bereiche zu parallelisieren. Die Cosimulation von Hardware und Software liefert dabei ein wichtiges Hilfsmittel, um das Design zu testen und zu verifizieren.

Die HW/SW-Cosimulation verfolgt zwei konkurrierende Ziele. Das erste Ziel besteht darin, die Simulation mit der größtmöglichen Geschwindigkeit zu betreiben. Dem entgegen steht ein hoher Detailgrad der Simulationsergebnisse. Zum Erreichen einer hohen Genauigkeit der Simulationsergebnisse werden Modelle mit einem hohen Detailgrad verwendet. Aber je mehr Details simuliert werden, umso geringer ist die Simulationsgeschwindigkeit. Aus diesem Grund existieren verschiedene Ansätze zur HW/SW-Cosimulation, die versuchen, die Simulationsgeschwindigkeit zu erhöhen und dabei möglichst genaue Ergebnisse zu erzielen.

Der ursprüngliche Ansatz besteht darin einen **Hardwaresimulator** zu nutzen. Die Simulation findet unter Nutzung einer einzigen Simulationsumgebung statt, in der sowohl Hardware als auch die darauf ausgeführte Software simuliert wird. Dieser Ansatz bietet eine geringe Simulationsgeschwindigkeit, da das simulierte Hardwaremodell einen hohen Detailgrad besitzt. Der Vorteil besteht darin, dass keine Schnittstellen zu anderen Systemen notwendig sind.

Zur Beschleunigung der Simulation kann ein **Bus-Modell** verwendet werden (Abbildung 2.3.8). Ein Bus-Modell ist eine Hardwarebeschreibung, die nur ein ereignisdiskretes Modell des Bus-Interfaces des Prozessors simuliert. Die Funktionalität des Prozessors, dass heißt die Applikationssoftware, führt ein Befehlsinterpreter aus. Er verfügt über Information zu den benötigten Taktzyklen einer Sequenz von Befehlen zwischen zwei I/O-Operationen auf dem Bus. Dieses Modell ist nützlich, um die low-level Interaktion der Kommunikation auf dem Prozessorbus zu simulieren. Jedoch ist es unter Umständen nicht einfach, ein genaues Busmodell eines Prozessors zu erstellen.

Einen anderen Ansatz verfolgen **Instruction-set Simulatoren (ISS)**. Ihn stellt Abbildung 2.3.9 dar. Ein ISS bildet ein Modell der Befehlssatzarchitektur eines speziellen Prozessors nach. Das Modell enthält alle Details eines ereignisdiskreten Prozessormodells, welches die volle Funktionalität des Prozessors bereitstellt. Dieser Cosimulationsansatz verbindet einen Hardwaresimulator, auf dem die Hardwarekomponenten simuliert werden und den ISS, der die Ausführung der Software übernimmt.

Die **Heterogene Cosimulation** verfolgt die Kopplung von Hardware- und Software-Entwicklungstools. Dies erlaubt die schnelle Ausführung von Software-Anwendungscode

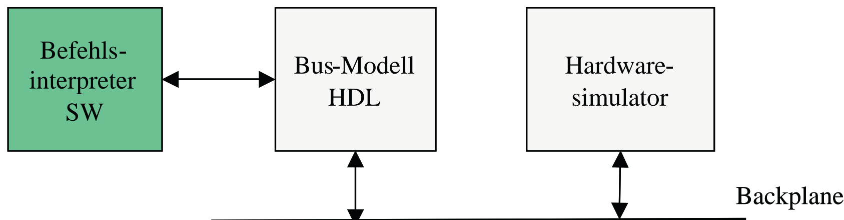


Abbildung 2.3.8.: HW/SW-Cosimulation mit dem Bus-Modell

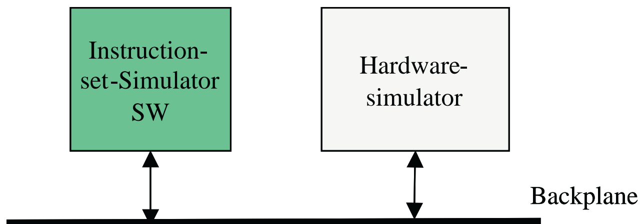


Abbildung 2.3.9.: HW/SW-Cosimulation mit einem Instruction-set-Simulator

auf einer simulierten Hardware. Die Cosimulation verschiedener Hardwarekomponenten und der Software läuft in einem heterogenen Netzwerk aus PC's oder Workstations.

Der Ansatz des **Hardware-Modells**, wie ihn Abbildung 2.3.10 zeigt, kann benutzt werden, falls der Zielprozessor bereits existiert. Zur Ausführung der Software findet der Zielprozessor statt eines Simulators Verwendung. Damit ist eine Echtzeitausführung der Software möglich. Die Simulation der Hardwarekomponenten findet durch einen Hardwaresimulator statt, der mit dem Prozessor gekoppelt ist.

Beim **compilierten Modell** kommt der Prozessor des ausführenden Rechners zum Einsatz (Abbildung 2.3.11). Die zu simulierende Software wird in Code dieses Prozessors transformiert und auf diesem ausgeführt. Eine Schnittstelle verbindet die Softwareausführung mit dem Hardwaresimulator, der die Ergebnisse der Hardwarekomponenten berechnet.

Die höchste Simulationsgeschwindigkeit erzielt unter allen Ansätzen die **Hardwareemulation**, welche Abbildung 2.3.12 veranschaulicht. Dieses Verfahren nutzt programmierbare Hardwarebausteine, wie zum Beispiel FPGAs. Auf Basis dieser Bausteine werden die entwickelten Hardwarekomponenten implementiert. Die Softwareausführung findet auf dieser Implementierung statt. Die Geschwindigkeit, die dieser Ansatz erreicht, kann bis zu einem Zehntel der realen Ausführungsgeschwindigkeit betragen.

Bei der Verwendung von mehreren Simulatoren spielt die Schnittstelle zwischen ihnen eine große Rolle. Die Schnittstelle stellt in den meisten Fällen den größten Engpass dar

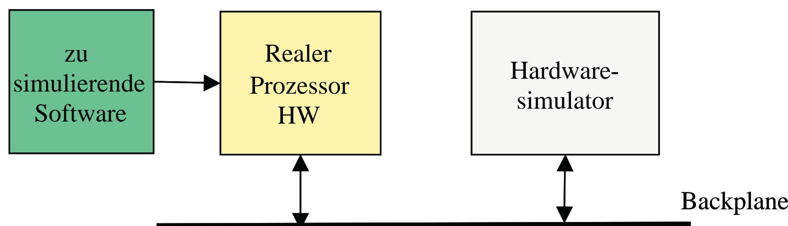


Abbildung 2.3.10.: HW/SW-Cosimulation mit dem Hardware-Modell

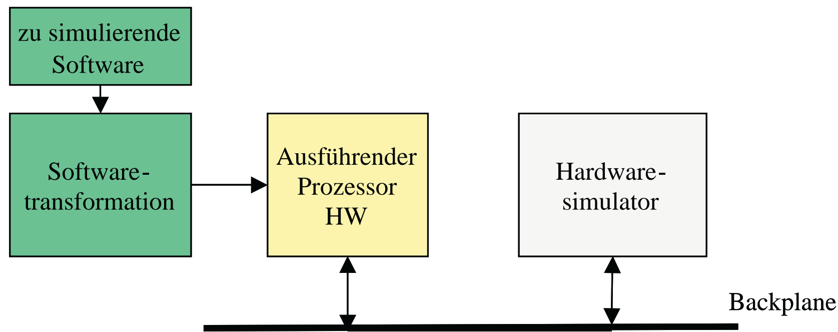


Abbildung 2.3.11.: HW/SW-Cosimulation mit dem compilierten Modell

und ist somit ein wichtiger Faktor für die resultierende Simulationsgeschwindigkeit. Die Kopplung der Simulatoren kann auf verschiedene Arten erfolgen.

Die Simulatoren können parallel arbeiten und dabei durch einen Synchronisationsmechanismus für den Datenaustausch gekoppelt sein. Das hat zur Folge, dass ein schneller Simulator eventuell lange auf einen langsameren warten muss. Außerdem entsteht durch die Synchronisation ein hoher Kommunikationsoverhead für die Schnittstellen.

Ein anderer Kopplungsansatz besteht in einer Master-Slave-Simulation. Dabei übernimmt ein Simulator die Rolle des Masters und ruft andere Simulatoren auf, wenn er Ergebnisse von ihnen benötigt. Bei der Ausführung mehrere Simulatoren auf einem Prozessor bringt diese Variante Vorteile. Erstens wird jeweils nur ein Simulator ausgeführt, dem die gesamte Prozessorleistung zur Verfügung steht. Zusätzlich verringert sich der Kommunikationsoverhead, da nur wenige Synchronisationsdaten ausgetauscht werden müssen.

Die Hardware/Software-Cosimulation stellt ein wichtiges Werkzeug dar, um die Entwicklung für zukünftige Hardware/Software-Systeme zu beschleunigen und zu vereinfachen. Eine ausführlichere Beschreibung der Cosimulationsansätze bieten [Vra98] und [Har04].

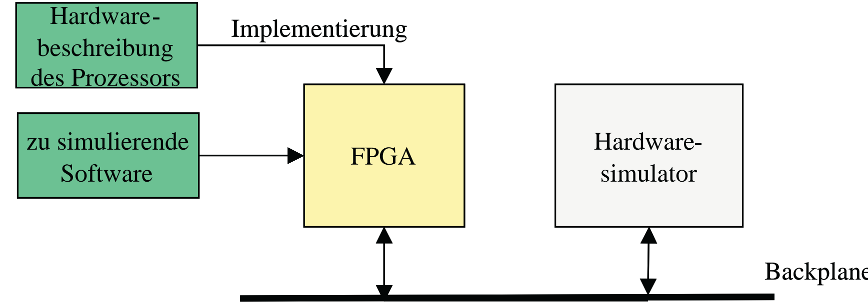


Abbildung 2.3.12.: HW/SW-Cosimulation mit Hardwareemulation

2.3.4. Hardware-Akzeleratoren

Der Nachteil einer HDL-Simulationssoftware ist die stark sinkende Simulationsgeschwindigkeit mit steigendem Detailgrad und steigender Designgröße. Um diesen Nachteil der Simulation zu vermeiden, werden Hardwareakzeleratoren eingesetzt. Sie emulieren das System auf FPGAs oder simulieren es auf Spezialprozessoren. Dadurch steigert sich die Geschwindigkeit der Simulation zum Teil erheblich. Nach Angaben einiger Hersteller solcher Systeme liegt der Beschleunigungsfaktor zwischen 10 und 10 000.

Ein Hardwareakzeleratorsystem besteht allgemein aus einer Spezialhardware und einem Softwarepaket. Die Spezialhardware hat die Aufgabe, die Simulation oder Emulation durchzuführen und die Simulationsergebnisse zu liefern. Das Softwarepaket ist dafür zuständig, den ausführbaren Code bzw. die Konfigurationsdaten für den Hardwareakzelerator aus einer HDL-Beschreibung zu compilieren bzw. zu generieren. Außerdem hat es die Aufgabe, die Kommunikation zwischen dem Hostrechner und dem HW-Akzelerator zu managen. Dies beinhaltet den compilierten Code in den HW-Akzelerator zu laden, Befehle zur Steuerung und zum Debuggen an den HW-Akzelerator zu senden und die Simulationsdaten zu empfangen und aufzuzeichnen.

Tharas - Hammer 100

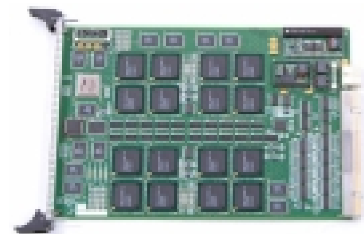
Den Hardwareakzelerator Hammer® 100 stellt die Firma Tharas Systems [Thaa] her. Es ist ein Simulationssystem, das aus Spezialprozessoren besteht. Die Prozessoren berechnen das Simulationsmodell, das aus einer Hardwarebeschreibungssprache kompiliert wurde.

Das System besteht aus bis zu 8 Boards mit je 16 Spezialprozessoren je nach Ausbaustufe. Die Prozessoren sind in 0,18 Mikrometertechnologie mit 5 Metallisierungsebenen gefertigt. Das System ist in einem CompactPCI-Gehäuse untergebracht, wie in Abbildung 2.3.13 zu sehen ist.

Die Prozessoren sind speziell dafür gebaut, um Hardwarebeschreibungskonstrukte zu beschleunigen. Alle Prozessoren können miteinander kommunizieren und in jedem Befehlszyklus gemeinsam Daten benutzen. Das Gehäuse nutzt eine geschützte Backplane, um eine hohe Prozessorkonnektivität zu liefern. Die Busarchitektur wurde speziell entworfen, um den hohen Kommunikationsanforderungen eines hoch parallelen Rechensystems gerecht zu werden.

Als Hardwarebeschreibungssprachen werden der IEEE Verilog 1364-2001 und der IEEE VHDL 1076-2002 Standard unterstützt. Der Compiler analysiert eine Hardwarebeschreibungssprache auf syntaktische und semantische Korrektheit. Aus der Beschreibung generiert er optimalen, parallelen Code für die Spezialprozessoren. Der Compiler übersetzt bis zu 50 Millionen RTL-Gatteräquivalente je Stunde.

Hammer® 100 System



Prozessorboard des Hammer® 100

Abbildung 2.3.13.: Hammer® 100 System von Tharas, Quelle [Thab]

Das Hammer 100 System unterstützt verschiedene Hardwarebeschreibungskonstrukte, wie zum Beispiel Latches, mehrere Takte, Gated Clocks und Modelle mit Signalstärken. Alle synthetisierbaren Verilog und VHDL Konstrukte können beschleunigt werden. Darüber hinaus ist es auch möglich, Testbenches zu beschleunigen, sowie Modelle mit Verzögerungszeiten auszuführen.

Zur Software gehört auch der Runtime Manager. Er erleichtert die Laufzeitkontrolle der Hardware und die Kommunikation zwischen dem Akzelerator und dem Host-Software-Simulator. Die Hammer 100 Software erlaubt eine nahtlose Integration von existierenden Verifikationsumgebungen. Somit kann eine verteilte Ausführung der Simulation auf dem Hammer 100 und einem Software-HDL-Simulator ebenfalls vollzogen werden.

Der Debugger kommuniziert mit dem Hardware Trace-Buffer, um während des Debuggen des Designs einen schnellen Datengewinn aus dem Akzelerator zu gewährleisten. Das Debugging wird in Hardware ausgeführt und ist dadurch sehr schnell. Auch schrittweises Debugging des Designs ist möglich. Die Aufzeichnung der Simulationsdaten kann in einem Tracefile erfolgen.

Nach Angaben von Tharas Systems [Thab] beschleunigt das Hammer 100 System die Simulation um den Faktor 10 bis 10000 gegenüber dem schnellsten Software-HDL-Simulator. Die Ausbaustufen des Hammer 100 sind für 2, 4, 8, 16 und 32 Millionen Gatteräquivalente ausglegt. Bei einer Zusammenschaltung mehrerer Hammer 100 simuliert das System bis zu 64, 96 oder 128 Millionen Gatteräquivalente. Der Preis des Simulationssystems liegt zwischen 2,65 und 7,5 US-Cent pro Gatterequivalentkapazität.

Mehr Informationen zu diesem System bietet die Homepage von Tharas Systems [Thaa] und die Broschüre zum Hammer 100 System [Thab].

Mentor Graphics - VStation

Das VStation-System von Mentor Graphics [Mena] gliedert sich in zwei Teile. Die Hardwareseite bildet die VStation™PRO und das Softwarepaket heißt VStation™TBX. Das System stellt eine komplette Umgebung bereit, die es ermöglicht, komplexe Designs von 1,6 bis 120 Millionen Gatter zu verifizieren.

Die VStationPRO ist eine FPGA-basierte Emulationsumgebung, die sich aus mehreren Boards zusammensetzt und dadurch leicht skalierbar ist. Ein Board besitzt die Ressources, um entweder 1,67, 3,33 oder 6,67 Millionen Gatter zu emulieren. Die Emulationsgeschwindigkeit des Systems reicht von 500 KHz bis 2 Mhz. Nach außen stellt das System je nach Ausbaustufe zwischen 512 und 4608 I/O-Verbindungen bereit.

Der VStationPRO VirtualLogic™ Compiler erlaubt ein direktes Mapping des Designs von der Register-Transfer-Ebene auf die FPGAs der VStationPRO ohne zuvor eine Synthese auszuführen. Dadurch werden Signalnamen und die Hierarchie des Designs beibehalten. Eine Simulatorähnliche Debugumgebung ermöglicht es, alle Signale des Designs zu überwachen. Auch Quellcodedebugging mit Haltepunkten und das Anzeigen von Waveforms wird unterstützt.

Die Software VStationTBX (TestBench-XPress) liefert zusätzlich noch eine Verifikationsumgebung für die Simulation und Emulation. Es werden die Sprachen SystemC und SystemVerilog unterstützt. VHDL kann nur als RT-Beschreibung verarbeitet werden. VStationTBX erlaubt eine schnelle Kommunikation zwischen der VStationPRO, einem Simulator und den Verifikationsprogrammen auf dem Hostrechner.

Weiterführende Informationen zum VStation-System finden sich auf der Homepage der VStation [Menb]. Produktbeschreibungen zur VStationPRO liefern [Menc] und [Men03]. Mehr Informationen über den VStationTBX enthalten [Mend] und [Men04].

Abbildung 2.3.14.: VStation™Pro System von Mentor Graphics, Quelle [Men03]

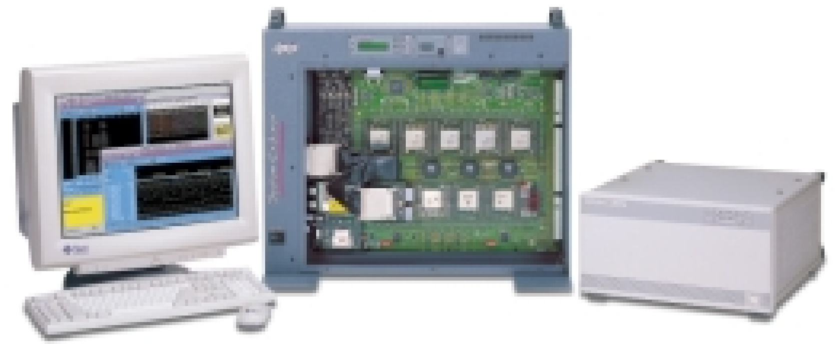


Abbildung 2.3.15.: Aptix® System Explorer™, Quelle [Apt00]

Aptix - System Explorer™

Der Aptix® System Explorer™ (Abbildung 2.3.15) ist ein System für Emulation, Rapid Prototyping und HW/SW-Cosimulation. Es erlaubt die Emulation und das Debugging eines System-on-Chip Designs.

Die Grundlage des Systems besteht aus der Field Programmable Circuit Board® Architektur. Diese Architektur kombiniert eine Prototypingfläche mit einer programmierbaren Verbindungsstruktur, den FPIC®s (Field Programmable Interconnect Component®). FPICs sind elektronisch konfigurierbare bi-direktionale Verbindungsstrukturen und stellen die Verbindungen zwischen den Prototypingflächen her. Die Prototypingflächen können mit FPGAs und Systemkomponenten wie DSPs, RAM oder Prozessoren bestückt werden. Dadurch bildet der System Explorer eine rekonfigurierbare Prototypingplattform.

Zum System Explorer gehört auch ein Softwarepaket. Es besteht aus den Programmen Design Pilot™, Explorer 2000™ und Expeditor™. Der Design Pilot führt das logische Mapping durch und generiert die Netzlisten. Er bildet die entworfene Logik und Soft-IPs auf die Ziel-FPGAs ab. Außerdem führt er eine automatische hierarchische Blockgruppierung und Partitionierung des Entwurfs durch.
Der Explorer 2000™ führt das Mapping des partitionierten Designs auf die physischen Komponenten des Prototypingsystems aus. Er routet die einzelnen FPGAs und Komponenten untereinander durch die FPICs und konfiguriert anschließend den Prototypen. Nach der Konfiguration führt die Software eine Automatisierung des Hardware-Debugging durch. Der Explorer 2000 routed automatisch Probes[3] und konfiguriert den Agilent Logikanalysator zum Erfassen von Simulationsdaten.
Die Aufgabe des Expeditor™ besteht darin, eine Schnittstelle zu anderen Simulatoren und Testwerkzeugen herzustellen. Damit kann eine verteilte Simulation erfolgen.

Dem System Explorer liegt eine blockbasierte Verifikationsmethodik zu Grunde. Diese erlaubt die schnelle Erstellung von Prototypen, ein einfaches Debugging und eine rasche

[3]Probes sind Leitungen innerhalb der FPGA, die nach außen geführt werden, um die Signalwerte auf den Leitungen abgreifen zu können.

Typ	**MP3CF**	**MP4CF**
Emulationskapazität in ASIC Gattern	2,5 Millionen	3 Millionen
Blockmemory	6 MBit	10 MBit
Prototypingfläche	1920 Pins	2880 Pins
max. Anzahl FPGAs	12	20
Anzahl Abgreifpunkte	1500	2000
Einsatzzweck	DSP-basierte Designs mit mäßigen Anforderungen an die Verbindungsstruktur zwischen den Komponenten	optimiert für Prototypen mit großen internen Bussen und hohen Anforderungen an die Verbindungsstruktur

Tabelle 2.3.2.: Technische Daten des Aptix® System Explorer™

Implementierung von Designveränderungen. Der Prototyp wird Block für Block nach der Hierarchie des Designs aufgebaut. Da Logikänderungen meist auf eine FPGA begrenzt sind, ermöglicht das System nach einer Korrektur am Design, die einzelne Ersetzung der fehlerbehafteten FPGA und somit eine schnelle Änderung des Designs.

Im Entwurfsprozess kann mit Hilfe des System Explorers eine parallele Entwicklung von Software und Hardware erfolgen. Der Prototyp kann benutzt werden, um das System-on-Chip in seiner realen Systemumgebung zu emulieren. Es kann begonnen werden, Schnittstellen zu erproben und andere digitale oder analoge Subsysteme, während die SoC-Entwicklung noch voranschreitet.

Das frühe Vorhandensein eines Prototypen erlaubt es, dem Kunden schon früh im Entwurfsprozess eine Demonstration zu bieten und seine Resonanz in die weitere Entwicklung mit einfließen zu lassen.

Der System Explorer wird in zwei Ausführungen hergestellt. Die Bezeichnungen sind MP3CF und MP4CF. Tabelle 2.3.2 stellt die technischen Daten der beiden Systeme gegenüber. Weiterführende Informationen zum Aptix System Explorer sind auf der Homepage des Unternehmens [Apt] sowie in der Broschüre zum System Explorer [Apt00] enthalten.

2.4. Fazit

Die vorangehenden Punkte beschäftigten sich mit SystemC als neue Entwurfssprache, mit dem Nutzen von Intellectual Properties und der Bedeutung der Simulation im Hard-

wareentwurf. Dazu wurden einige Simulationswerkzeuge und -methoden vorgestellt.

SystemC bringt die Voraussetzungen mit, um den Entwurfsprozess von Hardwaresystemen und Hardware/Software-Systemen weiter zu beschleunigen. Ein Nachteil ist noch die mangelnde Unterstützung von sprachfremden IPs über der Register-Transfer-Ebene.

Intellectual Properties bilden zukünftig ein Schlüsselkonzept bei der Entwicklung von SoC-Designs. Durch ihren Einsatz können die Entwicklungszeit und -kosten weiter gesenkt werden. Es existieren bereits Lösungen zum einfachen Handel mit IPs. Jedoch behindern teilweise rechtliche Unklarheiten im Moment noch den IP-Handel. Aber in absehbarer Zukunft wird sich dieses Konzept durchsetzen.

Die Simulation ist ein wichtiges Werkzeug im Entwicklungsprozess. Es existieren verscheidene Simulationswerkzeuge, die verschiedene Ansätze verfolgen. Ein Ansatz besteht in einer reinen Simulationssoftware wie Modelsim und SystemC. Ansätze zur Cosimulation verfolgen das Ziel, verschiedene Simulatoren miteinander zu koppeln, um die Simulation von Hardware und Software parallel ausführen zu können. Hardware-Akzeleratoren sollen durch Spezialhardware die Simulation großer Designs beschleunigen. Doch sie bieten auch die Möglichkeit der Kopplung mit Softwaresimulationslösungen. Die vorgestellten Hardware-Akzeleratoren erlauben teilweise die Nutzung von SystemC. Die Unterstützung beginnt jedoch erst auf Register-Transfer-Ebene. In höheren Abstraktionsebenen fehlt diese Unterstützung noch.

3. Die Methode der Adaptierung

Diese Kapitel beschreibt die Adaptierung eines Hardware-Akzelerators an die SystemC-Verhaltenssimulation. Der erste Teil beinhaltet die Herleitung des Hardware/Software-Interfaces[1] und die Analyse seiner Anforderungen. Eine Realisierung des HW/SW-Interfaces durch einen Interfaceblock wird im zweiten Abschnitt vorgestellt. Anschließend werden Details zur technischen Umsetzung dargestellt. Die Einsatzmöglichkeiten des HW/SW-Interfaces zeigt der letzte Abschnitt dieses Kapitels auf.

3.1. Das Hardware/Software-Interface

Das Ziel dieser Arbeit ist es, ein Verfahren zu entwicklen, welches die Simulation von VHDL-IPs in SystemC, auf einer höheren Abstraktionsebene als der RT-Ebene, erlaubt. Bisher ist die Einbindung dieser IPs für den Entwurf und somit auch für die Simulation erst auf Netzlistenebene möglich. In früheren Entwurfsstadien ist eine Simulation des Gesamtsystems mit diesen IPs jedoch schon sehr hilfreich, um Tests mit dem Systemverhalten durchführen zu können und frühzeitig Fehler zu entdecken.

Zur Simulation einer VHDL-IP mit SystemC existieren zwei grundsätzliche Möglichkeiten. Die erste Möglichkeit ist die Konvertierung der IP in ein SystemC-Modul (Abbildung 3.1.1). Die Konvertierung muss dabei von Hand erfolgen. Dieser Schritt verursacht zusätzliche Arbeit. Das erstellte SystemC-Modul muss gegen die VHDL-IP verifiziert werden, um sicherzustellen, dass ihr Verhalten übereinstimmt. Dieser Prozess bindet Ressourcen und kostet meist viel Zeit und Geld.

Einen Ansatz aus der Cosimulation vorfolgt die zweite Variante. Dabei findet die Simulation der VHDL-IP in einem VHDL-Simulator und die des SystemC-Modells in einem SystemC-Simulator statt. Die Kopplung der beiden Simulatoren realisiert eine Software/Software-Schnittstelle (Abbildung 3.1.2). Anforderungen, die die SW/SW-Schnittstelle erfüllen muss, sind der Austausch von Simulationsdaten und Kontrollsignalen sowie die Synchronisation zwischen den Simulatoren, damit zum richtigen Zeitpunkt die korrekten Simulationsdaten vorhanden sind.

[1]Für Interface wird in den folgenden Abschnitten der deutsche Begriff Schnittstelle verwendet

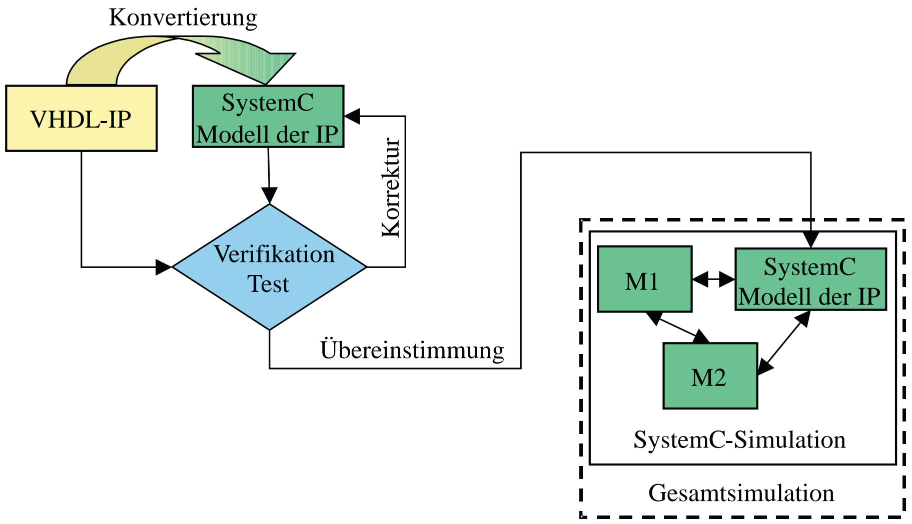


Abbildung 3.1.1.: Adaptierung durch Konvertierung

Allerdings kostet die Ausführung von zwei Simulatoren auf einem Rechner viel Rechenleistung. Die Folge ist eine geringere Simulationsgeschwindigkeit. Da kontinuierlich Daten zwischen den Simulatoren ausgetauscht werden müssen, spielt nicht nur die Ausführungszeit der beiden Simulatoren eine große Rolle sondern auch die Zeit, die die Kommunikation zwischen diesen beansprucht.
Der Vorteil dieser Variante liegt darin, dass die IP nicht von Hand konvertiert werden muss. Dieses Kriterium ist wichtig, weil dadurch die Zeit für Tests und Verifizierung eingespart wird. Um die Simulation zu beschleunigen, wird, zusätzlich zum Cosimulationsansatz, der Ansatz von Hardware-Akzeleratoren eingebracht. Das heißt, dass die IP in Hardware emuliert wird, was eine schnellere Ausführung der IP erlaubt, und mit dem SystemC-Simulator gekoppelt wird.

Der Einsatz von kommerziellen Hardware-Akzeleratorsystemen ist durch ihre hohen Kosten hier nicht praktikabel. Eine Lösung für dieses Problem wäre eine günstigere han-

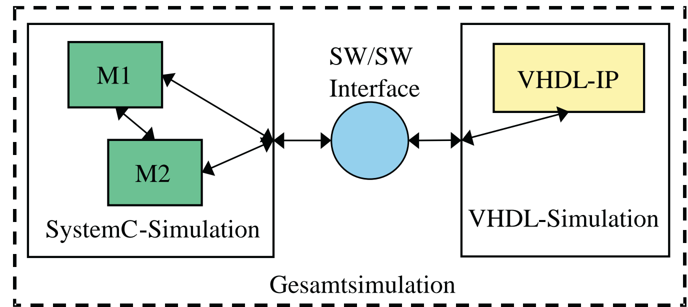


Abbildung 3.1.2.: Adaptierung eines Simulators

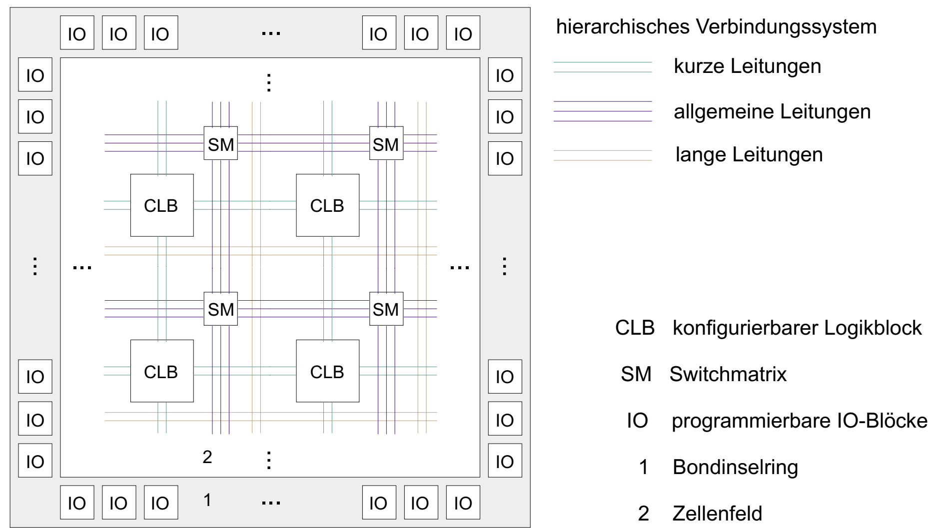


Abbildung 3.1.3.: Schematischer Aufbau eines SRAM-basierten FPGA

delsübliche Hardware, die schnell konfiguriert werden kann. Mit dieser Hardware sollte der Anwender auch in der Lage sein, größere Schaltungen schnell zu realisieren.

Diese Voraussetzungen erfüllen SRAM-basierte Field Programable Gate Arrays (FPGA). FPGAs sind programmierbare Hardwarebausteine und gehören zur Gruppe der anwenderprogrammierbaren Schaltungen. Den grundsätzlichen Aufbau von FPGAs stellt Abbildung 3.1.3 dar. Sie bestehen im wesentlichen aus konfigurierbaren Logikblöcken, konfigurierbaren IO-Blöcken und einer konfigurierbaren hierarchischen Verbindungstruktur. Mit Hilfe von FPGAs können kombinatorische und sequentielle digitale Schaltungen realisiert werden. Mehr Informationen zu FPGAs bieten [Wan98] und die Homepages von FPGA-Herstellern [Alt, Atm, Lat, Xilc].

Um jedoch die FPGA nutzen zu können, ist die Synthesefähigkeit der IP unabdingbar. Durch diese Einschränkung entfällt die Verwendung von Hard-IPs (vgl. Abschnitt 2.2). Sie sind bereits synthetisiert und mit Layout und Timing versehen. Soft-IPs dagegen sind synthetisierbar.

Durch die Nutzung der FPGA muss im Folgenden auf der Hardwareseite von Emulation statt von Simulation gesprochen werden. Emulation ist das funktionelle Nachbilden eines Systems durch ein anderes. Die Implementierung der IP ist nun kein Modell mehr, wie für eine Simulation, sondern ist eine Nachbildung der realen digitalen Schaltung mit Hilfe eines FPGAs.

Eine **Hardware/Software-Schnittstelle** tritt nun an die Stelle der Software/Software-Schnittstelle (Abbildung 3.1.4), um die SystemC-Simulation mit der IP-Emulation zu koppeln. Doch wie soll die Kopplung erfolgen und welche Anforderungen werden damit an die Hardware/Software-Schnittstelle gestellt?

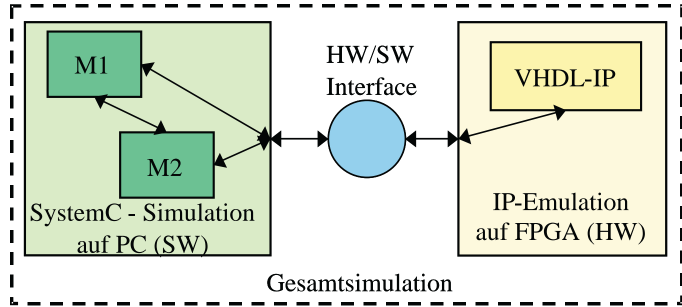


Abbildung 3.1.4.: Allgemeines Hardware/Software-Interface

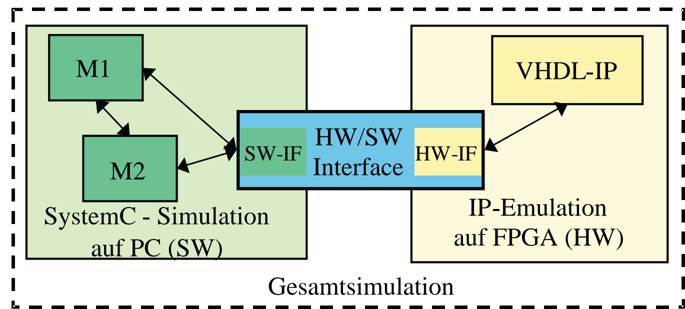


Abbildung 3.1.5.: Genaueres Modell des Hardware/Software-Interface

Die Kopplung der SystemC-Simulation erfolgt über eine SW/SW-Schnittstelle. Auf der Hardwareseite befindet sich eine HW/HW-Schnittstelle, um die Verbindung mit der IP herzustellen. Abbildung 3.1.5 stellt diese genauere Betrachtung der HW/SW-Schnittstelle dar. Dies ermöglicht es, die eigentliche Grenze zwischen Hardware und Software für die SystemC-Simulation sowie für die IP-Emulation unsichtbar zu machen. Welchen Anforderungen die HW/SW-Schnittstelle genügen muss, um diese Art der Verbindung herzustellen, wird im folgenden Abschnitt analysiert.

3.1.1. Anforderungsanalyse für das HW/SW-Interface

Die Anforderungen an die HW/SW-Schnittstelle sind zunächst die gleichen wie bei einer SW/SW-Schnittstelle:

- Austausch von Simulationsdaten und Kontrollsignalen ermöglichen
- Synchronisation zwischen Simulation und Emulation gestatten

- Nutzung der IP ohne diese zu verändern

Zunächst soll die Anforderung des Datenaustausch näher betrachtet werden. Er geschiet über zwei Schnittstellen. Zum Einen über die Schnittstelle von IP und HW/SW-Schnittstelle. Diese HW/HW-Schnittstelle hängt allein von der IP ab. Da keine Änderungen an der IP vorgenommen werden sollen, bilden die Ein- und Ausgangssignale der IP die einzige Kommunikationsmöglichkeit. Das hat zur Folge, dass die HW/HW-Schnittstelle genau durch die Eingänge und Ausgänge der IP realisiert wird.

Die andere Schnittstelle befindet sich auf der Softwareseite. Dort muss die SystemC-Simulation mit der HW/SW-Schnittstelle verbunden werden. Prinzipiell ist es bei dieser SW/SW-Schnittstelle möglich, sie um Kontrollsignale zu erweitern. Das Minimum bilden aber auch hier die Ein- und Ausgänge der IP, da sonst Simulationsdaten verloren gehen würden. Der Nachteil bei einer Einführung zusätzlicher Signale liegt darin, dass der Simulationskern verändert werden müsste, um die Signale zu erzeugen. Eine bessere Variante, die den Simulationskern unbeeinflusst lässt, besteht darin, das HW/SW-Interface für den Simulationskern unsichtbar zu halten und nur die Ein- und Ausgänge der IP für die Simulation bereitzustellen. Daraus ergibt sich eine weitere Anforderung an die HW/SW-Schnittstelle. Sie muss intern Kontrollsignale erzeugen und auswerten können, um den Transport der Simulationsdaten sicherzustellen.

Ein weiteres Problem, welches sich aus dem Datentransport durch die HW/SW-Schnittstelle ergibt, ist die Überwindung der Grenze zwischen Hardware und Software. Dies kann durch Standardschnittstellen am PC erfolgen. Dabei ist es notwendig, dass diese Schnittstelle die Kommunikation vom PC zur FPGA und umgekehrt beherrscht, da die Simulationsdaten in beide Richtungen ausgetauscht werden müssen. Standardschnittstellen besitzen feste Parameter hinsichtlich der Daten, die sie in einem Kommunikationszyklus übermitteln können. Da unterschiedliche IPs unterschiedliche Schnittstellenbreiten besitzen, kann keine Aussage getroffen werden, ob die gesamten Simulationsdaten in einem Kommunikationszyklus übertragbar sind. Deshalb sollte die HW/SW-Schnittstelle in der Lage sein, die Simulationsdaten intern zu serialisieren, sie vollständig und ohne Verlust über die HW/SW-Grenze zu transportieren und auf der anderen Seite wieder richtig zu parallelisieren. Das heißt, innerhalb der HW/SW-Schnittstelle muss eine Datentransformation gestattet sein.

Nachdem die Anforderungen an den Datenaustausch herausgearbeitet sind, stellt sich nun das Problem der Synchronisation zwischen der SystemC-Simulation und der IP-Emulation. Die SystemC-Simulation ist zyklusbasiert (vgl. Abschnitt 2.3.1). Hier dient das Taktsignal als Auslösesignal für die Neuberechnung der Signale des im Test befindlichen Designs. Da keine Änderungen am Simulationskern vorgenommen werden sollen, findet dieses Auslösesignal auch bei der Emulation Verwendung. Das heißt, dass die IP über das Taktsignal der SystemC-Simulation gesteuert wird. Eine Master-Slave-Simulation ist die Folge, bei der die SystemC-Simulation den Master darstellt, da sie das steuernde Signal generiert. Die Emulation dagegen nimmt, durch die Reaktion auf das Mastersignal, die Rolle des Slave an.

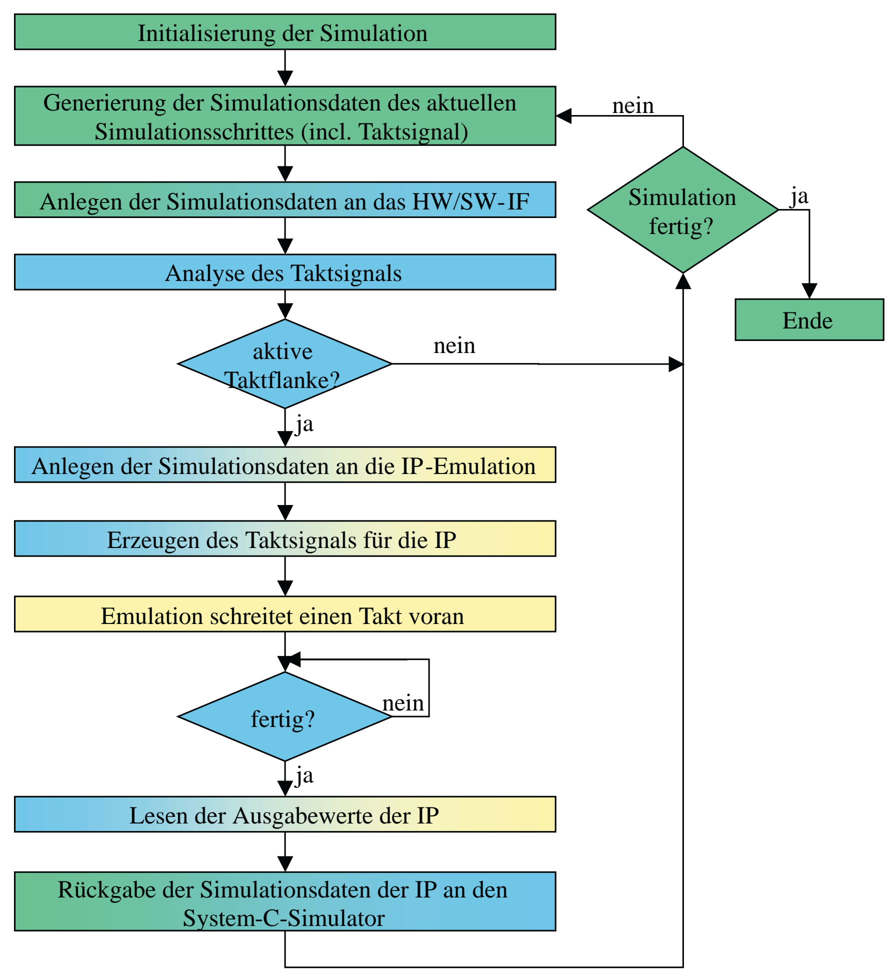


Abbildung 3.1.6.: Ablauf der Synchronisation durch das HW/SW-Interface

Jedoch wirft die Synchronisation über das Taktsignal wiederum ein Problem auf. Es besteht darin, dass das Taktsignal durch die HW/SW-Schnittstelle und über die darin enthaltene HW/SW-Grenze transportiert werden muss. Es muss garantiert werden, dass alle Eingangssignale korrekt anliegen, bevor das Taktsignal angelegt wird. Durch eine reine Übertragung des Taktsignals ist dies nur schwer möglich.

Eine Lösung besteht darin, das Taktsignal auf der Softwareseite zu analysieren. Bei der aktiven Taktflanke wird dann zuerst die Übertragung der Simulationsdaten ausgeführt, dann ein Kontrollsignal zur Takterzeugung auf der Hardwareseite generiert. Die HW/SW-Schnittstelle generiert für die Emulation nun genau einen Takt und gibt eine Rückmeldung an den Softwareteil nach dessen Beendigung. Am Ende werden die Ausgangsdaten der Emulation zur Simulation übertragen, welche weiter fortschreiten kann. Abbildung 3.1.6 stellt diese Art der Synchronisation im Rahmen der Simulation dar. Zur Umsetzung dieser Lösung benötigt die HW/SW-Schnittstelle einen Mechanismus, der es erlaubt, Eingangssignale zu analysieren und interne Kontrollsignale sowie Aus-

gangssignale zu generieren. Es muss also möglich sein, in der HW/SW-Schnittstelle eine Steuerung zu integrieren.

Die letze Anforderung stellt die Nutzung der IP dar, ohne sie zu verändern. Diese Anforderung wurde durch die vorangegangen Maßnahmen erfüllt. Eine Änderung der Schnittstelle oder der Funktionalität der IP fand nicht statt.

Zusammenfassend sollen hier noch einmal die Anforderungen an die HW/SW-Schnittstelle dargestellt werden. Die folgenden Anforderungen ergab die durchgeführte Analyse. Die HW/SW-Schnittstelle muss

1. zwei Komponenten kommunizieren lassen, ohne dass an den Komponenten Änderungen durchgeführt werden müssen.
2. den bidirektionalen Datentransport zwischen den Komponenten sicherstellen.
3. die HW/SW-Grenze intern überwinden.
4. die Möglichkeit bieten, Daten intern zu transformieren.
5. die Integration einer Steuerung erlauben.

3.2. Der Interfaceblock als HW/SW-Interface

Zur Realisierung der HW/SW-Schnittstelle benötigt man ein Schnittstellenkonstrukt, dass die im vorangegangen Abschnitt herausgestellten Anforderungen im Wesentlichen erfüllt. Falls das Konstrukt nicht allen Anforderungen genügt, sollte es so flexibel und erweiterbar sein, dass es sich auf die neuen Bedingungen leicht anpassen lässt. Ein Schnittstellenkonstrukt, dass einen Großteil der gestellten Anforderungen erfüllen kann, ist der Interfaceblock, der im folgenden Abschnitt kurz vorgestellt wird.

3.2.1. Das Modell des Interfaceblocks

Ein Interfaceblock (IFB) dient als Schnittstelle zwischen inkompatiblen Kommunikationskomponenten. Diese Kommunikationskomponenten können komplexe Strukturen (z.B. Bussysteme) oder funktionale Komponenten (z.B. Algorithmen) sein. Diese werden in der IFB-Terminologie entsprechend als Medium beziehungsweise Task bezeichnet.

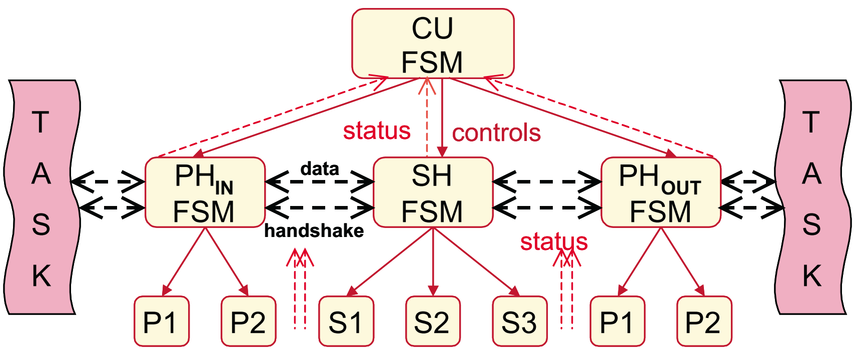


Abbildung 3.2.7.: Makrostruktur des Interfaceblock

In einem Interfaceblock können Daten zwischen Sender und Empfänger transformiert werden. Dadurch ist ein Einsatz als Adapter zwischen physisch inkompatiblen bzw. semantisch inkompatiblen Kommunikationskomponenten möglich.

Der Interfaceblock ist modular aufgebaut. Er besteht, wie Bild 3.2.7 zeigt, aus einer Controlunit (CU), zwei Protokollhandlern (PH) und einem Sequenzhandler (SH). Jeder der Handler besitzt zusätzlich mindestens einen Modus, der die eigentliche Funktionalität implementiert.

Die Controlunit übernimmt die Steuerungsaufgaben innerhalb des Interfaceblocks. Sie koordiniert und kontrolliert die Funktion der Protokollhandler und des Sequenzhandlers. Außerdem bestimmt sie den jeweils aktiven Modus eines Handlers. Sie steuert die Handler über Kontrollsignale und erhält deren aktuellen Zustand durch Statussignale.

Die Protokollhandler sind die externen Schnittstellen des Interfaceblocks. Die eigentliche Kommunikation mit den verbundenen Tasks und Medien erfolgt hier. Basierend auf seinen Modi verarbeitet der Protokollhandler die Protokolle der angeschlossenen Kommunikationskomponenten. Die Kommunikation beinhaltet sowohl das Senden als auch das Empfangen von Daten.

Das Bindeglied zwischen den Protokollhandlern stellt der Sequenzhandler dar. In ihm werden die eingehenden Nutzdaten konform zum ausgehenden Protokoll transformiert. Die Transformation folgt einer Abbildungsvorschrift, die die Sequenzhandlermodi implementieren. Die Transformation ist nicht nur auf strukturelle Änderungen der Daten beschränkt, sondern ermöglicht auch eine semantische Änderung.

Weitere Informationen zur Grundidee des Interfaceblocks, zur Beschreibungsform eines Interfaceblocks durch das Interface Synthese (IFS) Format, zur technischen Umsetzung und zum Stand der aktuellen Entwicklungen sind in [Ihm01], [HVI01], [IVH02a], [IVH02b], [IBJK+03], [IVH03], [Fic03], [Fla03] und [IH04] enthalten.

3.2.2. Analyse der Leistung des Interfaceblocks

Um den Interfaceblock als HW/SW-Schnittstelle einsetzen zu können, ist zunächst der Nachweis erforderlich, dass der Interfaceblock auch die gestellten Anforderungen (vgl. Abschnitt 3.1.1) theoretisch erfüllen kann.

Die erste Anforderung bestand darin, zwei Komponenten kommunizieren zu lassen, ohne an den Komponenten Änderungen durchzuführen. Der Interfaceblock erfüllt die Funktion eines Adapters. Ein Adapter ist nach [Wika] ein Gerät, dass zur Verbindung verschiedener mechanischer oder elektrischer Geräte dient, deren Anschlüsse wegen unterschiedlicher Formate oder Normen nicht zueinander passen. In dem vorliegenden Fall sind die zu verbindenden „Geräte“ die SystemC-Simulation und die IP-Emulation. Die Anschlüsse stellen die Schnittstellen dar, die in Software bzw. in Hardware vorliegen. Dadurch ist eine direkte Verbindung nicht möglich. Zur Verbindung wird die Eigenschaft des IFBs als Adapter genutzt. Somit ist die Verbindung jedenfalls theoretisch möglich.

Die nächste Anforderung umfasst den bidirektionalen Datentransport zwischen den Komponenten. Der IFB dient nach dem Modell als Schnittstelle zwischen Kommunikationskomponenten. Eine Schnittstelle (Interface) ist ein Teil eines Systems, das dem Austausch von Informationen, Energie oder Materie mit anderen Systemen dient [Wikb]. Nach der Definition beherrscht eine Schnittstelle den Austausch von Informationen, die im vorliegenden Fall die Simulationsdaten sind. Da der IFB eine Schnittstelle ist und Daten senden und empfangen kann, erfüllt er auch diese gestellte Anforderung.

Die Integration einer Steuerung in den IFB erfolgt durch die Controlunit. Sie erfüllt innerhalb des IFBs Steuerungs- und Kontrollaufgaben. Sie kontrolliert die Handler, indem sie Steuerungssignale an diese übermittelt und Statussignale der Handler auswertet. Somit ist sie der zentrale Punkt im IFB, von dem aus eine Steuerung arbeiten kann.

Weiterhin soll die HW/SW-Schnittstelle eine interne Datentransformation zulassen. Das kann in den Modi der Handler geschehen. Sie implementieren die Funktionalität zur Datenübertragung. Die Modi können also auch mit Algorithmen versehen sein, die Daten transformieren.

Die transformierten Daten müssen nun noch durch den IFB transportiert werden können. Innerhalb des IFBs findet Kommunikation zwischen der Controlunit und Protokollhandlern bzw. Sequenzhandler statt. Außerdem kommunizieren die Protokollhandler mit dem Sequenzhandler, um den Datentransport durch den IFB zu realisieren. Kommunikation, also der Austausch von Daten, läuft nach einem Protokoll ab, damit der Sender und Empfänger sich gegenseitig verstehen. Da das Modell des IFBs das interne Kommunikationsprotokoll nicht festlegt, ist die Verwendung eines Protokolls möglich, das den Transport der transformierten Daten ermöglicht.

Zur Überwindung der HW/SW-Grenze innerhalb des IFB soll die freie Auswahl eines Protokolls dienen. Ein Protokoll wird über eine Schnittstelle übertragen. Die Benutzung

einer Schnittstelle, die Hardware und Software physisch verbindet, bringt die Lösung dieses Problems. Der IFB ist aber bisher nur für reine Hardwareanwendungen konzipiert. Aus diesem Grund muss geprüft werden, ob sich das Modell dahingehend erweitern lässt, dass es die Einbindung einer physischen HW/SW-Schnittstelle gestattet.

3.2.3. Erweiterung des Interfaceblocks

Zur Feststellung der Erweiterbarkeit des IFBs wird der IFB schrittweise zwischen der SystemC-Simulation und der IP-Emulation aufgebaut. Die Ausgangssituation stellt die Verbindung eines Softwaretasks (SystemC-Simulation) mit einem Hardwaretask (IP-Emulation) dar. Die Verbindung soll mit Hilfe eines Interfaceblocks erfolgen. Abbildung 3.2.8 veranschaulicht die Ausgangssituation.

Zunächst richtet sich die Betrachtung auf den Hardwaretask, da dem Interfaceblock eine Hardwareanwendung zugrunde liegt. Die Verbindung zwischen dem Interfacebock und der Intellectual Property wird durch einen Protokollhandler bewerkstelligt (Abbildung 3.2.9). Er wird im Folgenden als $PH_{\text{HW-out}}$ bezeichnet, da er den hardwareseitigen Ausgang des Interfaceblock bildet.

Die Funktion, die die Schnittstelle $PH_{\text{HW-out}} \leftrightarrow$ IP-Emulation erfüllen muss, ist das Anlegen der Triggerdaten an die IP und das kontrollierte Abgreifen der Tracedaten von der IP während eines Emulationszyklus. Zur Sicherstellung der Datenkonsistenz, also dass die Daten nur aus dem gerade ablaufenden Emulationszyklus stammen, muss dieser Vorgang über Kontrollsignale gesteuert werden. Die Daten, die über die Schnittstelle transportiert werden, müssen bis zu ihrer Verarbeitung verfügbar sein. Aus diesem Grund erfolgt eine Zwischenspeicherung der Daten im IFB. Die Speicherung findet im Modus des $PH_{\text{HW-out}}$ statt, da die Modi der Handler diese Funktionalität bereitstellen können.

Die Verbindung von SystemC-Simulation und Interfaceblock realisiert wiederum ein Protokollhandler. Dieser Protkollhandler hat die Aufgabe, die Kommunikation mit der

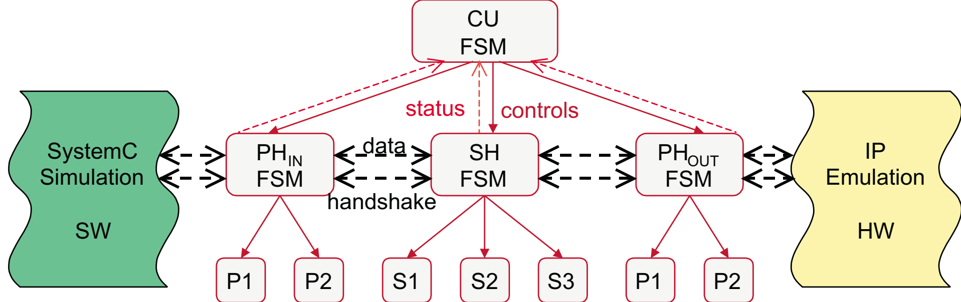


Abbildung 3.2.8.: Verbindung eines SW-Task und eines HW-Task durch einen Interfaceblock

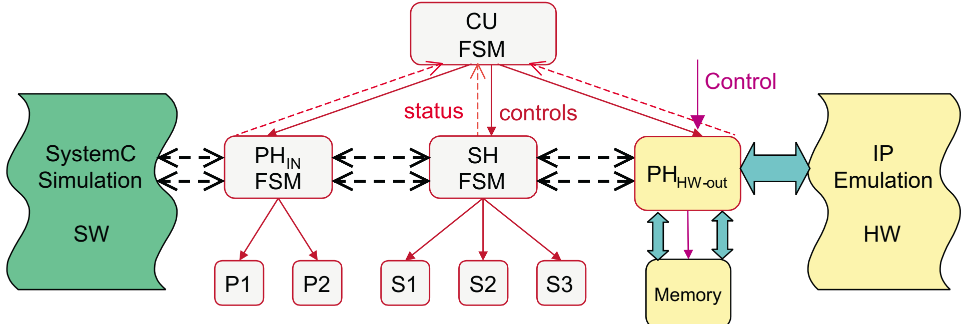


Abbildung 3.2.9.: Die Hardwareschnittstelle des Interfaceblocks, der Protokollhandler$_{\text{HW-out}}$

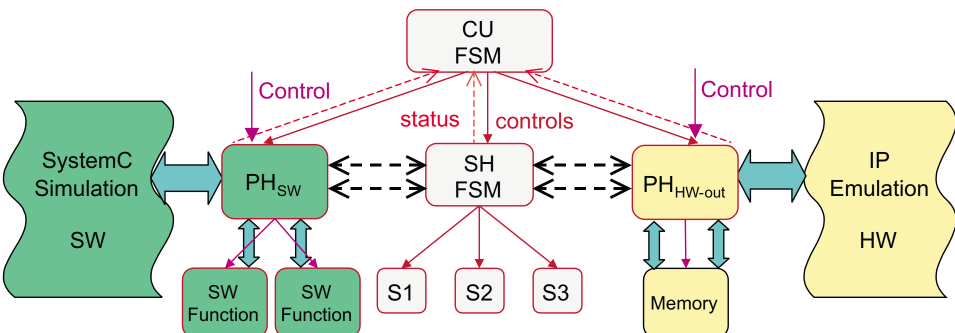


Abbildung 3.2.10.: Die Softwareschnittstelle des Interafceblocks, der Protokollhandler$_{\text{SW}}$

SystemC-Simulation auf Softwareseite sicherzustellen. Im Folgenden wird er als PH$_{\text{SW}}$ bezeichnet (Abbildung 3.2.10). Neu im IFB-Modell ist, dass der Protokollhandler eine SW-Schnittstelle darstellt. An der Aufgabe der Schnittstelle ändert sich nichts. Die Funktionalität wird in Software jedoch durch Funktionen bestimmt, die Algorithmen implementieren und nicht durch kombinatorische Logik und endliche Automaten. Die Datenübergabe findet bei Funktionen über Parameter und nicht durch das Anlegen eines elektrischen Pegels statt. Das stellt aber kein Hinderniss dar. Eine Funktion kann ebenso Parameter auswerten und auch Rückgabewerte erzeugen. Außerdem erlaubt Software die Implementierung von umfangreichen Funktionen. Es spricht also nichts dagegen, dem Interfaceblock eine SW-Schnittstelle in Form eines Softwareprotokollhandlers zu geben. Die Modi liegen ebenfalls in Software und sind durch Funktionen implementiert, die vom PH$_{\text{SW}}$ aufgerufen werden.

Nun stellt sich die Frage, an welche Stelle die physische HW/SW-Schnittstelle platziert wird. Es gibt dabei die Möglichkeiten, sie zwischen PH$_{\text{SW}}$ und SH oder SH und PH$_{\text{HW-out}}$ anzusiedeln. Um einen Großteil des IFB-Modells beizubehalten, wird die physische HW/SW-Schnittstelle als Schnittstelle von PH$_{\text{SW}}$ und SH genutzt (Abbildung 3.2.11). Da der Sequenzhandler in Hardware liegt, wird er weiterhin als SH$_{\text{HW}}$ bezeichnet. In Abschnitt 3.1.1 wurde bereits festgestellt, dass wahrscheinlich eine Transformation der

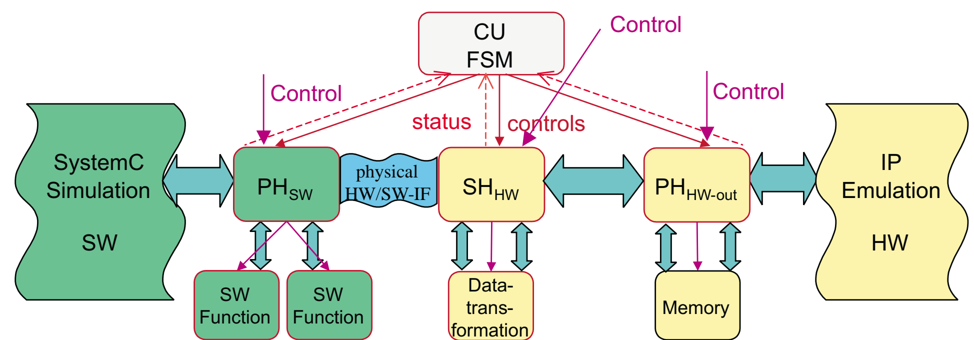


Abbildung 3.2.11.: Die physische HW/SW-Schnittstelle im Interfaceblock

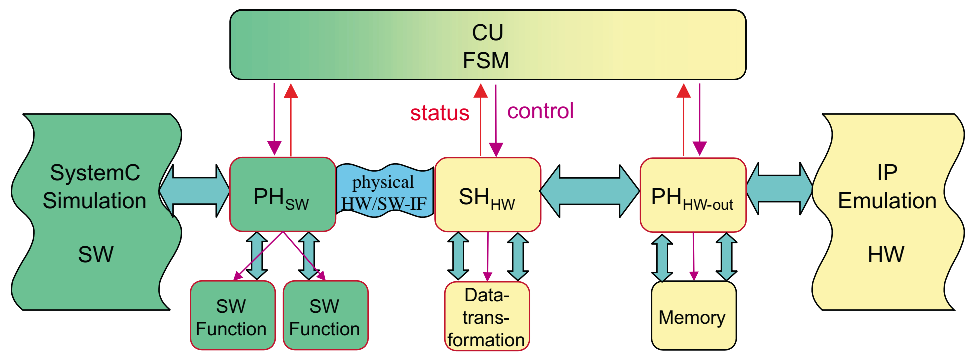


Abbildung 3.2.12.: Controlunit zur Steuerung von Hardware und Software

Daten, die über die physische HW/SW-Schnittstelle transportiert werden, notwendig ist. Die Aufgabe des Sequenzhandlermodus besteht nun darin, die empfangenen und die zu sendenden Daten vom bzw. zum $\mathrm{PH}_{\mathrm{SW}}$ in ein protokollkonformes Format umzuwandeln.

Zur Kontrolle der Abläufe innerhalb des IFB kommt die Controlunit zum Einsatz. Sie hat die Aufgabe, die Handler zu steuern und zu überwachen. Bei der Verwendung des IFBs als HW/SW-Schnittstelle, muss der $\mathrm{PH}_{\mathrm{HW\text{-}out}}$, der $\mathrm{SH}_{\mathrm{HW}}$ aber auch der $\mathrm{PH}_{\mathrm{SW}}$ von der Controlunit gesteuert werden. Das erfordert, dass die Controlunit Software als auch Hardware ansteuern kann (Abbildung 3.2.12).

Die physische Trennung von Hardware- und Softwareseite des IFBs, gestaltet die Steuerung durch eine einzige Controlunit schwierig. Aus diesem Grund erfolgt eine Teilung der Controlunit in einen Softwareteil ($\mathrm{CU}_{\mathrm{SW}}$) und einen Hardwareteil ($\mathrm{CU}_{\mathrm{HW}}$), wie Abbildung 3.2.13 zeigt. Der Vorteil liegt nun darin, dass die jeweilige Controlunit ihren Teil direkt kontrollieren kann, ohne Steuersignale über die HW/SW-Grenze zu schicken und damit größere Verzögerungszeiten im Kontrollfluss zu verursachen.

Ein Problem, welches durch die Teilung auftritt, ist die Kommunikation zwischen den Controlunits. Zur geregelten Steuerung des IFBs und zur Synchronisation zwischen

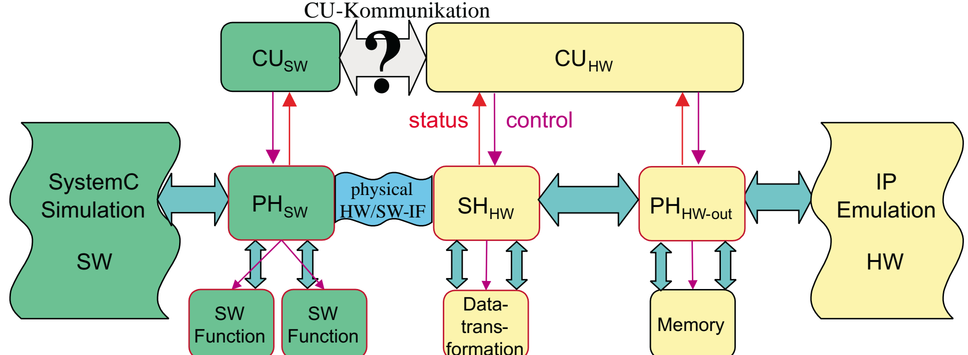


Abbildung 3.2.13.: Teilung der Controlunit in Hardware- und Softwareteil

Hardware- und Softwareteil ist eine Kommunikation zwischen CU_{SW} und CU_{HW} notwendig. Als Hinderniss ist dabei die HW/SW-Grenze zu überwinden. Dies kann auf zwei Arten erfolgen. Zum Einen über eine zweite physische HW/SW-Schnittstelle, die allein für den Transport und Austausch von Kontroll- und Statussignalen zwischen den Controlunits zuständig ist. Die Lösung bedeutet einen höhere Aufwand, in Bezug auf die Bereitstellung von Hardware. Außerdem muss eine zweite Schnittstelle in Software als auch in Hardware implementiert werden.

Die zweite Möglichkeit besteht darin, die Kommunikation über die physische HW/SW-Schnittstelle zu bewältigen, über die auch die Simulationsdaten ausgetauscht werden. Der Vorteil dieser Variante ist die Nutzung der vorhandenen physischen Schnittstelle ohne zusätzlichen Hardwareaufwand. Allerdings benötigt man jetzt ein zusätzliches Protokoll, dass über dem Protokoll der HW/SW-Schnittstelle angesiedelt ist, um eine Unterscheidung zwischen Simulationsdaten und Controlunitdaten zu ermöglichen. Dies bringt die Einführung eines weiteren Protokollhandlers auf der Hardwareseite des IFBs mit sich (Abbildung 3.2.14). Er wird im Folgenden als $PH_{HW\text{-}in}$ bezeichnet. Diese Maßnahme ist notwendig, da der Sequenzhandler laut Definition nur für die Änderung von Daten zuständig ist. Die Verarbeitung von Protokollen ist dem Protokollhandler vorbehalten.

Die CU-Kommunikation könnte also wie folgt ablaufen. Die CU_{SW} will ein Kontrollsignal an die CU_{HW} schicken. Dazu leitet sie das Signal an den PH_{SW} weiter, welcher es in das Protokoll zur Datenunterscheidung umwandelt und es mittels des normalen HW/SW-Protokoll an den Hardwareteil des IFBs sendet. Die Stelle im Hardwareteil, an dem die Daten empfangen werden, ist der $PH_{HW\text{-}in}$. Er prüft, ob die empfangenen Daten für die Simulation oder für die Controlunit bestimmt sind. Die Kontrollsignale werden an die CU_{HW} weitergeleitet. Sie kann die Signale auswerten und eine Antwort an die CU_{SW} über den gleichen Weg zurücksenden.

Als Ergebnis liegt jetzt ein erweiterter Interfaceblock vor. Das heißt, dass das Modell des Interfaceblocks dahingehend erweiterbar ist, um damit auch eine HW/SW-Schnittstelle

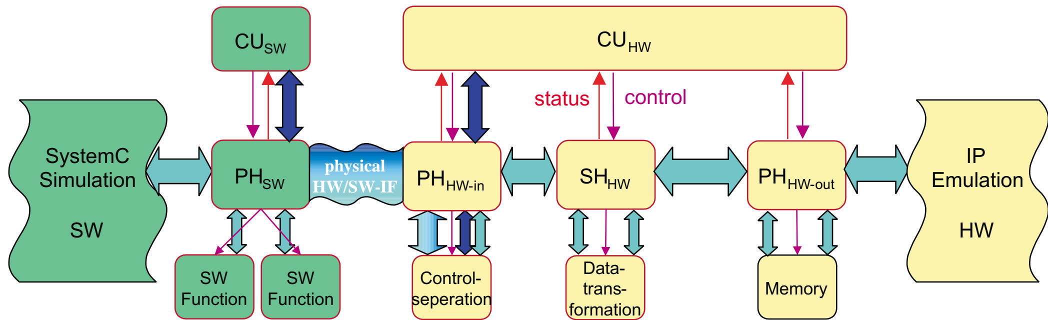


Abbildung 3.2.14.: Controlunit in Software und Hardwareteil geteilt

zu realisieren. Das Modell des Interfaceblocks ist also nicht auf reine Hardwareanwendungen begrenzt. Um dieses Ergebnis weiter zu untermauern, folgt den bisher theoretischen Betrachtungen, eine mögliche technischen Umsetzung dieses Modells am Beispiel des Simulationsinterfaceblocks.

3.3. Technische Umsetzung der Adaptierung durch einen Simulationsinterfaceblock

Dieser Abschnitt stellt eine Möglichkeit der technischen Umsetzung eines erweiterten Interfaceblocks dar. Da die Erweiterung speziell für die Kopplung von SystemC-Simulation und IP-Emulation zugeschnitten ist, wird der erweiterte Interfaceblock im weiteren Verlauf dieser Arbeit als **Simulationsinterfaceblock (SimIFB)** bezeichnet.

3.3.1. Die hardwareseitige Implementierungsplattform

Als Implementierungsplattform für den Hardwareteil und die Emulation der Intellectual Property kommt das Digilent 2E Development Board (Abbildung 3.3.15) zum Einsatz. Es ist eine Entwicklungsplatine, die mit einem Xilinx Spartan2E FPGA-Chip versehen ist. Die genaue Bezeichnung des FPGA-Chips lautet XC2S200E-PQ208. Mit dem FPGA können bis zu 200 000 Gatteräquivalente nachgebildet werden. Die Entwicklungsplatine

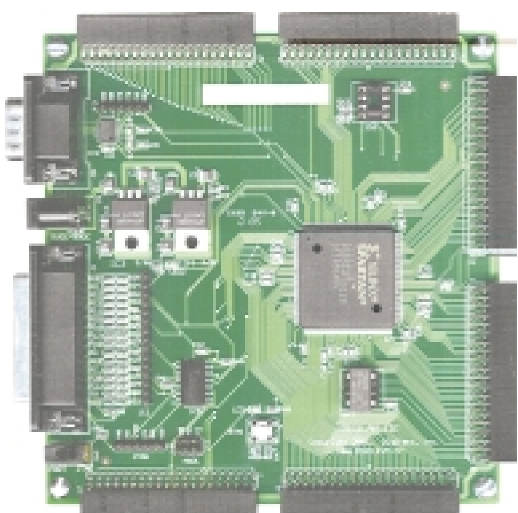


Abbildung 3.3.15.: Digilent 2E Developmentboard mit Xilinx Spartan 2E FPGA, Quelle [Dig]

verfügt außerdem über mehrere Extensionheader sowie über eine serielle, eine parallele und eine JTAG-Schnittstelle.

Mehr Informationen zur Entwicklungsplatine bietet das dazugehörige Handbuch [Dig02]. Erläuterungen zum Xilinx Spartan 2E FPGA enthält das Datenblatt [Xil03].

3.3.2. Umsetzung des HW/SW-Interfaces

Als physische Hardware/Software-Schnittstelle kommen unter Nutzung des Digilent 2E nur dessen Schnittstellen in Frage. Dass heißt, eine Verbindung ist über den seriellen, den parallelen, den JTAG-Port oder über den Extensionheader möglich.

In Abschnitt 3.1.1 wurde die Verwendung von Standardschnittstellen eines PCs angedacht. Ein PC stellt standardmäßig nur die serielle und die parallele Schnittstelle bereit. Somit entfällt die Verwendung von JTAG-Port und Extensionheader.

Die Wahl der Schnittstelle hängt nun von der Leistung der Schnittstelle ab. Tabelle 3.3.1 stellt die serielle und die parallele Schnittstelle einander gegenüber. Das wichtigste Leistungsmerkmal für die Verbindung der SystemC-Simulation und der IP-Emulation stellt die Übertragunsgeschwindigkeit dar. Deshalb findet die parallele Schnittstelle für die HW/SW-Schnittstelle Verwendung.

Die Kommunikation über die parallele Schnittstelle findet über das EPP-Protokoll (Enhanced Parallel Port) statt. Das EPP-Protokoll ist im IEEE-Standard 1284 definiert. Die

Merkmal	**serielle Schnittstelle nach RS-232**	**parallele Schnittstelle nach IEEE 1284 (EPP)**
Übertragunsart	seriell asynchron	parallel synchron
Übertragungsmodi	senden empfangen	Daten senden Adressen senden Daten lesen Adressen lesen
Synchronisation	Signalflanke	Handshake
maximale Geschwindigkeit	115 200 Bit/s = 14 400 Byte/s	2 MByte/s
Anzahl der Datenbits pro Zyklus	8	8
Fehlererkennung	Parität	-

Tabelle 3.3.1.: Vergleich von serieller Schnittstelle nach RS-232 und paralleler Schnittstelle nach IEEE 1284

Nutzung des EPP-Protokolls bringt einige Eigenschaften mit sich, die für die HW/SW-Schnittstelle und deren Umsetzung von Bedeutung sind.

Zum Einen ist das Protokoll ein Master-Slave-Protokoll. Dass heißt, die Kommunikation kann nur von einem Kommunikationspartner initiiert werden. Im Abschnitt 3.1.1 wurde festgestellt, dass es sich bei der vorliegenden Verbindung von SystemC-Simulation und IP-Emulation um eine Form der Master-Slave-Cosimulation handelt. Somit stellt diese Eigenschaft des Protokolls kein Hinderniss dar. Den Master der HW/SW-Schnittstelle bildet die Softwareseite, also der PC auf dem die SystemC-Simulation abläuft. Die Hardwareseite des Systems (Hardwareseite des IFBs und IP-Emulation) stellt den Slave dar. Somit existiert auch innerhalb des IFB eine Master-Slave-Beziehung zwischen den Controlunits. Doch dazu mehr im Abschnitt 3.3.7.

Eine weitere Eigenschaft des EPP-Protokolls, die für die Trennung von Datenfluss und Steuerfluss über die HW/SW-Schnittstelle von Bedeutung ist, bilden die vier verschiedenen Übertragungsmodi. Mit Hilfe des Protokolls ist der Master in der Lage, Daten und Adressen zu schreiben bzw. zu lesen. Die Modi werden durch unterschiedliche Formen des Handshakes realisiert. Zur Veranschaulichung stellt Abbildung 3.3.16 den Protokollautomaten des EPP-Protokoll auf der Slaveseite dar. Ausführlichere Informationen zur parallelen Schnittstele finden sich in [Axe97], [Pea04] und [War].

Die unterschiedlichen Übertragungsmodi bedeuten für die HW/SW-Schnittstelle, dass für die Kommunikation eine Adressbreite von 8 Bit zur Verfügung steht. Mit jeder Adresse können 8 Bit Daten adressiert werden. Mit 8 Bit lassen sich $2^8 = 256$ verschiedene

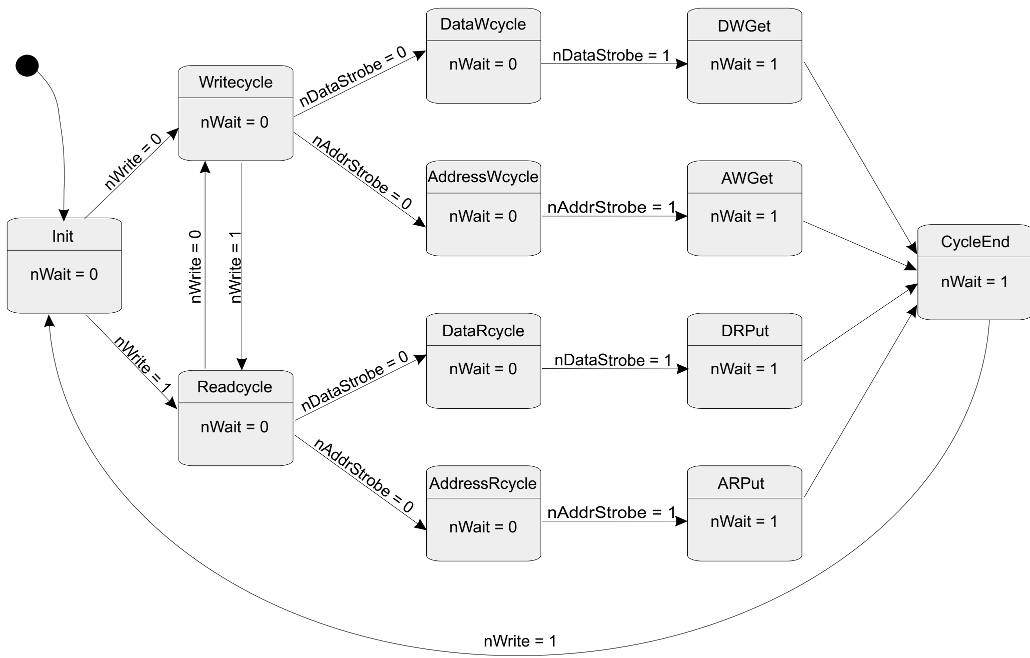


Abbildung 3.3.16.: Protokollautomat des EPP-Protokoll

Adressen darstellen. Da jeder Adresse 8 Bit Daten zugewiesen werden können, ergeben sich daraus $256 \cdot 8\,\text{Bit} = 2048\,\text{Bit}$ Daten, die adressiert geschrieben und 2048 Bit Daten, die adressiert gelesen werden können. Mit Hilfe dieses Mechanismus ist es möglich, die Daten mit einer Semantik zu versehen.

Auf dieser Basis setzt das Protokoll zur Datenunterscheidung auf. Es dient dazu, die Simulationsdaten von den Kontrollsignalen der Kommunikation zwischen den Controlunitteilen zu trennen. Allerdings wird hier nicht die gesamte Leistung der EPP-Schnittstelle genutzt. Adressen werden hier nicht doppelt belegt. Dass heißt, mit einer Adresse kann entweder geschrieben oder gelesen werden. Das Protokoll zur Datenunterscheidung beruht auf einer Teilung des Adressraumes. Die ersten 16 Adressen (00h[2] bis 0Fh) dienen zur Kommunikation der Controlunits. Diese haben somit $16 \cdot 8\,\text{Bit} = 128\,\text{Bit}$ für Kontrollsignale zur Verfügung. Für den Austausch von Simulationsdaten stehen die Adressen 10h bis FFh zur Verfügung. Die ersten Adressen ab 10h stehen den Eingängen der IP zur Verfügung und sind somit schreibbare Adressen. Im Anschluss daran folgen die lesbaren Adressen. Die erste lesbare Adresse (FRA, first readable address) berechnet sich aus der Formel $\text{FRA} = 10\text{h} + \lceil \frac{\text{Anzahl Eingangsbits der IP} - 1}{8} \rceil$. Somit ergibt sich für die maximale Anzahl von Ein- und Ausgängen der IP $(256 - 16) \cdot 8 = 1920$.

[2]Hexadezimale Zahlendarstellung

3.3.3. Der PH$_{sw}$

Die Aufgaben des Protokollhandlers der Softwareseite lauten wie folgt:

- Schnittstelle zur SystemC-Simulation bereitstellen
- Schnittstelle zum Parallelport des PCs implementieren
- Simulationsdaten senden
- Simulationsdaten lesen
- Kommunikation der CU$_{SW}$ und CU$_{HW}$ erlauben

Eine Anforderung an den Simulationsinterfaceblock besteht nach Abschnitt 3.1.1 darin, der SystemC-Simulation eine SW-Schnittstelle bereitzustellen, bei der der Simulationskern nicht verändert werden muss. Aus diesem Grund wird der PH$_{SW}$ als SystemC-Modul implementiert. Als Schnittstelle kommen die SystemC-Ports *sc_in<>* und *sc_out<>* zum Einsatz. Die Eingänge und Ausgänge der IP bestimmen die Breite und Anzahl der Ports. Lokale Variablen nehmen die Werte der Ports für die weitere Verarbeitung auf (Abbildung 3.3.17).

Der Zugriff auf den Parallelport des PCs wird über die Bibliothek *dlportio.lib* hergestellt. Diese Bibliothek wurde nicht in dieser Arbeit entwickelt, sondern als bereits fertige Lösung von der Website www.driverlinx.com [Scib] bezogen. Das Programm *port95nt.exe* [Scia] von der Website muss auf dem Rechner installiert sein, um die Bibliothek nutzen zu können.

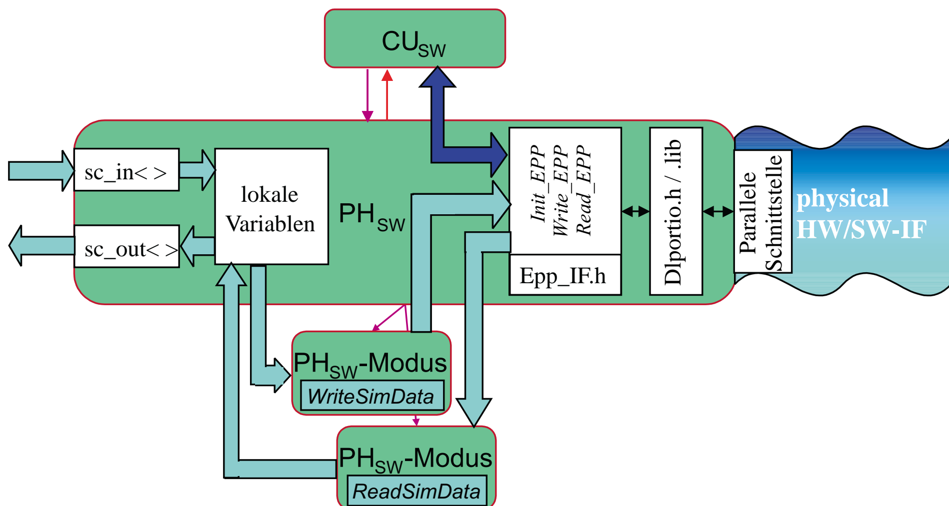


Abbildung 3.3.17.: Detailierter Aufbau des PH$_{SW}$

Funktion	Parameter	Aufgabe
Init_EPP	→BADDR	Initialisierung der Schnittstelle
Write_EPP	→BADDR →Adresse →Datenbyte	Schreiben des Datenbyte auf die angegebene Adresse
Read_EPP	→BADDR →Adresse ←Datenbyte	Lesen des Datenbytes von der angegebenen Adresse

Tabelle 3.3.2.: Grundfunktionen zum Zugriff auf die parallele PC-Schnittstelle (BADDR = Basisadresse des Parallelports über den die Kommunikation stattfinden soll; → Eingangsparameter, ← Rückgabeparameter)

Auf die Funktionen der Bibliothek bauen die drei Funktionen *Init_EPP, Write_EPP* und *Read_EPP* auf. Ihre Parameter und Aufgaben führt Tabelle 3.3.2 auf. Nur über diese Funktionen kann auf die parallele Schnittstelle zugegriffen werden. Diesen Weg nutzen alle Funktionen der Softwareseite des IFB, die auf die Schnittstelle zugreifen müssen. Das betrifft die PH_{SW}-Modi und die CU_{SW}. Der Zugriff über fest definierte Funktionen bringt den Vorteil, dass die Bibliothek *dlportio.lib* auch ausgetauscht werden kann und nur diese Funktionen angepasst werden müssen, falls eine andere Art des Schnittstellenzugriffs implementiert werden soll.

Den Transport der Simulationsdaten übernehmen die Modi des PH_{SW}. Den Modus zum Schreiben realisiert die Funktion *WriteSimData*. Den für das Lesen zuständigen Modus bildet die Funktion *ReadSimData*. Die Übergabeparameter sind bei beiden Funktionen gleich. Sie bestehen aus der Basisadresse des Parallelports, einem Zeiger auf ein Feld, dass die Simulationsdaten enthält bzw. aufnimmt und einem Wert für die Größe des Feldes in Byte.

Den Ablauf eines Funktionszyklus des PH_{SW} stellt Abbildung 3.3.18 dar. Am Anfang erfolgt das Übertragen der Eingänge auf lokale Variablen. Sie dienen als Zwischenspeicher, um die Daten auf die Breite der Eingänge der IP zu konvertieren. Anschließend wird geprüft, ob das asynchrone Resetsignal aktiv ist. Es besitzt Vorrang vor dem Taktsignal, da auch bei Registern das asynchrone Resetsignal die höhere Priorität hat. Ist das Signal aktiv, dann setzt die CU_{SW}-Funktion *SetWriteMode* die CU_{HW} in den Schreibmodus. Nun beginnt das Setzen der Zähler und das Übertragen der Simulationsdaten. Die Ausführung des asynchronen Reset für die IP-Emulation schließt sich an. Nachdem dieser Schritt beendet ist, setzt die Funktion *SetReadMode* die CU_{HW} in den Lesemodus. Das Auslesen der Emulationsergebnisse erfolgt in lokale Variablen, die zum Schluss auf die Ausgänge übertragen werden.

Ist das asynchrone Resetsignal nicht aktiv, so findet die Prüfung auf Aktivität des Taktsignals statt. Die einzelnen Schritte sind denen der Resetausführung gleich. Der einzige Unterschied besteht darin, dass nach dem Schreiben der Simulationsdaten kein Reset

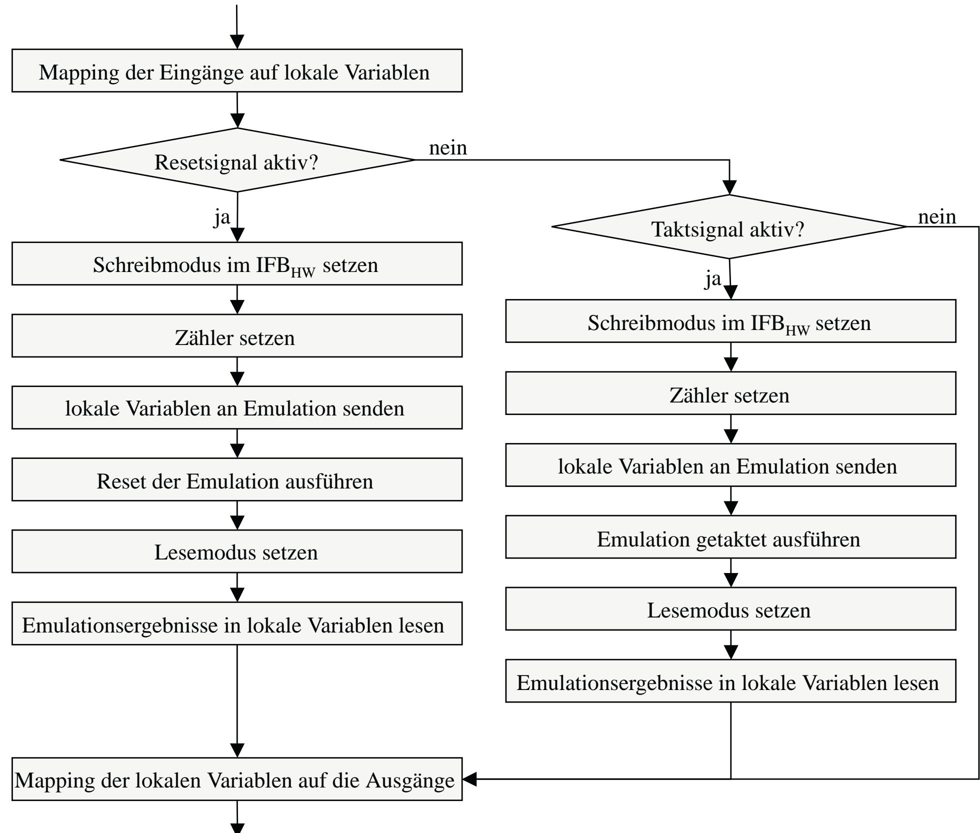


Abbildung 3.3.18.: Ablaufplan zur Umsetzung des PH_{SW}

durchgeführt wird, sondern das Taktsignal, entsprechend der gesetzten Zählerwerte, an die IP-Emulation angelegt wird.

3.3.4. Der $PH_{HW\text{-}in}$

Der hardwareseitige Protokollhandler (Abbildung 3.3.19), der die Schnittstelle zur physischen HW/SW-Schnittstelle bildet, hat folgende Aufgaben:

- Kommunikation über die parallele Schnittstelle realisieren
- Trennung von Simulationsdaten und CU-Kommunikation

Für die Kommunikation über die parallele Schnittstelle implementiert der Modus des $PH_{HW\text{-}in}$ den Protokollautomaten für die Slave-Seite des EPP-Protokolls. Abbildung 3.3.16 stellt den Protokollautomaten dar. Es werden 4 EPP-Übertragungsarten durch den Protokollautomaten implementiert. Dazu zählen das byteweise Lesen von Adressen,

Schreiben von Adressen, Lesen von Daten und Schreiben von Daten. Für die vorliegenden Aufgaben wird jedoch die Funktion des Lesens von Adressen nicht benötigt. Der Protokollautomat stellt in Registern die empfangene Adresse und die empfangenen Daten bereit. Ein weiteres Register speichert Daten zwischen, die von der Softwareseite abgerufen werden.

Die empfangene Adresse ist Grundlage für die zweite Aufgabe des $PH_{HW\text{-}in}$, die Trennung von Simulationsdaten und CU-Kommunikation. Die CU-Kommunikation findet über die Adressen 00h bis 10h statt. Dieser Adressraum ist wiederum in einen schreibbaren (00h bis 07h) und einen lesbaren Teil (08h bis 10h) unterteilt. Es stehen also 8 Adressen zum Lesen und 8 zum Schreiben bereit. Die genaue Belegung und Funktion der Adressen ist im Abschnitt 3.3.7 beschrieben. Alle Adressen, die größer als 10h sind, werden unverändert an den SH_{HW} weitergegeben. Die Auswertung dieser Adressen ist nicht Aufgabe der $PH_{HW\text{-}in}$.

Der endliche Automat Handlercontrol (Abbildung 3.3.19) dient allein zur internen Kontrolle des Handlers und zum Starten eines Modus. Den Datentransport durch den IFB beeinflusst er nicht.

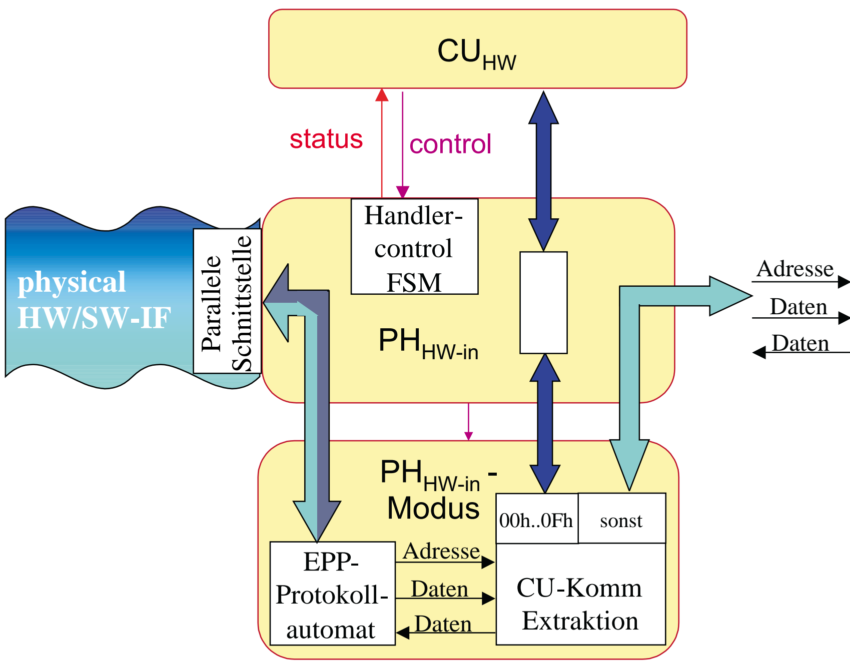


Abbildung 3.3.19.: Detailierter Aufbau des $PH_{HW\text{-}in}$

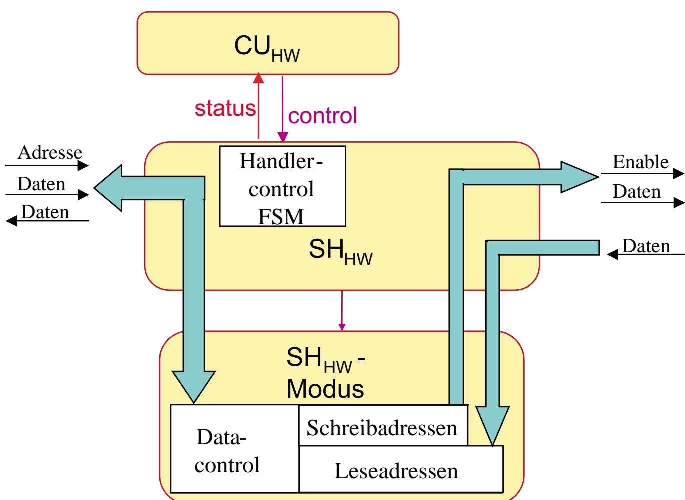


Abbildung 3.3.20.: Detailierter Aufbau des SH_{HW}. Die Werte der Schreibadressen bilden den Bereich nach Formel 3.1, die Leseadressen nach Formel 3.2

3.3.5. Der SH_{HW}

Die Aufgabe des Sequenzhandlers im Hardwareteil des Simulationsinterfaceblocks liegt in der korrekten Verteilung der Simulationsdaten. Durch die HW/SW-Schnittstelle liegen die Daten in Form von Tupel (Adresse, Datenbyte) vor. Anhand der Adresse muss nun zuerst eine Unterscheidung getroffen werden, ob es sich um eine Schreibadresse oder eine Leseadresse handelt.

$$\text{Schreibadressen:} \quad 10\text{h} \qquad \ldots 10\text{h} + \left\lceil \frac{n_i - 1}{8} \right\rceil - 1 \tag{3.1}$$

$$\text{Leseadressen:} \quad 10\text{h} + \left\lceil \frac{n_i - 1}{8} \right\rceil \ldots 10\text{h} + \left\lceil \frac{n_i - 1}{8} \right\rceil + \left\lceil \frac{n_o}{8} \right\rceil - 1 \tag{3.2}$$

n_i ... Anzahl Eingangsbits der emulierten IP mit Taktsignal
n_o ... Anzahl Ausgangsbits der emulierten IP

Den Bereich der Schreibadressen gibt Formel 3.1 an. Er ist abhängig von der Anzahl der Eingangsleitungen n_i der emulierten IP. Da das Taktsignal gesondert behandelt wird, muss es von den Eingangsleitungen abgezogen werden. Die niedrigste Adresse, die überhaupt Daten transportieren kann, stellt die 10h dar.
Die Leseadressen schließen sich an den Bereich der Schreibadressen an. Sie sind zusätzlich abhängig von der Anzahl der Ausgangsleitungen der emulierten IP. Den Adressraum gibt Formel 3.2 an.

Nach der Dekodierung der Adresse werden die Daten zugeordnet. Für die Schreibadressen kommt dazu ein Demultiplexer zum Einsatz, da das Datenbyte je nach Adresse auf unterschiedliche Leitungen gelegt werden muss. Außerdem werden Enablesignale erzeugt, die zur Kommunikation mit dem $\text{PH}_{\text{HW-out}}$ dienen. Für die Abfrage von Leseadressen kommt ein Multiplexer zum Einsatz. Er legt die Daten vom $\text{PH}_{\text{HW-out}}$, die der aktuellen Adresse entsprechen, auf die Datenleitung zum $\text{PH}_{\text{HW-in}}$.

3.3.6. Der $\text{PH}_{\text{HW-out}}$

Die Aufgabe des Protokollhandlers der Hardwareseite, der die Schnittstelle zur IP-Emulation bildet, besteht darin, einen Speicher für die Simulationsdaten bereitzustellen. Zu speichern sind sowohl die anzulegenden Daten als auch die Emulationsergebnisse. Zusätzlich muss der $\text{PH}_{\text{HW-out}}$ noch einen Mechanismus enthalten, mit dem eine kontrolliert Auslösung des Taktsignals für die Emulation möglich ist.

Für jedes Eingangs- sowie Ausgangssignal der emulierten IP wird ein eigenständiger Speicher benötigt. Deshalb kommen Flipflops als Speicher zum Einsatz. Da die IP n_i Eingänge und n_o Ausgänge besitzt, muss der Modus des $\text{PH}_{\text{HW-out}}$ $n_i + n_o$ Flipflops zur Speicherung der Simulationsdaten bereitstellen.

Um eine Steuerung der Datenspeicherung zu erlauben, werden die Flipflops mit Enable-Signalen gesteuert. Die Flipflops mit den Eingangsdaten der IP werden durch den SH_{HW} gesteuert. Die Flipflops, die die Ausgangswerte der IP übernehmen, durch ein Kontrollsignal von der CU_{HW}, da diese die genaue Gültigkeit der Daten kennt.

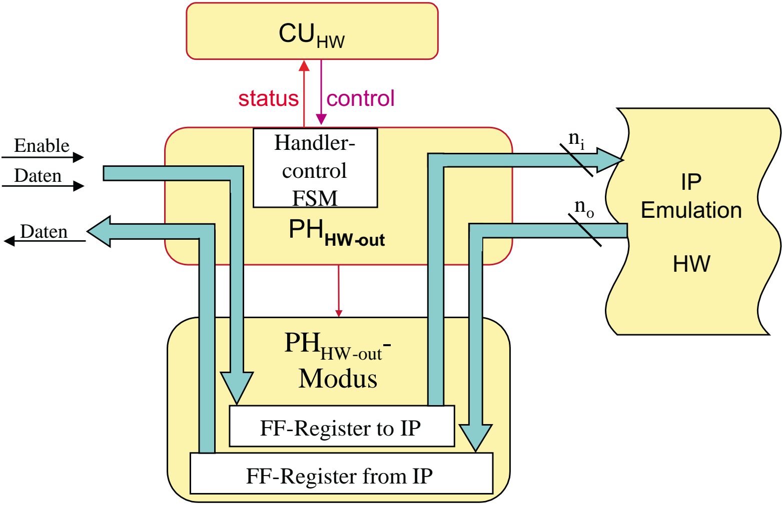


Abbildung 3.3.21.: Detailierter Aufbau des $\text{SH}_{\text{HW-out}}$

Unter den n_i Flipflops für die Eingangssignale der IP befindet sich auch ein Flipflop für das Taktsignal. Es wird mittels eines Kontrollsignals von der CU_{HW} gesetzt oder gelöscht, da die CU_{HW} die Taktsteuerung implementiert.

3.3.7. Die Controlunit

Die Controlunit (Abbildung 3.3.22) besteht aus einem Hardware- (CU_{HW}) und einem Softwareteil (CU_{SW}), die beide zusammenarbeiten. Sie kommunizieren miteinander über die HW/SW-Schnittstelle mit Hilfe von PH_{SW} und $PH_{HW\text{-}in}$. Die Grundlagen der Kommunikation wurden bereits in den Abschitten 3.3.3 und 3.3.4 erläutert.

In den Abschnitten 3.1.1 und 3.2.3 wurde herausgearbeitet, dass zwischen der SystemC-Simulation und der IP-Emulation eine Master-Slave-Beziehung besteht. Diese Beziehung wird auf den IFB übertragen, so dass der Softwareteil, also die CU_{SW}, die Rolle des Masters annimmt und der Hardwareteil, also die CU_{HW}, die Rolle des Slave.

Zunächst soll der Hardwareteil der Controlunit näher betrachtet werden. Er besteht aus einem Automaten zur Kontrolle des Interfaceblocks, der zusätzlich Kontrollregister, Statusregister und Zähler beinhaltet, und eine Komponente Autoreset. Über die Kontrollregister erhält der Automat Steuersignale von der CU_{SW}. Die Statusregister erlauben der CU_{SW}, den aktuellen Zustand des Hardwareteils abzufragen. Die Funktion der Kontroll- und Statusregister, die sie erfüllen, sowie die Adresse, über die sie angesprochen werden, führen Tabelle 3.3.3 und 3.3.4 auf.

Das Zustandsdiagramm der endlichen Automaten stellt Abbildung 3.3.23 dar. Aus Gründen der Übersichtlichkeit sind die Ausgaben der Zustände in Tabelle 3.3.5 aufgeführt. Zu Beginn führen die Zustände *Start_PHin*, *Start_SH* und *Start_PHout* das Starten der Hand-

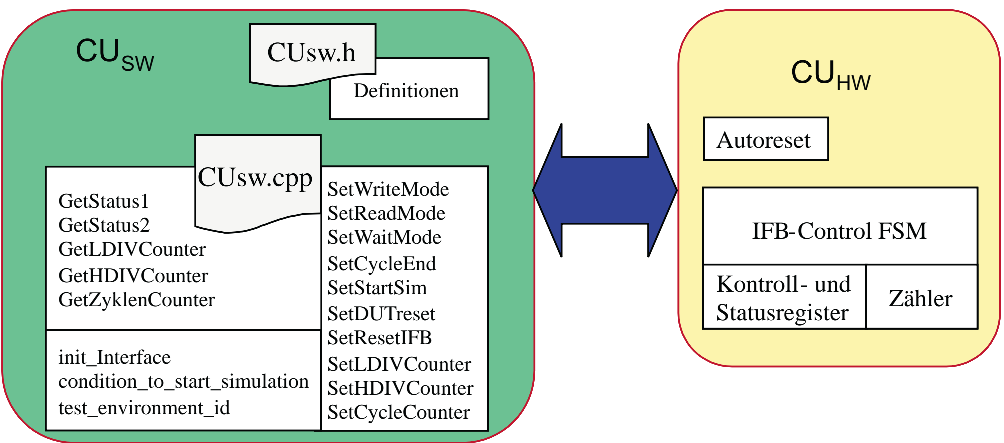


Abbildung 3.3.22.: Detailierter Aufbau der Controlunits

Adresse	w/r	Funktion
00 h	w	Neutrale Adresse
01 h	w	Kontrollregister
02 h	w	LDIV Zähler
03 h	w	HDIV Zähler
04 h	w	Zyklenzähler High-Byte
05 h	w	Zyklenzähler Low-Byte
06 h	w	keine
07 h	w	keine
08 h	r	Statusregister1
09 h	r	Statusregister2
0A h	r	LDIV Zählerwert
0B h	r	HDIV Zählerwert
0C h	r	Zyklenzählerwert High-Byte
0D h	r	Zyklenzählerwert Low-Byte
0E h	r	keine
0F h	r	Environment ID

Tabelle 3.3.3.: Die Register der hardwareseitigen Controlunit. (w - schreibbar, r - lesbar)

lerautomaten durch. Anschließend wartet der Automat im Zustand *Waiting* auf Signale der CU_{SW}.

Von diesem Zustand aus erreicht der Automat durch Setzen des Signals *cwrite* bzw. *cread* die Zustände *DWrite* bzw. *DRead*. In diesen Zuständen sollten die Simulationsdaten geschrieben bzw. gelesen werden. Dies ist nicht zwingend notwendig. Um jedoch ein versehentliches Starten der Emulation zu verhindern, sollten diese Funktionen in den Zuständen ausgeführt werden.

In den Zustand *NoAction* geht der Automat, falls das Startsignal gesetzt ist, aber noch kein Wert für die Anzahl der Simulationszyklen im Zyklenzähler eingetragen ist. Dies verhindert, dass ein Emulationszyklus durchlaufen wird ohne eine Angabe der Zyklenanzahl.

Der Zustand *DUTreset* führt einen asynchronen Reset der IP-Emulation durch. Da der asynchrone Reset dem Taktsignal übergeordnet und von ihm unabhängig ist, wird die Emulation dabei ohne Taktsignal ausgeführt. Die Speicherung der Ausgabedaten der Emulation erfolgt dabei wie in einem normalen Taktzyklus.

Beim Start der Emulation durch das Signal *cstart* wechselt der Automat in den Zustand *DPrepare* falls der Zyklenzähler einen Wert ungleich Null enthält. Der Zustand dient der Vorbereitung auf die Emulation. Es werden die Flipflops zur Aufnahme der Emulationsdaten im $PH_{HW\text{-}out}$-Modus freigegeben und die Zähler LDIV und HDIV geladen.

Bit	7	6	5	4	3	2	1	0
Kontrollregister	creset	-	-	cdutreset	cnext	cstart	cwrite	cread
Statusregister1	phin_runs	sh_runs	phout_runs	snext	swork	sready	swrite	sread
Statusregister2	-	-	-	-	-	zero_cycles	zero_hdiv	zero_ldiv

Tabelle 3.3.4.: Belegung der Kontroll- und Statusregister der CU_{HW}

Anschließend wechselt der Automat in den Zustand *Clk_gen1*, in dem die Taktgenerierung beginnt. Der Zustand *Clk_gen1* erzeugt den Highpegel in positiver Logik während der Zustand *Clk_gen0* den Lowpegel generiert. Die Dauer der generierten Pegel bestimmten die Werte der Zähler HDIV und LDIV. Mit Hilfe der Zähler ist es möglich, die IP-Emulation wahlweise mit einem symmetrischen oder einem asymmetrischen Taktsignal zu betreiben. Die Frequenz, mit der die IP emuliert wird, berechnet sich nach der Formel $f_{IP} = \frac{f_{IFB}}{HDIV+LDIV+2}$. Die Herleitung der Formel beschreibt Anhang A. Um die gewünschte Anzahl an Taktperioden zu erzeugen, werden die Zustände *Clk_gen1* und *Clk_gen0* so oft nacheinander abgearbeitet, bis der Zyklenzähler den Wert 0 erreicht hat.

Nach Abschluss der Taktgenerierung wechselt der Automat in den Zustand *End_Cycle*. Dieser Zustand sperrt die Flipflops des $PH_{\text{HW-out}}$-Modus wieder, damit die aufgenommenen Emulationsdaten nicht verfälscht werden können. Der Automat verweilt in diesem Zustand, bis das Signal *cnext* oder *cread* gesetzt wird.

Das Signal mit der obersten Priorität ist *creset*. Es bringt den Automaten in den Zustand *DoReset*. Er löst ein Signal für die Komponente Autoreset aus. Das Signal veranlasst die Komponente, einen asynchrone Reset für die Hardwareseite des Interfaceblocks auszulösen. Somit ist es möglich, den Interfaceblock durch ein Signal der Softwareseite rückzusetzen.

Der Softwareteil der Controlunit setzt sich aus einem Definitionsteil und einem Funktionsteil zusammen. Die Definitionen werden in der Datei *CUsw.h* getroffen. Sie dienen zum Festlegen der Adressen der Kontroll- und Statusregister des Hardwareteils, sowie der Festlegung von vordefinierten Werten, mit denen zum Beispiel die Kontrollregister gesetzt und die Statusregister ausgewertet werden können. Diese Definitionen stehen der gesamten Softwareseite des Interfaceblocks zur Verfügung.

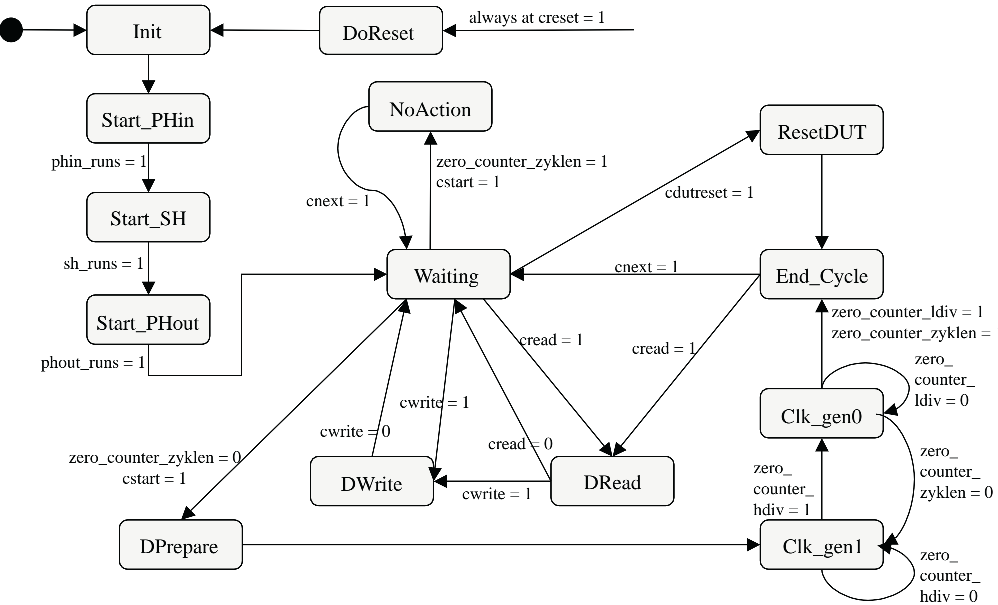


Abbildung 3.3.23.: FSM der Controlunit. Die Ausgaben der Zustände sind in Tabelle 3.3.5 zu finden.

Die Funktionen der CU_{SW} sind in Tabelle 3.3.6 zusammengefasst. Sie teilen sich wie folgt in drei Gruppen:

- Funktionen zum Schreiben in die Kontrollregister
- Funktionen zum Abruf der Statusregister
- Hilfsfunktionen

Die Funktionen zum Schreiben in die Kontrollregister ermöglichen es, Funktionen im Hardwareteil des Interfaceblocks auszulösen. Der Automat kann in verschiedene Zustände versetzt werden, die Emulation gestartet oder der Interfaceblock zurückgesetzt werden. Außerdem erlauben die Funktionen das Setzen der Zähler HDIV, LDIV und das Taktzyklenzählers.

Der Abruf von Statusregistern dient der CU_{SW} zur Information über den Zustand der CU_{HW}. Außerdem können die aktuellen Werte der Zähler abgerufen werden.

Die Hilfsfunktionen haben die Aufgabe, die Arbeit des Softwareteils des Interfaceblocks zu unterstützen. So sind Funktionen zur Initialisierung der parallelen Schnittstelle, zum Testen von Signalen auf eine bestimmte Bedingung vorhanden. Außerdem eine Funktion, die vor der Aufnahme der Kommunikation mit der Hardwareseite sicherstellen soll, dass sich der richtige Interfaceblock auf dem FPGA befindet.

Zustand / **Signal**	Init	Start_PHin	Start_SH	Start_PHout	Waiting	DWrite	DRead	DPrepare	Clk_gen1	Clk_gen0	NoAction	End_Cycle	DoReset	ResetDUT
phin_start	0	1	0	0	0	0	0	0	0	0	0	0	0	0
phin_stop	1	0	0	0	0	0	0	0	0	0	0	0	1	0
sh_start	0	0	1	0	0	0	0	0	0	0	0	0	0	0
sh_stop	1	0	0	0	0	0	0	0	0	0	0	0	1	0
phout_start	0	0	0	1	0	0	0	0	0	0	0	0	0	0
phout_stop	1	0	0	0	0	0	0	0	0	0	0	0	1	0
load_counter_zyklen	0	0	0	0	1	1	1	1	0	0	0	0	0	0
inclk0_counter_zyklen	0	0	0	0	0	0	0	0	0	1	0	0	0	0
inclk1_counter_zyklen	0	0	0	0	0	0	0	0	1	0	0	0	0	0
load_counter_hdiv	0	0	0	0	1	1	1	1	0	1	0	0	0	0
go_counter_hdiv	0	0	0	0	0	0	0	0	1	0	0	0	0	0
load_counter_ldiv	0	0	0	0	1	1	1	1	1	0	0	0	0	0
go_counter_ldiv	0	0	0	0	0	0	0	0	0	1	0	0	0	0
cstart_reset	0	0	0	0	0	0	0	0	1	1	1	0	0	0
sread	0	0	0	0	0	0	1	0	0	0	0	0	0	0
swrite	0	0	0	0	0	1	0	0	0	0	0	0	0	0
sready	0	0	0	0	0	0	0	0	0	0	0	0	0	0
swork	0	0	0	0	0	0	0	1	1	0	0	0	0	1
snext	0	0	0	0	0	0	0	0	0	0	1	1	0	0
ffreg_out_en	0	0	0	0	0	0	0	1	1	1	0	1	0	1
ffreg_in_en	0	0	0	0	0	0	0	0	1	1	0	1	0	1
setsimclock	0	0	0	0	0	0	0	0	1	0	0	0	0	0
activate_reset_loc	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Tabelle 3.3.5.: Die Ausgaben des Clockcontrolautomaten in seinen Zuständen

3.4. Anwendungsmöglichkeiten

Der Haupteinsatzzweck dieses Hardware/Software-Interfaces ist die gekoppelte Simulation einer SystemC-Verhaltensbeschreibung und einer synthesefähigen VHDL-IP.

Eine weitere Anwendung liegt in der Simulation eines Prototypen auf einem FPGA unter Verwendung eines SystemC-Testbenches. Dies stellt den Haupteinsatzzweck ohne weitere SystemC-Module dar. Die einzigen Module sind dabei der Testbench, die Mainfunktion und das Koppelmodul des Interfaceblocks. Durch die Verwendung von SystemC können die Simulationsdaten des ausgeführten Designs in Tracefiles aufgezeichnet werden und erlauben so ein einfaches Hardwaredebugging.

Unter Benutzung eines FPGA, das dynamische Rekonfiguration erlaubt, können auch dynamisch rekonfigurierbare Designs emuliert werden. Diese Anwendung erlaubt den

Funktionsname	**Aufgabe**
SetWriteMode	Setzt cwrite-Bit in CU_{HW}
SetReadMode	Setzt cread-Bit in CU_{HW}
SetWaitMode	Setzt Kontrollregister auf 00h
SetCycleEnd	Setzt cnext-Bit in CU_{HW}
SetStartSim	Setzt cstart-Bit in CU_{HW}
SetDUTreset	Setzt Bit für asynchronen Reset der IP in CU_{HW}
SetResetIFB	Setzt creset-Bit in CU_{HW} und löst damit Reset des IFB aus
SetLDIVCounter	Setzt den Zählen LDIV in CU_{HW}
SetHDIVCounter	Setzt den Zähler HDIV in CU_{HW}
SetCycleCounter	Setzt den Zyklenzähler in CU_{HW}
GetStatus1	Liest erstes Statusbyte von CU_{HW}
GetStatus2	Liest zweites Statusbyte von CU_{HW}
GetLDIVCounter	Liest Wert des LDIV-Zähler
GetHDIVCounter	Liest Wert des HDIV-Zähler
GetZyklenCounter	Liest Wert des Zyklenzählers
condition_to_start_simulation	Prüft, ob ein Signal einen bestimmten Wert hat
init_Interface	initialisiert die parallele Schnittstelle
test_environment_id	prüft, ob sich auf dem FPGA der richtige IFB befindet

Tabelle 3.3.6.: Funktionen der CU_{SW}

Test dieser Designs und wiederum die Aufzeichnung von Testdaten in einem Tracefile oder durch den Nutzer unter Verwendung von C/C++ Operationen.

Weiterhin ist vorstellbar, dass mit Hilfe eines Simulationsinterfaceblocks rechenzeitintensive Teile der Designbeschreibung auf ein FPGA ausgelagert werden. Die benötigte Rechenleistung nimmt ab und eine Beschleunigung der Simulation kann erreicht werden.

4. Der Simulationsinterfaceblock-Generator

Dieses Kapitel stellt das Programm **Simulationsinterfaceblock-Generator** vor. Seine Entwicklung fand im Rahmen dieser Arbeit statt.
Zunächst werden die Gründe für die Implementierung des Programms beschrieben. Der sich anschließende Abschnitt beschreibt die Merkmale und die Möglichkeiten des Programms. Danach wird die Arbeitsweise vorgestellt. Am Ende dieses Kapitels wird auf die Benutzung des Programms eingegangen und Verbesserungsmöglichkeiten aufgezeigt.

4.1. Motivation

Der Simulationsinterfaceblock verbindet eine IP-Emulation mit einer SystemC-Simulation. Die Parameter des Interfaceblocks hängen immer von den Eingängen und Ausgängen der emulierten IP ab. Aus diesem Grund muss der Simulationsinterfaceblock für jede IP neu erstellt werden.

Ein Ziel der Kopplung von Emulation und Simulation durch einen Interfaceblock liegt darin, Zeit bei der Simulationsvorbereitung einzusparen. Die Erstellung eines Simulationsinterfaceblocks kann aber auch viel Zeit in Anspruch nehmen. Die manuelle Erstellung des Simulationsinterfaceblocks erhöht zudem die Wahrscheinlichkeit, dass sich Fehler im Design einschleichen.

Um die menschlichen Fehler einzugrenzen und die Zeit für die manuelle Erstellung einzusparen, soll hier im Folgenden eine automatische Lösung vorgestellt werden. Die Lösung besteht im Programm Simulationsinterfaceblock-Generator und wurde im Rahmen dieser Arbeit entwickelt.

4.2. Arbeitssweise

Dieser Abschnitt stellt die Arbeitsweise des Simulationsinterfaceblock-Generators vor. Zunächst erfolgt die Vorstellung der Eingaben für das Programm und die gewünschten Ausgaben. Anschließend wird die interne Datenstruktur, auf der das Programm arbeitet, erklärt, bevor der Ablauf des Programms beschrieben wird.

4.2.1. Eingaben und Ausgaben

Für die Entwicklung eines Programmes sind als erstes die Eingaben und Ausgaben zu betrachten (Abbildung 4.2.1). Als Eingaben stehen dem Simulationsinterfaceblock-Generator die VHDL-IP, die emuliert werden soll und Templates zur Verfügung. Zusätzlich können vom Benutzer noch einige Parameter während des Programmlaufs festgelegt werden.

Nach dem Programmlauf sollen sowohl der Softwareteil also auch der Hardwareteil des Simulationsinterfaceblocks zur Verfügung stehen. Der Softwareteil besteht aus C/C++-Quelldateien, die in ein SystemC-Design eingebunden werden. Der Hardwareteil beinhaltet als erste Ausgabe die VHDL-Quelldateien des Simulationsinterfaceblock und die daran angebundene IP. Aus diesen Quelldateien erfolgt, unter Nutzung von externen Tools zur Synthese, Platzierung und Trassierung, die Generierung der Konfigurationsdatei für das FPGA.

4.2.2. Die internen Datenstrukturen

Ein Programm benötigt zur Speicherung und Verarbeitung von Daten eine Datenstruktur. Die Hauptdatenstruktur im Simulationsinterfaceblock-Generator ist die `ENTITY_STRUCTUR`.

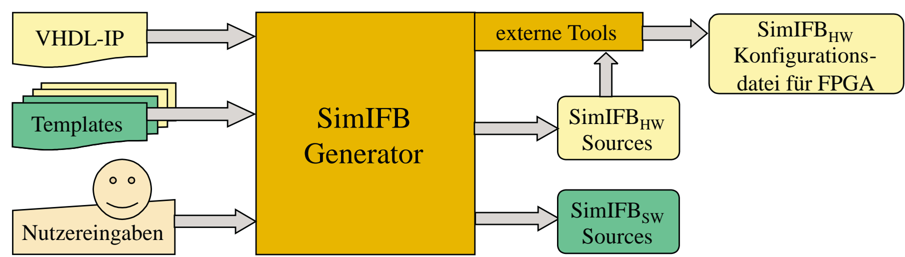


Abbildung 4.2.1.: Eingaben und Ausgaben des SimIFB-Generators

Sie nutzt die Datenstrukturen PORT_LIST und GENERIC_LIST. Zur Unterstützung bestimmter Operationen existieren des Weiteren die Datenstrukturen WORD_LIST und FILELIST. Die genaue Beschreibung der Datenstrukturen erfolgt in den sich anschließenden Abschnitten.

Die Datenstruktur ENTITY_STRUCTUR

Die grundlegende Datenstruktur im Simulationsinterfaceblock-Generator ist die ENTITY_STRUCTURE. Sie speichert alle Daten, die für die Erstellung des Simulationsinterfaceblocks notwendig sind. Die Daten bestehen aus Analyseergebnissen der VHDL-IP und darauf aufbauender Berechnungen, sowie aus Parametern, die vom Nutzer zur Laufzeit des Programms festgelegt werden. Außerdem existieren einige Funktionen, die eine einfache Arbeit mit der Datenstruktur erlauben.

Zunächst soll die Funktion der Variablen der Datenstruktur, die Tabelle 4.2.1 aufführt, vorgestellt werden. Die Variable *filename* speichert den Namen der Quelldatei, die den Toplevel der VHDL-IP darstellt. Den Namen der VHDL-Entity dieser Datei enthält *component_name* und die Position innerhalb der Datei *compname_position*.

Die Zeiger *generics* und *ports* verweisen auf die Datenstrukturen GENERIC_LIST bzw. PORT_LIST. Sie dienen zum Erfassen der Generics und Ports der Entity. Ihre genaue Funktion beschreiben Abschnitt 4.2.2 bzw. Abschnitt 4.2.2.

ENTITY_STRUCTUR	
Datentyp	**Variablenname**
char*	filename
char*	component_name
int	compname_position
GERNERIC_LIST*	generics
PORT_LIST*	ports
int	data_width_to_dut
int	hs_width_to_dut
int	data_width_from_dut
int	byte_to_dut
int	byte_from_dut
unsigned char	id
int	hdiv_counter
int	ldiv_counter
int	cycle_counter
int	epp_port_addr
int	fpga_prog_after_gen

Tabelle 4.2.1.: Aufbau der Datenstruktur ENTITY_STRUCTUR

Funktion	**Aufgabe**
make_entitystructur	Erzeugt ein neues Element vom Typ `ENTITY_STRUCTUR`
init_entitystructur	Initialisiert die Datenstruktur vom Typ `ENTITY_STRUCTUR`
reserve_wordspace_ename	Reserviert Speicherplatz für die Variable *component_name*
reserve_wordspace_fname	Reserviert Speicherplatz für die Variable *filename*
check_entity_for_clock	Durchsucht die Ports der Entity nach einem Taktsignal und legt eines an, falls die Suche fehlschlägt.

Tabelle 4.2.2.: Überblick zu den Funktionen zur Datenstruktur ENTITY_STRUCTUR

Die Variablen *data_width_to_dut* und *data_width_from_dut* geben die benötigte Anzahl Signale an, um Daten zur und von der IP zu transportieren. *Byte_to_dut* und *byte_from_dut* stellen die Werte auf ein volles Byte gerundet dar. Die Variable *hs_width_to_dut* gibt die Breite der Handshakesignale an, die der SH_{HW} zum Zugriff auf den Speicher im $PH_{HW\text{-}out}$ benötigt.

In *id* wird eine 8 Bit lange Identifikationsnummer gespeichert. Ein Zufallsgenerator erzeugt sie. Sie dient zur Identifikation des Hardwareteils des Simulationsinterfaceblocks durch den Softwareteil um sicherzustellen, dass die richtigen Teilstücke miteinander arbeiten.

Die Variablen *hdiv_counter*, *ldiv_counter* und *cycle_counter* speichern die Werte, auf die während der späteren Simulation die Zähler des Hardwareteils eingestellt werden. Die Funktion der Zähler wurde bereits im Abschnitt 3.3.7 erläutert.

Epp_port_addr enthält die Adresse, über die die parallele Schnittstelle des ausführenden PCs angesprochen wird. Der Wert in *fpga_prog_after_gen* gibt an, ob der Simulationsinterfaceblock-Generator das FPGA nach der Generierung des Simulationsinterfaceblocks sofort konfigurieren soll.

Zur Datenstruktur `ENTITY_STRUCTUR` gehören neben den Variablen auch einige Funktionen, die Tabelle 4.2.2 aufführt. Sie dienen zur einfacheren Arbeit mit der Datenstruktur.

Die Datenstruktur PORT_LIST

Zur Speicherung der Struktur der durch den Simulationsinterfaceblock-Generator analysierten VHDL-Entity der IP, dient die Datenstruktur `PORT_LIST`. Sie wird durch eine

PORT_LIST	
Datentyp	**Variablenname**
char*	port_name
int	port_position
unsigned char	port_direction
unsigned int	port_width
unsigned int	port_high
unsigned int	port_low
unsigned char	port_align
unsigned int	map_from
unsigned int	map_to
unsigned char	special
unsigned char	active
unsigned char	sign
struct port_list*	prev
struct port_list*	next

Tabelle 4.2.3.: Aufbau der Datenstruktur PORT_LIST

doppelt verkettete lineare Liste gebildet. Tabelle 4.2.3 stellt die Variablen eines Elementes der Liste dar. Die Verkettung der einzelnen Elemente realisieren die Zeiger *prev* und *next*.

Eine VHDL-Entity besteht aus Ports, die die Schnittstelle der jeweiligen Komponente definieren. Für jeden Port existiert in der Liste ein Element. Die Variable *port_name* gibt den Namen des Ports an. *Port_position* speichert die Position des Ports in der VHDL-Datei. Die Richtung des Ports, also ob es sich um einen Eingang, Ausgang, einen bidirektionalen Port oder einen Buffer handelt, gibt *port_direction* an.

Ein Port besitzt eine Breite *port_width*. Die Grenzen des Ports bestimmen *port_high* und *port_low*. Die Ausrichtung des Ports, dass heißt „to“ oder „downto“ falls er ein Vektor ist, speichert die Variable *port_align*.

Nach der Entityanalyse berechnet das Programm noch Werte zur Abbildung der Signale innerhalb des Interfaceblocks. Die Variablen *map_from* und *map_to* nehmen diese auf. Spezielle Attribute eines Ports, wie Takt- oder Resetsignal, legt die Variable *special* fest. In diesem Zusammenhang bestimmt *active* die aktive Taktflanke dieses Signals. Ob der Port im erstellten SystemC-Modul vorzeichenbehaftet ist oder kein Vorzeichen besitzt, beeinflusst die Varibale *sign*.

Neben den Variablen erleichtern einige Funktionen die Arbeit mit der Datenstruktur. Ihre Aufgaben stellt Tabelle 4.2.4 dar.

Funktion	**Aufgabe**
make_port	Erzeugt ein neues Element vom Typ `PORT_LIST`
init_portlist	Initialisiert die Datenstruktur
insert_port	Fügt ein Element in eine bestehende Liste ein
reserve_wordspace	Reserviert Speicherplatz für die Variable *port_name*
sort_portlist_width	Sortiert die Liste absteigend nach der Breite der Ports *port_width*
sort_portlist_direction	Sortiert die Liste nach der Richtung der Ports *port_direction*

Tabelle 4.2.4.: Überblick zu den Funktionen zur Datenstruktur PORT_LIST

`GENERIC_LIST`	
Datentyp	**Variablenname**
char*	generic_name
int	generic_value
struct generic_list*	prev
struct generic_list*	next

Tabelle 4.2.5.: Aufbau der Datenstruktur GENERIC_LIST

Die Datenstruktur GENERIC_LIST

Die Datenstruktur `GENERIC_LIST` wurde eingeführt, um generische Parameter in der VHDL-Entity zu unterstützen. Allerdings existiert bis jetzt nur die Definition der Datenstruktur. Tabelle 4.2.5 führt die enthaltenen Variablen auf. Eine doppelt verkettete lineare Liste realisert die Datenstruktur ähnlich wie die Liste der Ports (vgl. Abschnitt 4.2.2). Die Nutzung von VHDL-Generics ist im Simulationsinterfaceblock-Generator noch nicht implementiert. Aus diesem Grund wird an dieser Stelle auf eine ausführlichere Beschreibung verzichtet.

Die Datenstruktur WORD_LIST

Die Datenstruktur `WORD_LIST` hat die Aufgabe, das Parsen und Analysieren der VHDL-Entity der IP zu unterstützen. Dies geschieht, indem mit Hilfe der Datenstruktur die einzelnen syntaktischen Elemente der Entity wortweise gespeichert werden. Sie ist durch eine doppelt verkettete lineare Liste realisert, deren Elemente die Variablen aus Tabelle 4.2.6 enthalten.

Die Verkettung der Liste wird durch die Zeiger *prev* und *next* hergestellt. Einzelne Worte speichert die Variable *word*. Die Position des Wortes in der VHDL-Datei beinhaltet die Variable *position*.

Zur Datenstruktur gehören die Funktionen *make_word*, *insert_word* und *reserve_wordspace*, deren Aufgaben Tabelle 4.2.7 beschreibt.

WORD_LIST	
Datentyp	**Variablenname**
char*	word
int	position
struct word_list*	prev
struct word_list*	next

Tabelle 4.2.6.: Aufbau der Datenstruktur WORD_LIST

Funktion	Aufgabe
make_word	Erzeugt ein neues Element vom Typ `WORD_LIST`
insert_word	Fügt ein Element in eine bestehende Liste ein
reserve_wordspace	Reserviert Speicherplatz für die Variable *word*

Tabelle 4.2.7.: Überblick zu den Funktionen zur Datenstruktur WORD_LIST

Die Datenstruktur FILELIST

Die Aufgabe der Datenstruktur `FILELIST` besteht darin, eine Liste von Dateinamen zu speichern. Sie findet Verwendung beim Kopieren der IP und der darin deklarierten Komponenten in das neue Zielverzeichnis.

Die Datenstruktur ist eine einfach verkettete lineare Liste, bei der die Verkettung durch den Zeiger *next* erreicht wird. Außerdem kann sie in jedem Element der Liste einen Dateinamen (*filename*) und den Bearbeitungsstatus dieser Datei (*processed*) speichern. Tabelle 4.2.8 gibt den Aufbau der Datenstruktur wieder.

FILELIST	
Datentyp	**Variablenname**
char*	filename
int	processed
struct filelist*	next

Tabelle 4.2.8.: Aufbau der Datenstruktur FILELIST

Funktion	**Aufgabe**
filelist_new	Erzeugt ein neues Element vom Typ `FILELIST`
filelist_addfile	Fügt einen Dateinamen in die Liste ein, falls dieser noch nicht in der Liste vorhanden ist
filelist_free	Gibt den Speicherplatz für die Liste wieder frei

Tabelle 4.2.9.: Überblick zu den Funktionen zur Datenstruktur FILELIST

Die Funktionen zur Arbeit mit der Datenstruktur führt Tabelle 4.2.9 auf. Darin ist auch die Aufgabe der Funktionen dargestellt.

4.2.3. Der Programmablauf

Der erste Schritt, den der Simulationsinterfaceblock-Generator im Programmablauf durchführt, ist die Analyse der VHDL-IP. Bevor jedoch die Analyse durchgeführt werden kann, muss die Quelldatei aufbereitet werden. Der erste Arbeitsgang liest die Quelldatei der VHDL-IP in den Speicher. Anschließend entfernt die Funktion *vhdl_remove_comments* die Kommentare aus dem Quelltext. Die Formatierungszeichen, wie z.B. Zeilenumbrüche, Tabulatoren und mehrfache Leerzeichen, entfernt die Funktion *string_remove_multispace*. Um ein einfacheres Parsen des Quelltextes zu ermöglichen, wandelt die Funktion *string_tolowercase* den Text in Kleinbuchstaben um. Am Ende der Vorbereitung sucht *vhdl_entity_borders* die Grenzen der Entity im Text. Es wird hierbei davon ausgegangen, dass der Quelltext nur eine VHDL-Entity beschreibt oder die erste Entity im Text das Toplevel der IP darstellt. Ihre Position bestimmt der Anfang des Schlüsselwortes *entity* und das Ende die Position des Schlüsselwortes *end*. Anhand dieser Grenzen löst die Funktion *string_extract* die Entity aus dem Text heraus, indem sie den Bereich an den Beginn des Textes schreibt und den Rest mit Leerzeichen auffüllt.

Sind alle vorbereitenden Arbeiten durchgeführt, startet durch die Funktion *vhdl_parse_entity* die Analyse der Entity. Sie beginnt mit einem syntaktischen Trennen des Textes mit Hilfe der Funktion *vhdl_seperate_syntax*. Das bedeutet, dass syntaktische Elemente im Text durch ein Leerzeichen voneinander getrennt werden. Zu den syntaktischen Elementen zählen zum Beispiel die Schlüsselwörter *entity*, *port* und *is*, aber auch Zeichenketten für Namen, sowie Begrenzungszeichen, wie Semikolon und Klammern. Nach der Trennung kommt die Datenstruktur `WORD_LIST` zum Einsatz, in der die syntaktischen Elemente wortweise gespeichert werden.

Auf dieser Datenstruktur erfolgt das eigentliche Parsen der Entity. Die Syntax, auf die der Parser aufbaut, stellt das Syntaxdiagramm in Abbildung 4.2.2 dar. Das Syntaxdiagramm gibt nicht die vollständige Syntax für eine VHDL-Entity wieder. Es ist nur auf die in diesem Zusammenhang benötigten syntaktischen Konstrukte reduziert. Andere mögliche Konstrukte wie Attribute werden durch den Parser ignoriert. Auch die

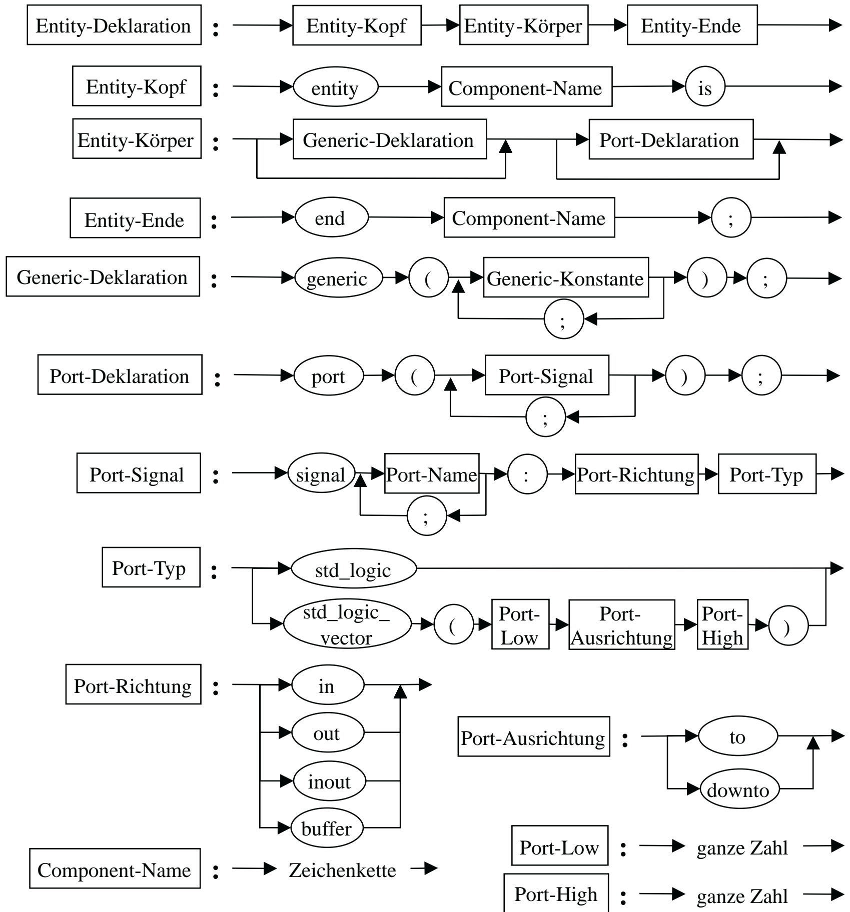


Abbildung 4.2.2.: Syntaxdiagramm einer VHDL-Entity nach dem der Parser des Simulationsinterfaceblock-Generators arbeitet

Auswertung von Generics vernachlässigt der Parser, da ihre Verwendung im Simulationsinterfaceblock-Generator bis jetzt noch nicht implementiert ist.

Der Parser stellt zunächst den Namen der Entity fest und speichert ihn in der Datenstruktur `ENTITY_STRUCTURE`. Anschließend baut er eine Liste der gefundenen Ports und ihrer Parameter mit der Datenstruktur `PORT_LIST` auf. Wurden alle Ports analysiert, prüft die Funktion *vhdl_complete_entity* die Liste der Ports auf Vollständigkeit. Fehlen Informationen zu einem Port, so ergänzt diese die Funktion. Die Namen der Ports können durch die Umwandlung des Quelltextes in Kleinbuchstaben jedoch verfälscht sein. Um zu gewährleisten, dass die Groß- und Kleinschreibung der Portnamen erhalten bleibt, ersetzt die Funktion *vhdl_replace_names* nach dem Parsevorgang die Namen in der Portliste durch ihre Originale aus der Quelldatei. Nach dem Parsevorgang steht nun eine Liste der Ports der IP bereit, in der alle aus der Entity gewonnenen Informationen gespeichert sind.

An dieser Stelle benötigt der Simulationsinterfaceblock-Generator noch ein paar Infor-

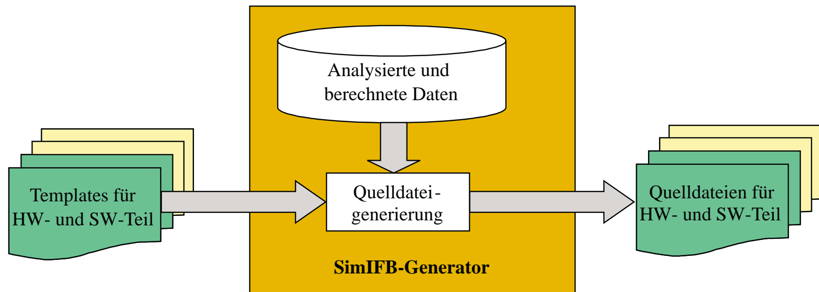


Abbildung 4.2.3.: Erzeugung der Quelldateien des Simulationsinterfaceblocks aus Templates

mationen vom Benutzer. Er muss das Taktsignal sowie das asynchrone Resetsignal und deren aktive Signalflanken festlegen. Diese Informationen dienen später dem Simulationsinterfaceblock zur Auslösung des Transportes der Simulationsdaten. Um die Kompatibilität mit dem SystemC-Design zu gewährleisten, in das der Simulationsinterfaceblock eingebunden werden soll, kann der Benutzer festlegen, ob Vektoren vorzeichenbehaftet sind oder nicht. Der Wert des Zyklenzählers, die Adresse der parallelen Schnittstelle und ob die FPGA durch den Simulationsinterfaceblock-Generator konfiguriert werden soll, sind weitere vom Benutzer einzustellende Parameter.

Nach dem Sammeln aller benötigten Informationen erfolgt eine Sortierung der Portliste. Zuerst sortiert die Funktion *sort_portlist_width* die Ports absteigend nach ihrer Breite. Im Anschluss sortiert *sort_portlist_direction* sie nach ihrer Richtung. Die Sortierung dient zur Vorbereitung für das Signalmapping. Das Signalmapping bildet die Ports mit Hilfe der Funktion *map_entity_signals* auf interne Signale des Simulationsinterfaceblocks ab. Außerdem wird geprüft, ob die Abbildung zulässig ist, dass heißt, dass nicht mehr Signale durch den Simulationsinterfaceblock transportiert werden müssen, als die HW/SW-Schnittstelle zulässt. Für solch einen Fall bricht das Programm mit einer Fehlermeldung ab.

War das Mapping erfolgreich, startet die Generierungsphase der Quelltexte. Die Generierung erfolgt aus Templates, was schematisch Abbildung 4.2.3 darstellt. Diese Templates enthalten bereits große Stücken an Quellcode, da viele Teile für den Simulationsinterfaceblock immer gleich sind. Codestücke, die speziell angepasst werden müssen, ersetzt im Template eine Marke. Diese Marken werden während der Generierung durch einen aus den vorhandenen Daten berechneten Wert oder ein anhand dieser Daten generiertes Codestück ersetzt. Anschließend erfolgt die Speicherung des fertigen Quelltextes im Zielverzeichnis.

Die Generierungsphase besteht aus drei Teilabschnitten. Der erste Schritt startet durch Aufruf der Funktion *generate_HWpart*. Er erzeugt alle notwendigen VHDL-Quelldateien des Simulationsinterfaceblocks . Außerdem instanziiert er den Simulationsinterfaceblock und die VHDL-IP unter einem Toplevel, um diese später gemeinsam zu synthetisieren. Während der Bearbeitung einer Quelldatei wird diese auch in eine Projektdatei

eingetragen, die alle Dateien zusammenfasst. Sie bildet die Informationsgrundlage für die spätere Synthese. Für die Synthetisierung müssen auch alle in der IP instanziierten Komponenten mit zu den VHDL-Quelldateien kopiert werden. Deshalb sucht die Funktion *generate_dut* alle instanziierten Komponenten und sammelt sie mit Hilfe der Datenstruktur `FILELIST`. Anschließend kopiert sie die Funktion zu den anderen VHDL-Quellen und fügt sie zur Projektdatei hinzu. Die Voraussetzung, dass diese Funktion ihren Zweck erfüllt, besteht darin, dass sich die instanziierten Komponenten im gleichen Verzeichnis wie der Quelltext der IP befinden und ihr Dateiname mit dem Namen ihrer Komponentendeklaration plus der Dateierweiterung .vhd übereinstimmt.

Nach der Erzeugung der VHDL-Quellen steht die Generierung von Skripten an. Diese Skripte bauen auf externen Programmen auf. Sie dienen der Synthese (xst), der Übersetzung (ngdbuild), dem Mapping (map), der Platzierung und Trassierung (par), der Bitstromgenerierung (bitgen), der FPGA-Konfigurierung (impact) und der Timinganalyse (trce) der erzeugten VHDL-Quellen. Die externen Programme sind in der Xilinx Entwicklungsumgebung ISE [Xilb] enthalten. Da in dieser Arbeit nur Xilinx FPGAs zur Verfügung standen, beschränkt sich der Simulationsinterfaceblock-Generator auf die Nutzung der Xilinx-Tools. Eine ausführliche Beschreibung der Tools ist in [Xila] und [Xild] zu finden. Eine Erweiterung des Programms ist aber durchaus denkbar.
Nach der Generierung der Skripte erfolgt ihre Ausführung durch die Funktion *execute_hwscripts*. Am Ende der Skriptsphase analysiert die Funktion *analyze_timing* das Logfile des Timinganalyseskriptes. Es sucht nach der größten Verzögerungszeit und berechnet daraus die Werte für die LDIV- und HDIV-Zähler. Dies dient dazu, dass die Emulation mit dem richtigen Takt ausgeführt wird, damit die Emulationsergebnisse auch solang abgegriffen werden, bis an den Ausgängen der IP ein korrektes Ergebnis anliegt. Die LDIV- und HDIV-Zähler benötigt die dritte Generierungsphase, um alle Quelldateien des Softwareteils erstellen zu können.

Die dritte Generierungsphase erzeugt den Softwareteil, also die C/C++ Quell- und Headerdateien des Simulationsinterfaceblocks. Sie startet durch Aufruf der Funktion *generate_SWpart*. Der erste Schritt generiert das SystemC-Modul, welches die Schnittstelle des Simulationsinterfaceblocks zur SystemC-Simulation bereitstellt. Dieses Modul trägt den Namen der Entity der VHDL-IP. Die Headerdatei bildet die Schnittstelle, die Quelldatei realisiert die Funktionalität des PH_{SW}. Beide Dateien werden komplett durch den Simulationsinterfaceblock-Generator erstellt und nicht aus Templates. Alle anderen Dateien des Softwareteils generiert das Programm wiederum aus Templates.

Während des Programmlaufs schreibt der Simulationsinterfaceblock-Generator Informationen für den Benutzer zu den einzelnen Operationen sowie aufgetretene Fehler in ein Logfile. Nach den Generierungsphasen ist das Programm abgeschlossen und die erzeugten Daten können verwendet werden. Mehr Informationen zur Verwendung der Daten bietet der Abschnitt 4.3.

4.3. Benutzung des Programms

Dieser Abschnitt erklärt die Benutzung des **Simulationsinterfaceblock-Generator**. Er gibt Hinweise zum Aufbau des Programms und zu den einzelnen Phasen, die während der Ausführung durchschritten werden. Außerdem enthält dieser Abschnitt Informationen zu den Nutzereingaben, die das Programm in den einzelnen Phasen erwartet. Am Ende wird ein Überblick über die vom Programm erzeugten Daten gegeben.

4.3.1. Programmstart und -aufbau

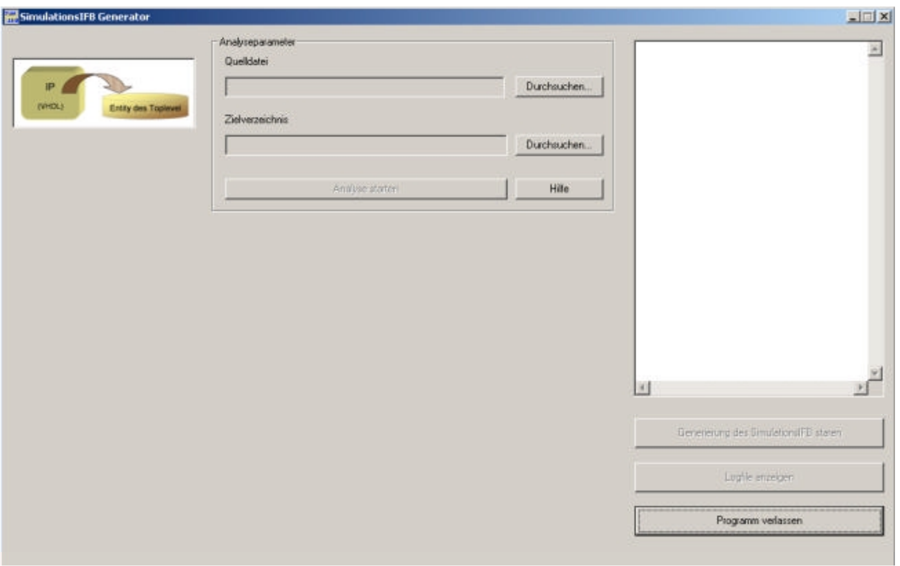


Abbildung 4.3.4.: Ansicht des Simulationsinterfaceblock-Generators nach dem Programmstart

Das Programm Simulationsinterfaceblock-Generator startet durch den Aufruf der Datei ***simifb_generator.exe***. Es erscheint ein Fenster auf dem Bildschirm, das Abbildung 4.3.4 darstellt. Die linke Seite des Fensters zeigt durch Grafiken die aktuelle Bearbeitungsphase an. In der Mitte des Fensters liegt der Bereich für die Eingaben des Benutzers. In den einzelnen Feldern des Nutzerbereichs existiert jeweils ein Hilfe-Button, um Informationen zu den entsprechenden Eingabefeldern zu erhalten. Der rechte Teil enthält ein Statusfeld, in dem Informationen vom Programm ausgegeben werden. Darunter befinden sich die Button zum Starten der Generierung, zum Anzeigen des Logfiles und zum Verlassen des Programms.

4.3.2. Die Analyse der Quelldatei

Die erste Eingabe, die der Nutzer durchzuführen hat, ist die Auswahl der VHDL-IP für die ein Simulationsinterfaceblock erstellt werden soll. Dies erfolgt im Bereich *Analyseparameter* unter dem Punkt Quelldatei. Mit einem Klick auf den oberen Durchsuchen. . . -Button öffnet sich ein Dialogfeld, in dem die Datei ausgewählt werden kann. Das Feld neben dem Button zeigt nach der Auswahl die Datei und ihr Verzeichnis an.

Die zweite Eingabe betrifft das Zielverzeichnis, in dem die generierten Dateien gespeichert werden sollen. Der entsprechende Dialog kann über den zweiten Durchsuchen. . . -Button erreicht werden. Den Pfad stellt das nebenstehende Textfeld nach der Auswahl dar.

Nach der Auswahl beider Parameter ist die Analysefunktion anwählbar (Abbildung 4.3.5). Durch Klicken des Button Analyse starten beginnt das Programm die VHDL-IP zu analysieren und ihre interne Datenstruktur aus der Entity der VHDL-IP aufzubauen.

4.3.3. Festlegung von Signalparametern

Es werden zwei weitere Felder nach Beendigung der Analyse sichtbar. Zum Einen das Feld *Auswahl des Taktsignals* und zum Anderen das Feld *Auswahl des asynchronen Resetsignals*. Abbildung 4.3.6 zeigt das Programm in diesem Programmabschnitt.

Die in diesem Abschnitt erschienenen Felder dienen zur Auswahl des Taktsignals und des asynchronen Resetsignals aus den Eingängen der VHDL-IP. Auf Grund dieser Informationen erzeugt der Simulationsinterfaceblock-Generator die Sensitivitätsliste des SystemC-Moduls. Es existiert jeweils für Takt- und Resetsignal die Option, dass diese

Abbildung 4.3.5.: Ansicht des Feldes Analyseparameter

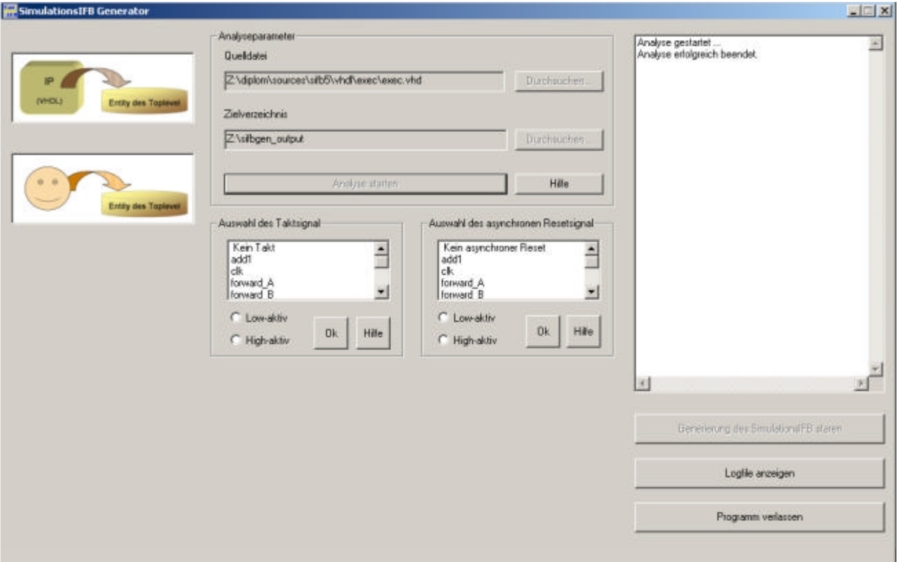


Abbildung 4.3.6.: Der Simulationsinterfaceblock-Generator nach der Analysephase

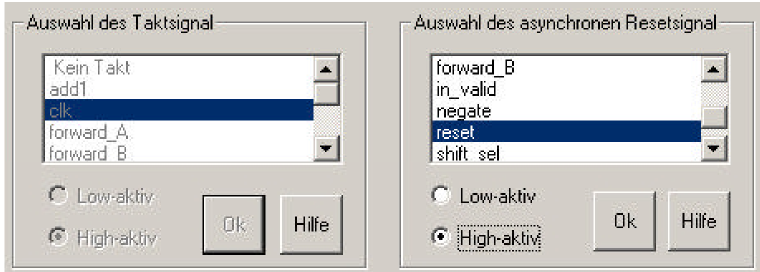


Abbildung 4.3.7.: Ansicht der Felder zur Einstellung der Signalparameter

Signale in der VHDL-IP nicht vorhanden sind. Zur Darstellung dieses Sachverhaltes kann der Listeneintrag „Kein Takt“ bzw. „Kein asynchroner Reset“ ausgewählt werden. Ist ein Signal ausgewählt, so muss der Benutzer zusätzlich dessen aktive Taktflanke festgelegen. Ein Klick auf den jeweiligen Ok-Button beendet die Eingabe der Signalparameter und lässt keine Änderung an den Einstellungen mehr zu. Abbildung 4.3.7 stellt eine detailierte Ansicht der Felder zur Auswahl des Takt- und des asynchronen Resetsignals dar. Fehler bei der Eingabe zeigt das Statusfeld an.

4.3.4. Weitere Parameter

Nun öffnet sich das letze Feld mit Benutzereingaben (Abbildung 4.3.8). In diesem Abschnitt können einige zusätzliche Parameter eingestellt werden. Auf der linken Seite des

Feldes können semantische Einstellungen zu den Vektoren vorgenommen werden. Alle Signale, die als Vektor in der Entity der VHDL-IP eingetragen sind, können näher spezifiziert werden. Der Nutzer kann die Auswahl treffen, ob der Vektor mit einem Vorzeichen behaftet sein soll oder nicht. Dies erleichtert die spätere Integration des generierten SystemC-Moduls in das SystemC-Design.

Rechts neben der Einstellung der Vektorsemantik befindet sich als erstes die Angabe der Taktzyklenanzahl, die die IP-Emulation in einem SystemC-Simulationstakt durchläuft. Voreingestellt ist der Wert 1. Der maximal mögliche Wert beträgt 65 536. Dieser Parameter kann dazu genutzt werden, um den Emulationstakt auf ein vielfaches des SystemC-Designtaktes einzustellen.

Im zweiten Parameterfeld ist die Auswahl der Adresse der parallelen Schnittstelle möglich. Zulässige Werte sind die Standardadressen 278h, 378h und 3BCh der Schnittstelle. Diese Adresse muss für den PC eingestellt werden, auf dem die SystemC-Simulation ablaufen wird. Es besteht jedoch die Möglichkeit, diesen Parameter nach der Generierung von Hand zu ändern. Die Erklärung folgt im Abschnitt 4.3.6.

Die Checkbox legt fest, ob die FPGA nach der Generierungsphase sofort konfiguriert werden soll, oder ob der Benutzer dies später selbst durchführen möchte. Um die spätere Konfigurierung der FPGA einfacher zu gestalten, wird das Skript zur Konfiguration *do_program.bat* immer erstellt und kann vom Benutzer dazu verwendet werden.

Eine Hilfestellung zu den Parametern kann über den Button [Hilfe] abgerufen werden. Abbildung 4.3.9 zeigt das Feld mit eingestellten Parametern.

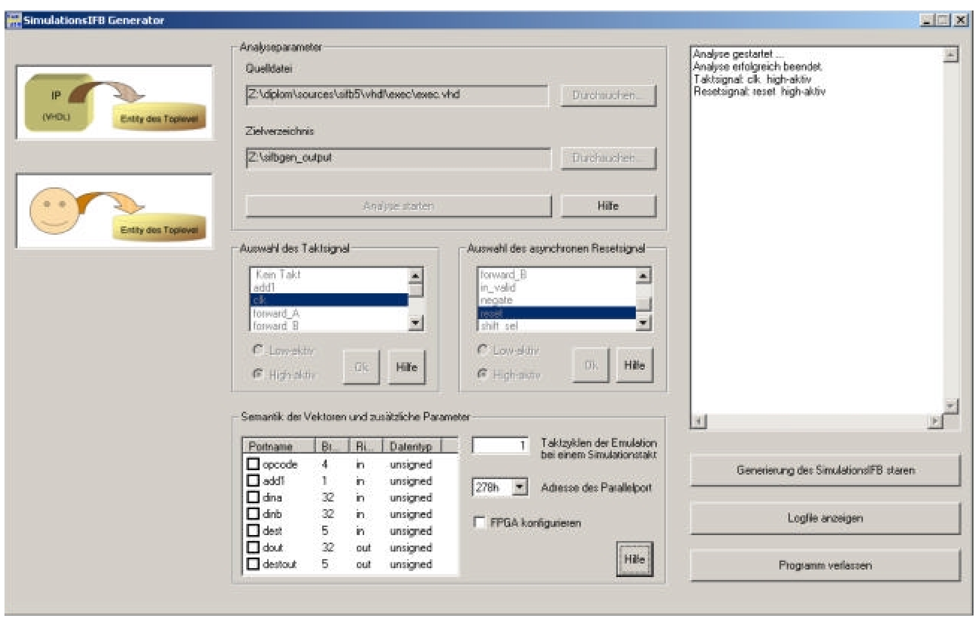


Abbildung 4.3.8.: Der Simulationsinterfaceblock-Generator nach der Einstellung der Signalparameter

Abbildung 4.3.9.: Detailierte Ansicht des Feldes zur Einstellung der Vektorsemantik und von zusätzlichen Parametern

4.3.5. Die Generierung

Der Button Generierung des SimulationsIFB starten löst die Erstellung des Simulationsinterfaceblocks aus. Als Erstes erfolgt die Generierung der Quelldateien des Hardwareteils (Abbildung 4.3.10). In diesem Schritt werden alle VHDL-Quelldateien erzeugt, die zur Synthese des Hardwareteils des Interfaceblocks und der Emulation der IP notwendig sind. Darunter fallen alle Dateien im Ausgabeverzeichnis *Hardware/Src* sowei die Projektdatei für die spätere Synthese.

Anschließend erstellt das Programm die Skripte für die Synthese, das Übersetzen, das Mapping, die Platzierung und Trassierung, die Generierung des Bitstroms und die Konfiguration des FPGAs. Diese Phase stellt Abbildung 4.3.11 dar. Die Ausführung der Skripte schließt sich an die Generierung an. Zur erfolgreichen Ausführung der Skripte ist es notwendig, dass die Entwicklungsumgebung Xilinx ISE auf dem Rechner installiert ist, der den Simulationsinterfaceblock-Generator ausführt. Im derzeitigen Zustand unterstützt der Simulationsinterfaceblock-Generator zur Ausführung der Emulation nur das Digilab 2E Development Board [Dig02] mit der darauf befindlichen Xilinx Spartan2E FPGA. Zur Unterstützung anderer FPGAs und Synthesewerkzeuge kann der Simulationsinterfaceblock-Generator später erweitert werden.

Die Generierung des Softwareteils des Simulationsinterfaceblocks schließt sich an die Skriptausführung an (Abbildung 4.3.12). Sie baut teilweise auf die Synthese des Hardwareteils, speziell auf die Timinganalyse, auf. In dieser Phase erstellt der Simulationsinterfaceblock-Generator das SystemC-Modul aus den analysierten Daten der Entity der VHDL-IP und den Eingaben des Nutzers. Das SystemC-Modul trägt den Namen

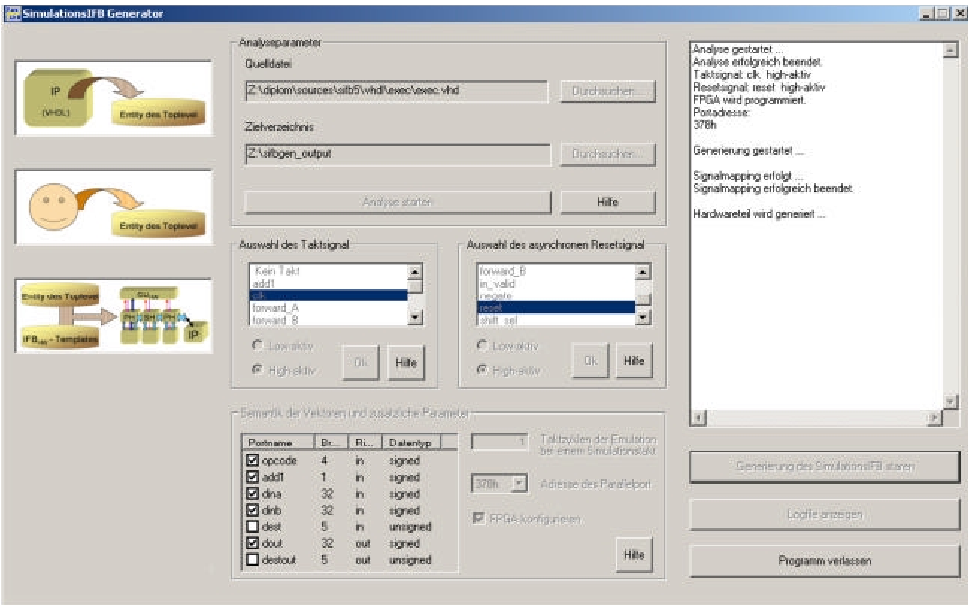


Abbildung 4.3.10.: Der Simulationsinterfaceblock-Generator bei der Generierung des Hardwareteils

der Entity der VHDL-IP. Zu diesem Modul gehören noch weitere Bibliotheken, die aus Templates erstellt werden. In den Templates trägt der Simulationsinterfaceblock-Generator nur einige Parameter, wie zum Beispiel die Größe der Simulationsdaten, ein. Nach dem fehlerfreien Abschluss der Generierung ist der Programmlauf beendet.

Das Statusfenster zeigt an, ob Fehler bei den einzelnen Schritten aufgetreten sind. Dies kann der Fall sein, wenn das Programm die Templates nicht fand oder die Ausführung der Skripte fehlschlug. Im letzeren Fall muss der Benutzer sicherstellen, dass das Verzeichnis mit den Xilinxtools in der Pfadumgebung des Betriebssystems vorhanden und die Xilinx Umgebungsvariable gesetzt ist. Tritt trotzdem ein Fehler bei der Synthese auf, so sollte die VHDL-Datei der IP auf mögliche syntaktische Fehler überprüft werden. Darüber, in welchem Abschnitt der Synthese der Fehler auftrat, geben die Logdateien des Syntheseprozesses Auskunft.

4.3.6. Überblick über die generierten Dateien

Dieser Abschnitt soll einen Überblick über die generierten Daten des Simulationsinterfaceblock-Generators geben. Die Verzeichnisstrukur stellt Abbildung 4.3.13 dar. Im Zielverzeichnis befinden sich zunächst die Unterverzeichnisse *Hardware* und *Software* sowie

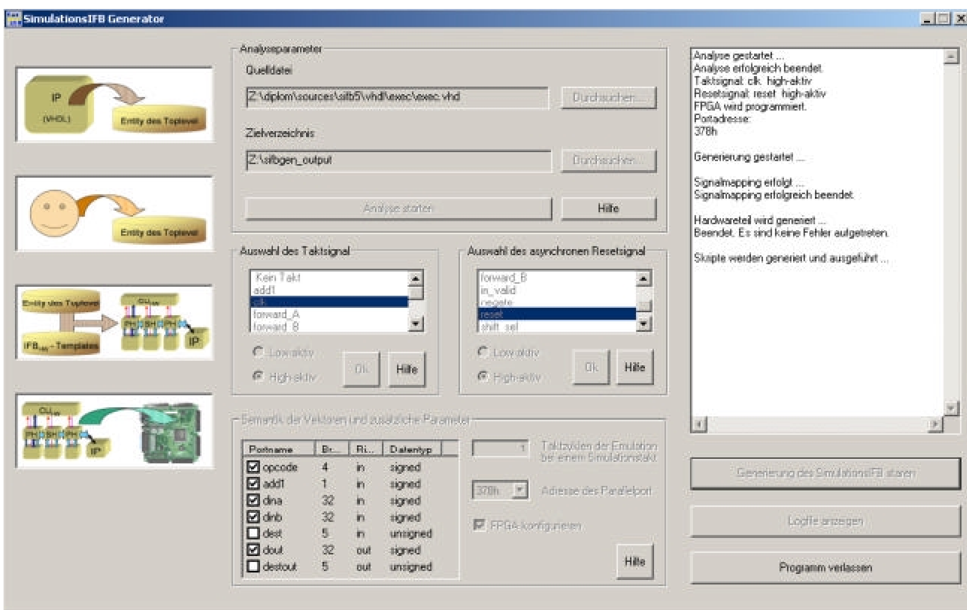


Abbildung 4.3.11.: Der Simulationsinterfaceblock-Generator bei der Generierung der Skripte

zwei Dateien. Die Datei *simifb-gen.log* ist das Logfile des Simulationsinterfaceblock-Generators und enthält die Statusausgaben und Fehlermeldungen, die während des Programmlaufs auftraten. Es kann auch über den Button Logfile anzeigen aus dem Programm heraus eingesehen werden. Die ausführbare Datei *exec_scripts.bat* ermöglicht es, den Implementierungsvorgang des Hardwareteils zu wiederholen, da sie alle ausgeführten Skripte aufruft.

Im Verzeichnis *Hardware* befinden sich alle Dateien, die zur Erstellung des Hardwareteils des Simulationsinterfaceblocks notwendig sind. Die Aufgaben der darin enthaltenen Dateien führt Tabelle 4.3.10 auf. Neben den Dateien verfügt das Verzeichnis über Unterverzeichnisse. Sie dienen der geordneten Speicherung der Implementierungsdaten des Hardwareteils des Simulationsinterfaceblocks.

Im Folgenden sind die Unterverzeichnisse und eine kurze Beschreibung der enthaltenen Daten aufgeführt:

Bit: Verzeichnis mit den Ausgaben der Bitfile-Generierung. Die Datei mit der Endung *bit* dient zur Konfiguration des FPGA.

Log: Dieses Verzeichnis enthält die Statusausgaben der ausgeführten Skripte. Dabei steht der erste Teil des Dateinamen für die ausgeführte Datei (z.B. xst.log für die Synthese mit xst.exe).

Map: Verzeichnis, dass die Daten aus dem Technologiemapping des Designs enthält.

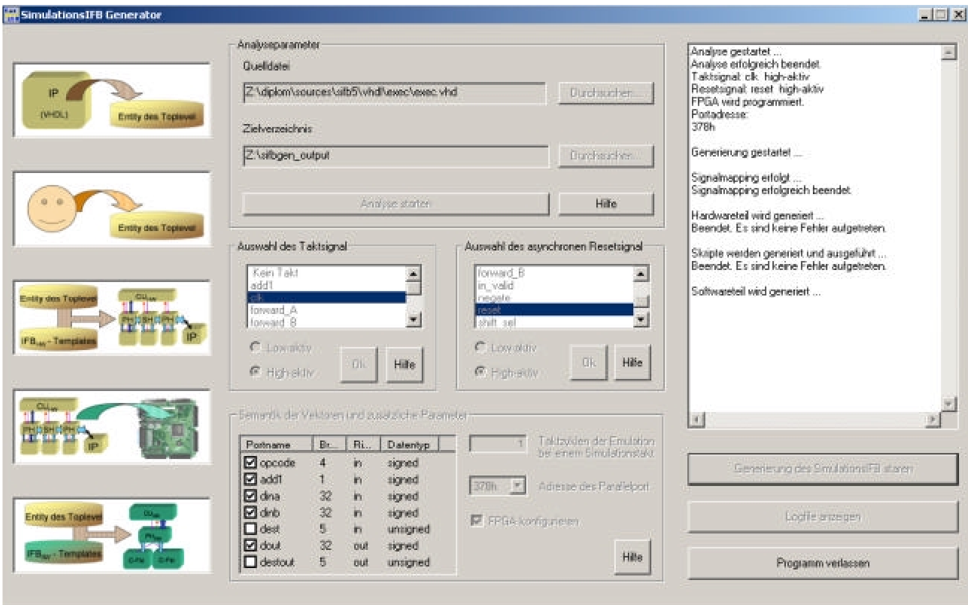


Abbildung 4.3.12.: Der Simulationsinterfaceblock-Generator bei der Generierung des Softwareteils

Ngdbuild: In diesem Verzeichnis befinden sich die Dateien, die durch die Übersetzung des Syntheseergebnisses erstellt wurden.

Par: Die Ergebnisse der Platzierung und Trassierung enthält dieses Verzeichnis.

Src: Die VHDL-Quelldateien des Hardwareteils, die vom Simulationsinterfaceblock-Generator erstellt wurden, speichert dieses Verzeichnis.

Trce: Dieses Verzeichnis sammelt die Ausgabedaten der Timinganalyse.

Xst: Die, durch die Synthese erstellten Daten, fasst das Verzeichnis xst zusammen.

Das Verzeichnis *Software* enthält alle Quelldateien, die für die Einbindung des Simulationsinterfaceblocks in eine SystemC-Simulation nötig sind. Das SystemC-Modul, welches die Schnittstelle zur SystemC-Simulation bildet, trägt den Namen der Entity der VHDL-IP und die Dateiendung .h. Darin ist die Schnittstelle definiert. Die Funktionalität stellt die gleichnamige Datei mit der Endung .cpp bereit. Alle anderen Dateien im Verzeichnis *Software* enthalten Funktionen, die von diesen Dateien genutzt werden. Dabei ist besonders die Datei *CUsw.h* zu erwähnen. Sie definiert alle wichtigen Parameter für den Softwareteil des Simulationsinterfaceblocks. Dazu zählen die Adresse der parallelen Schnittstelle, die Größe der Simulationsdaten, sowie die Adressen der Kontroll- und Statusregister des Hardwareteils des Simulationsinterfaceblocks .

Die Einbindung der Dateien des Softwareteils in ein bestehendes Design sollte aus dem generierten Verzeichnis erfolgen. Dazu muss beim Aufruf des C/C++-Compilers das

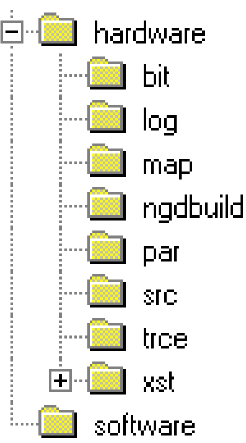


Abbildung 4.3.13.: Überblick über die Verzeichnisstruktur der generierten Daten

Dateiname	Aufgabe
sim_environment_hw.prj	Projektdatei für die Synthese
make.bat	Skript zur Ausführung aller Implementierungsskripte
do_xst.bat	Skript zum Starten der Synthese
xst_script	Hilfsdatei mit Parametern für die Synthese
do_ngdbuild.bat	Skript zum Ausführen des Übersetzungsvorgangs
do_map.bat	Skript für das Mapping des Desings
do_par.bat	Skript zum Ausführen der Platzierung und Trassierung
do_bitgen.bat	Skript zur Erstellung der Konfigurationsdatei für das FPGA
do_program.bat	Skript zum Starten der FPGA-Konfiguration durch das Programm Impact
program_script.cmd	Hilfsskript für Impact
do_trce.bat	Skript zum Durchführen der Timinganalyse

Tabelle 4.3.10.: Überblick zu den Dateien im Verzeichnis Hardware

Softwareverzeichnis als zusätzlicher Pfad für Includedateien angegeben werden. Für den Linker ist als Bibliothek die Datei *dlportio.lib* sowie das Softwareverzeichnis als weiteres Bibliotheksverzeichnis mit anzugeben. Das hat den Vorteil, dass die Ausgaben des Simulationsinterfaceblock-Generators in einem Verzeichnis gesammelt bleiben. Dies verringert die Gefahr, dass ein Hardwareteil auf das FPGA geladen wird, der nicht zum benutzen Softwareteil gehört und dadurch falsche Simulationsergebnisse erzielt werden.

4.4. Verbesserungsmöglichkeiten und Fazit

Um die Einsatzmöglichkeiten des Simulationsinterfaceblock-Generators zu verbessern, sind einige Erweiterungen im Programm denkbar. In der gegenwärtigen Version verarbeitet der Simulationsinterfaceblock-Generator nur Ports von VHDL-Entities mit den Richtungen in oder out. Zur Unterstützung einer größeren Anzahl von VHDL-IPs könnte das Programm für die Verarbeitung von bidirektionalen Ports (Richtung inout) erweitert werden. Mit der Analysemöglichkeit von VHDL-Generics wird das Konzept von generischen Portbreiten umgesetzt. Damit ist auch eine Unterstützung von parametrisierbaren IPs denkbar.

Die Menge der unterstützen IPs kann weiter steigen, indem Mechanismen in das Programm eingebaut werden, die andere Hardwarebeschreibungssprachen, wie zum Beispiel Verilog, analysieren können. Somit wird eine Adaptierung von IPs in verschiedenen Hardwarebeschreibungssprachen erreicht und das Programm gewinnt an Flexibilität.

Die vorliegenden Version des Simulationsinterfaceblock-Generators schränkt die hardwareseitige Implementierungsplattform noch auf das Digilab 2E Developmentboard ein. Um diese Restriktion aufzuheben, muss das Programm für die Unterstützung anderer FPGA-Typen und -Boards erweitert werden. Damit einher geht die Nutzung von unterschiedlichen Entwicklungsumgebungen und Tools zur Synthese und Implementierung. Da andere FPGA-Boards auch andere phyische Schnittstellen besitzen können, ist zudem eine Unterstützung anderer physischer HW/SW-Schnittstellen zur Überwindung der HW/SW-Grenze denkbar.

Der Simulationsinterfaceblock-Generator wurde entwickelt, um eine VHDL-IP mittels eines Simulationsinterfaceblocks an eine SystemC-Simulation automatisch zu koppeln. Das Programm nimmt dem Nutzer die manuelle Adaptierung der IP-Emlation an die SystemC-Verhaltenssimulation ab, indem es automatisch eine HW/SW-Schnittstelle zwischen beiden Tasks erzeugt. Dadurch sinkt die Zahl der möglichen Fehler, die bei einer manuellen Adaptierung auftreten können und der Zeitaufwand für die Adaptierung verringert sich stark. Das Programm implementiert die Grundfunktionalität der Adaptierung und zeigt, dass eine automatische Lösung praktikabel ist.

5. Demonstrator

Diese Kapitel beschäftigt sich mit dem Demonstrator zu den in dieser Arbeit gewonnenen Erkenntnissen. Im ersten Teil dieses Kapitels wird der allgemeine Nutzen eines Demonstrators herausgestellt. Der zweite Abschnitt beschäftigt sich mit dem Aufbau und der Funktionsweise des verwendeten Demonstrators. Anschließend erfolgt die Vorstellung der Ergebnisse, die mit seiner Hilfe gewonnen wurden. Es folgen einige Betrachtungen zur Simulationszeit des originalen SystemC-Designs und des modifizierten Designs mit der HW/SW-Schnittstelle. Das Ende des Kapitels fasst die Ergebnisse zusammen und zieht daraus Schlussfolgerungen.

5.1. Bedeutung eines Demonstrators

Ein Demonstrator dient dazu, die erarbeiteten theoretischen Konzepte im praktischen Einsatz zu erproben und zu validieren. Mit seiner Hilfe 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 besteht die Möglichkeit, dass es sich nicht oder nur schwer in die Praxis umsetzen lässt. Unter Umständen bildet dies eine Hürde für einen praktischen Einsatz des Modells. Auf der anderen Seite kann eine rein praktische Implementierung eventuell nicht allgemein modelliert werden und bleibt damit auf eine spezielle Anwendung beschränkt.

5.2. Aufbau und Funktionsweise

Zur Demonstration des Konzeptes der HW/SW-Schnittstelle und des, im Rahmen dieser Arbeit, erstellten Programms Simulationsinterfaceblock-Generator sollte ein Beispiel gewählt werden, welches die praktische Relevanz der Ergebnisse dieser Arbeit wiederspiegelt. Aus diesem Grund soll nach Möglichkeit auf ein bereits bestehendes Design zurückgegriffen werden und keine reine Selbstentwicklung Verwendung finden.

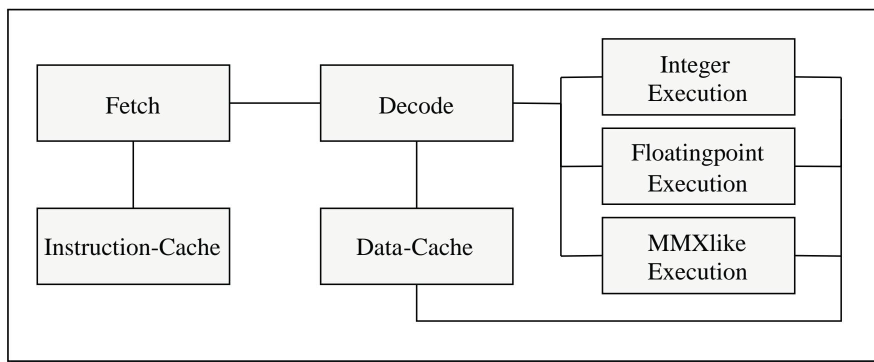


Abbildung 5.2.1.: Aufbau des Designs RISC-CPU

Als Grundlage des Demonstrators dient ein SystemC-Design aus den Beispielen zur SystemC-Version 2.0.1. Diese Version ist auf der SystemC-Website [Opea] frei verfügbar. Das Beispiel beschreibt ein einfaches Design einer RISC-CPU in SystemC. Es wurde von Synopsys® [Syn] entwickelt, um die Fähigkeiten von SystemC bei der HW/SW-Partitionierung zu veranschaulichen. Außerdem kann es als Instruction-set Simulator eingesetzt werden. Das Design findet sich nach dem Entpacken des heruntergeladenen SystemC-Paketes im Verzeichnis */systemc-2.0.1/examples/systemc/risc_cpu*. Es beinhaltet zusätzlich Programme für die CPU zum Testen der Funktionalität. Ein Assembler ist außerdem beigefügt, um selbst Programme zu entwickeln.

Den Aufbau der beschriebenen RISC-CPU stellt Abbildung 5.2.1 dar. Das Design der RISC-CPU setzt sich zusammen aus einem Instructioncache, einem Datencache, einer Fetcheinheit, die die Instruktionen aus dem Instructioncache liest, einer Decodiereinheit zum Umwandeln der Instruktionen in CPU-Operationen und jeweils einer Ausführungseinheit für Ganzzahlen, Fließkommazahlen und MMX-Berechnungen. Im Folgenden wird dieses SystemC-Design der RISC-CPU als originale RISC-CPU bezeichnet.

Im vorliegenden Szenario (Abbildung 5.2.2) soll die Ausführungseinheit für ganze Zahlen durch eine VHDL-IP ersetzt werden, die die gleiche Funktionalität bereitstellt. Die anderen Teile des SystemC-Designs, das heißt die Beschreibung des Prozessors und das verwendete Testprogramm bleiben unverändert. Dieses Design wird im Folgenden als modifizierte RISC-CPU bezeichnet. Die VHDL-IP dient als Eingabe für das Programm Simulationsinterfaceblock-Generator. Er generiert daraus die Hardware- und die Softwareseite des Simulationsinterfaceblocks und die Emulation für diese VHDL-IP. Die genaue Funktionsweise der Generierung beschreibt Kapitel 4. Die Implementierung von Hardware-Simulationsinterfaceblock und IP-Emulation dient zur Konfiguration eines Digilab 2E Development Boards. Der Softwareteil wird anstelle der Integer-Executionunit in das SystemC-Design eingebunden. Ein C++-Compiler erstellt daraus das Simulationsprogramm, welches ein PC, der über die parallele Schnittstelle mit dem FPGA-Board verbunden ist, ausführt.

Das Design der VHDL-IP wurde im Rahmen dieser Arbeit selbst entworfen. Es bildet

Opcode	Funktion	Verarbeitungsbreite
0000	-	-
0001	a + b + carry	32 Bit
0010	a - b - carry	32 Bit
0011	a + b	32 Bit
0100	a - b	32 Bit
0101	a * b	32 Bit
0110	a / b	16 Bit
0111	a nand b	32 Bit
1000	a and b	32 Bit
1001	a or b	32 Bit
1010	a xor b	32 Bit
1011	not a	32 Bit
1100	a << b	32 Bit
1101	a >> b	32 Bit
1110	a mod b	16 Bit
1111	-	-

Tabelle 5.2.1.: Funktionen der entwickelten Integer-Executionunit

alle Funktionen nach, die auch die SystemC-Beschreibung der Einheit enthält. Sie sind in Tabelle 5.2.1 aufgeführt. Nur der Dividierer unterliegt einer Einschränkung in der Verarbeitungsbreite. Dieser Kompromiss musste getroffen werden, da die verwendete Emulationsplattform, das Digilab 2E Developmentboard, nur über eine Xilinx Spartan2e FPGA (XC2S200E-PQ208) verfügt. Ihre Kapazität reicht nicht aus, um die Ausführungseinheit mit einem 32 Bit Paralleldividierer zu emulieren. Deshalb wurde die Verarbeitungsbreite der Division und der damit verbundenen Modulo-Brechnung auf 16 Bit gesenkt.

Diese Einschränkung stellt für den vorliegenden Demonstrator jedoch kein Hindernis dar, da im Testprogramm keine Division ausgeführt wird, die eine höhere Verarbeitungsbreite als 16 Bit beansprucht. Außerdem dient der Demonstrator dem Zweck, die Einsatzfähigkeit und korrekte Funktion des Konzeptes des Simulationsinterfaceblocks und des Simulationsinterfaceblock-Generators nachzuweisen und nicht die der VHDL-IP. Im Fall des Simulationsinterfaceblocks bedeutet es den korrekten Austausch der Daten über die HW/SW-Schnittstelle und die erfolgreiche Kopplung der SystemC-Simulation mit der IP-Emulation. Für den Simulationsinterfaceblock-Generator heißt korrekte Funktion, dass der Simulationsinterfaceblock in Abhängigkeit von der VHDL-IP so erstellt wird, dass er seine Aufgabe erfolgreich erfüllen kann.

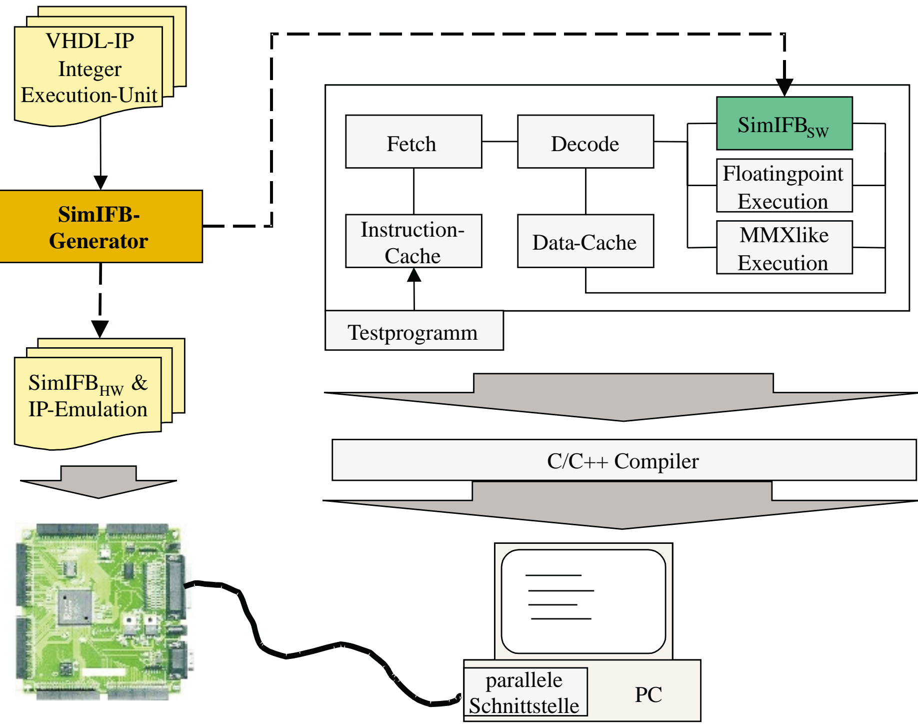


Abbildung 5.2.2.: Aufbau des Demonstrators

5.3. Nachweis der Funktion

Um den Nachweis zu erbringen, dass der Simulationsinterfaceblock seine Aufgabe erfüllt, sollen die Simulationsergebnisse der originalen und der modifizierten RISC-CPU miteinander verglichen werden. Dazu ist zunächst die Erstellung der Simulation der originalen RISC-CPU erforderlich. Dies geschieht einfach durch das Kompilieren und Linken der einzelnen SystemC-Module. Als Übersetzungs- und Linkprogramm kommt Microsoft Visual C++ 6.0 zum Einsatz. Hinweise zur Nutzung von SystemC in dieser Entwicklungsumgebung beschreibt Anhang B.

Die Erstellung der modifizierten RISC-CPU läuft nach Abbildung 5.2.2 ab. Zunächst wird das Programm Simulationsinterfaceblock-Generator gestartet. Ihm dient als Eingabe die VHDL-IP der Integer-Executionunit. Das Programm analysiert sie. Anschließend erfolgt die Festlegung der Signalparameter. Das Taktsignal *clk* und das asynchrone Resetsignal *rst* sind dabei high-aktiv. Die Datentypen der Vektoren werden entsprechend den Datentypen der originalen Ausführungseinheit (Abbildung 5.3.3) festgelegt. Abbildung 5.3.4 zeigt die Einstellungen im Programm. Die Generierung des Simulationsinterfaceblocks für die VHDL-IP der Integer-Executionunit stellt den letzen Schritt der Programmausführung dar.

```
sc_in<bool>          reset;
sc_in<bool>          in_valid;
sc_in<int>           opcode;
sc_in<bool>          negate;
sc_in<int>           add1;
sc_in<bool>          shift_sel;
sc_in<signed int>    dina;
sc_in<signed int>    dinb;
sc_in<bool>          forward_A;
sc_in<bool>          forward_B;
sc_in<unsigned>      dest;
sc_out<bool>         C;
sc_out<bool>         V;
sc_out<bool>         Z;
sc_out<signed int>   dout;
sc_out<bool>         out_valid;
sc_out<unsigned>     destout;
sc_in_clk            CLK;
```

Abbildung 5.3.3.: Schnittstellendefinition der originalen Integer-Executionunit. Der Datentyp bool gibt einfache Signale an. Int, signed int und unsigned geben Vektoren an.

Der Hardwareteil des Simulationsinterfaceblocks und die Implementierung der VHDL-IP befinden sich nun auf dem FPGA-Board. Sie sind bereit für die Ausführung. Um die Simulation der originalen und der modifizierten RISC-CPU besser vergleichen zu können, wird im Softwareteil des Simulationsinterfaceblocks Quellcode, zur Ausgabe der Simulationsdaten auf dem Bildschirm, manuell hinzugefügt. Dies erfolgt in der generierten Datei *exec.cpp*. Als Quelle dient hier die Datei *exec.cpp* der originalen RISC-CPU. Die anderen generierten Dateien bleiben unverändert.

Aus dem Design der originalen RISC-CPU wird die Integer-Executionunit entfernt und durch das generierte Modul ersetzt. Zur Übersetzung des Gesamtdesigns muss noch die Datei *dlportio.lib* zu den verwendeten Bibliotheken hinzugefügt werden. Außerdem ist als zusätzlicher Includepfad das Verzeichnis mit den generierten Quelldateien einzutragen. Nach der Durchführung dieser Schritte erfolgt das Übersetzen und Linken der modifizierten RISC-CPU. Auch hierbei kommt Microsoft Visual C++ 6.0 zum Einsatz.

Die Ausführung der Simulationen erzeugt deren Simulationsergebnisse, die auf dem Bildschirm ausgegeben werden. Die Ergebnisse sind in Anhang **??** auszugsweise gegenübergestellt. Ihre Gegenüberstellung zeigt, dass die Simulation der originalen und die der modifizierten RISC-CPU zu gleichen Simulationszeiten die gleichen Eingabe- und Ausgabedaten besitzen. Damit ist nachgewiesen, dass das Verhalten der beiden Designs übereinstimmt.

Nachdem sichergestellt ist, dass beide Designs über das gleiche Verhalten verfügen, soll

	Portname	Br...	Ri...	Datentyp
☑	opcode	4	in	signed
☑	add1	1	in	signed
☑	dina	32	in	signed
☑	dinb	32	in	signed
☐	dest	5	in	unsigned
☑	dout	32	out	signed
☐	destout	5	out	unsigned

Abbildung 5.3.4.: Festlegung der Datentypen im Simulationsinterfaceblock-Generator

der Nachweis erfolgen, dass der Simulationsinterfaceblock seine Aufgaben korrekt erfüllt. Seine Aufgaben lauten nach Abschnitt 3.1.1 wie folgt:

1. zwei Komponenten kommunizieren lassen, ohne dass an den Komponenten Änderungen durchgeführt werden müssen.
2. den bidirektionalen Datentransport zwischen den Komponenten sicherstellen.
3. die HW/SW-Grenze intern überwinden.
4. die Möglichkeit bieten, Daten intern zu transformieren.
5. die Integration einer Steuerung erlauben.

Am Demonstrator wurde gezeigt, dass die Simulationsdaten erfolgreich zwischen der SystemC-Simulation und der IP-Emulation in beide Richtungen ausgetauscht werden. Das zeigt, dass der Datentransport bidirektional über die HW/SW-Grenze möglich ist. Somit sind die Aufgaben 2 und 3 erfüllt. Da die Simulationsdaten für den Transport durch den Simulationsinterfaceblock intern umgewandelt werden und der Transport erfolgreich verläuft, ist auch die korrekte Erfüllung der 4. Aufgabe nachgewiesen. Die interne Steuerung, die Aufgabe 5 als Inhalt hat, arbeitet nach der Spezifikation, da sie den Datenaustausch steuert und dieser erfolgreich verläuft. Zum Nachweis der 1. Aufgabe müssen die kommunizierenden Tasks näher betrachtet werden. Das SystemC-Design bliebt, bis auf die Ersetzung der Integer-Executionunit, unverändert. Auch an der VHDL-Beschreibung der IP erfolgten keine Veränderungen. Da beide Komponenten durch den Simulationsinterfaceblock erfolgreich miteinander kommunizieren, gilt auch die 1. Aufgabe als erfüllt.

Den Simulationsinterfaceblock erzeugte der Simulationsinterfaceblock-Generator. Der Nachweis, dass der Simulationsinterfaceblock die ihm gestellten Aufgaben erfolgreich verrichtet, zeigt somit, dass er korrekt generiert wurde. Die Aufgabe, die die automatische Adaptierung der IP-Emulation auf dem FPGA mit der SystemC-Verhaltenssimulation durch einen Simulationsinterfaceblock beinhaltet, erfüllt der Simulationsinterfaceblock-Generator somit korrekt.

5.4. Betrachtungen zur Simulationszeit

Bei der Simulation spielt die benötigte Zeit eine große Rolle. Deshalb soll an dieser Stelle die Simulationszeit der originalen und der modifizierten RISC-CPU betrachtet werden.

Zu erwarten ist, dass die Zeit, welche die Simulation der modifizierten RISC-CPU beansprucht, größer ist, als die Simulationszeit der originalen RISC-CPU. Die Vermutung begründet sich darauf, dass bei der Simulation der modifizierten RISC-CPU zusätzlich Zeit für die Kommunikation zwischen SystemC-Simulation und IP-Emulation, sowie für die interne Kommunikation des Simulationsinterfaceblocks aufgewendet werden muss.

Daraus folgt, dass die Geschwindigkeit der physischen HW/SW-Schnittstelle einen wesentlichen Faktor für die Simulationszeit darstellt. Die Untersuchung ihrer Geschwindigkeit stellt sich deshalb als erste Aufgabe (Abschnitt 5.4.1). Anschließend wird die Simulationszeit, unter Berücksichtigung der gewonnenen Erkenntnisse zur HW/SW-Schnittstelle, betrachtet.

5.4.1. Geschwindigkeit der parallelen Schnittstelle

Für die Messung der Übertragungsgeschwindigkeit der parallelen Schnittstelle mit dem EPP-Protokoll wurde im Rahmen dieser Arbeit eine Messeinrichtung entwickelt. Sie setzt sich aus dem Softwareprogramm **Speedtest_sw** und dem VHDL-Design für das Digilab 2E Developmentboard **Speedtest_hw** zusammen. Beide Teile sind auf der beiliegenden CD-ROM zu finden (siehe Anhang **??**).

Das Programm Speedtest_sw hat die Aufgabe, fortlaufend Daten über die parallele Schnittstelle zu schreiben oder zu lesen. Für den Programmaufruf müssen zwei Parameter angegeben werden. Zu Einem der Arbeitsmodus und zum Anderen die Adresse der parallelen Schnittstelle im PC. Das Programm besitzt sechs verschiedene Modi, die jeweils eine andere Form der Datenübertragung durch das EPP-Protokoll ausführen. Ihre Funktionen sind in Tabelle 5.4.2 aufgeführt. Die Modi 5 und 6 realisieren ein abwechselndes Schreiben eines Adressbytes und Schreiben bzw. Lesen eines Datenbytes. Diese Formen der Datenübertragung sind für den Simulationsinterfaceblock besonders wichtig, da bei ihm jedem Datenbyte auch eine Adresse zugeordnet ist.

Den Aufbau des Hardwareteils zeigt Abbildung 5.4.5. Zur Messung der Übertragungsgeschwindigkeit des EPP-Protokolls dienen ein 32 Bit-Zähler und eine Stopuhr. Beide

Modus	Funktion
m1	Datenbyte schreiben
m2	Adressbyte schreiben
m3	Datenbyte lesen
m4	Adressbyte lesen
m5	Adressbyte schreiben, Datenbyte schreiben
m6	Adressbyte schreiben, Datenbyte lesen

Tabelle 5.4.2.: Die Modi der Software zur Messung der Übertragungsgeschwindigkeit der parallelen Schnittstelle im EPP-Modus

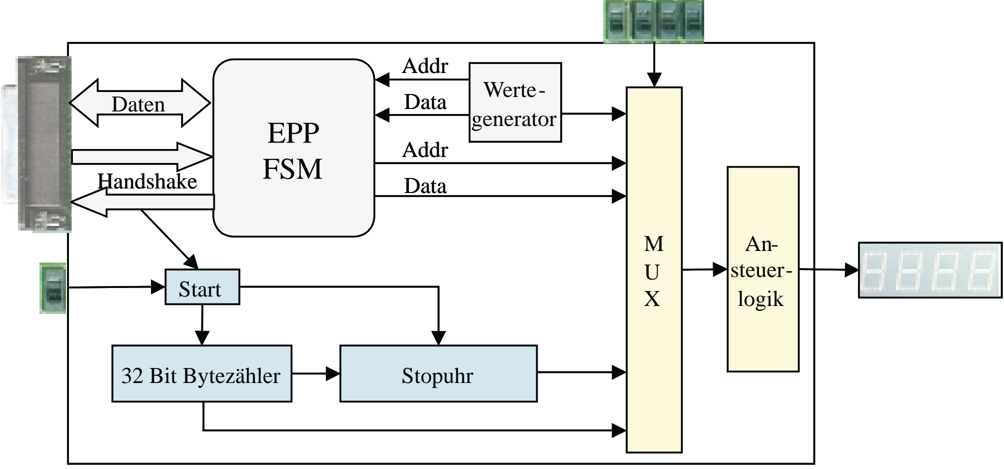


Abbildung 5.4.5.: Aufbau der Hardware zur Geschwindigkeitsmessung

werden über ein gemeinsames Startsignal aktiviert. Der Bytezähler reagiert auf das Signal *nWait* des EPP-Protokolls. In jedem Übertragungszyklus ändert das Signal zweimal seinen Wert, einmal von high zu low und einmal umgekehrt. In jedem Übertragunszyklus wird dabei genau ein Byte übertragen. Die Stopuhr beginnt die Zeitmessung, sobald der Bytezähler einen Wert ungleich Null annimmt. Sie kann bis zu 1 000 Sekunden zählen. Ihre Genauigkeit hängt von der eingesetzten Taktfrequenz ab. Im vorliegenden Fall liegt eine Taktung mit 50 Mhz vor. Daraus ergibt sich eine maximale Genauigkeit von 20 ns.

Zur Ausgabe der Daten dient eine 7-Segmentanzeige. Die Auswahl der Daten übernimmt ein Multiplexer. Es können die ausgehenden sowie die eingehenden Adressen und Daten angezeigt werden. Jedoch fällt ihnen für die Messung keine Bedeutung zu. Die wichtigen Daten bilden die Anzahl der übertragenen Bytes und die dafür benötigte Zeit. Auch sie können auf der Anzeige ausgegeben werden. Das Ausgabeformat ist hexadezimal.

Die Messungen der Geschwindigkeit erfolgen für die Modi 5 und 6. Modus 5 schreibt abwechselnd ein Adressbyte und ein Datenbyte auf die parallele Schnittstelle. Er nutzt

Modus	übertragene Byte	s	ms	µs	ns	Byte/s	Durchschnittsgeschwindigkeit
m5	18 860 504	30	543	789	500	617 460	618 404 Byte/s
	18 892 543	30	530	303	600	618 813	
	18 976 578	30	660	073	340	618 940	
m6	19 450 840	30	323	718	040	641 440	641 881 Byte/s
	19 517 029	30	355	897	920	642 940	
	19 529 100	30	454	054	240	641 264	

Tabelle 5.4.3.: Ermittelte Geschwindigkeiten der parallelen Schnittstelle

dazu die Funktion *Write_EPP*, die auch für die Datenübertragung im Simulationsinterfaceblock zum Einsatz kommt. Modus 6 schreibt ein Adressbyte und liest ein Datenbyte. Er verwendet die Funktion *Read_EPP*. Tabelle 5.4.3 stellt die Ergebnisse der Messungen dar.

Die Übertragungsgeschwindigkeit, die für die parallelen Schnittstelle mit dem EPP-Protokoll angegeben ist, beträgt nach [Pea04] zwischen 500 kByte/s und 2 MByte/s. Dies ergibt eine Zeit von 2 000 ns bis 500 ns für die Übertragung eines Bytes. In dieser Messung lag der Wert für reine Schreiboperationen bei durchschnittlich 618 404 Byte/s (1617 ns/Byte). Bei gemischten Schreib-Lese-Operationen erreichte die Schnittstelle eine Übertragungsgeschwindigkeit von durchschnittlich 641 881 Byte/s (1558 ns/Byte). Die Werte siedeln sich zwar im unteren Bereich der theoretisch angegebenen Geschwindigkeit an, liegen aber im spezifizierten Bereich.

5.4.2. Vergleich der Simulationszeiten

Um eine vergleichende Messung durchzuführen, müssen die gleichen Ausgangsbedingungen für beide Simulationen geschaffen werden. Sie werden auf dem gleichen PC (Pentium4 mit 2,4 Ghz, 512 MB RAM, Betriebssystem Windows XP Professional) ausgeführt. Für beide Simulationen kommt der gleiche Testbench zum Einsatz. Außerdem wird in den Modulen *exec.cpp* der Code zur Bildschirmausgabe auskommentiert. Das generierte Modul der modifizierten RISC-CPU wird in jedem Simulationsschritt einmal vollständig abgearbeitet. Die Ausführung des Originalmodul hingegen findet Abschnittsweise mit Hilfe von *wait()*-Anweisungen statt. Aus diesem Grund gibt das generierte Modul mehr Daten auf dem Bildschirm aus. Da Ausgabeoperationen auf der Standardausgabe aber viel Zeit beanspruchen, können sie die Messergebnisse stark verfälschen.

Die Zeitmessung für beide Simulationen erfolgt durch das Programm *timedrun.exe*, das im Rahmen dieser Arbeit erstellt wurde. Es führt zwei Skripte aus. Als erstes wird das Skript *copy_simulation.bat* ausgeführt, welches alle für die Simulation benötigten Dateien in das aktuelle Verzeichnis kopiert. Für die Ausführung des zweiten Skriptes *run.bat*

Simulation	Zeit	durchschnittliche Zeit
originale RISC-CPU	0,313 s 0,344 s 0,328 s	0,328 s
modifizierte RISC-CPU	0,391 s 0,359 s 0,406 s	0,385 s

Tabelle 5.4.4.: Gemessene Simulationszeiten der originalen und der modifizierten RISC-CPU. Es wurden je 3 Messungen durchgeführt und der Mittelwert daraus gebildet.

wird die Zeit gestoppt. Dieses Skript dient dazu, die Simulation aufzurufen und die Ausgaben in eine Datei umzuleiten. Die benötigte Zeit wird der Datei *time_taken_by_run.log* als letzter Eintrag angefügt.

Die Ergebnisse der Messung stellt Tabelle 5.4.4 dar. Die Simulation der originalen RISC-CPU benötigte eine durchschnittliche Zeit von 328 Millisekunden. Die Simulationszeit der modifizierten RISC-CPU liegt mit durchschnittlich 385 Millisekunden etwas darüber. Nun stellt sich die Frage, wodurch die um 57 Millisekunden größere Simulationszeit zu erklären ist.

Einen Unterschied zwischen der originalen und der modifizierten RISC-CPU stellt die Kommunikation über die HW/SW-Schnittstelle dar. Zunächst soll das Kommunikationsaufkommen und die dafür benötigte Zeit untersucht werden. Tabelle 5.4.5 enthält die Befehle, die in einem Simulationsschritt abgearbeitet werden. Befehle, die in Schleifen ausgeführt werden, sind mit Sternen gekennzeichnet. Die Anzahl der Schleifendurchläufe soll zunächst betrachtet werden, bevor wieder auf die Kommunikationszeit eingegangen wird.

Die erste Schleife dient zum sicheren Starten der IP-Emulation und besteht aus den Funktionen *SetStartSim* und *GetStatus1*. Sie wird in jedem Fall einmal durchlaufen. Im Normalfall endet die Schleife sofort nach dem ersten Durchlauf. Dies folgt aus der Übertragungszeit, die für ein Byte notwendig ist. Nach Abschnitt 5.4.1 beträgt die theoretisch kürzeste Zeit zur Übertragung eines Bytes über die parallele Schnittstelle 500 ns. Der Simulationsinterfaceblock ist mit 50 Mhz getaktet und besitzt demnach eine Taktperiode von 20 ns. Bevor die Funktion *GetStatus1* das Statusbyte liest, schreibt sie eine Adresse. Das heißt vom Ende der Funktion *SetStartSim* bis zum Lesen des Statusbyte vergehen mindestens 500 ns. Der Automat hat mindestens $500 : 20 = 25$ Taktperioden Zeit, um den Zustandswechsel auszuführen. Betrachtete man den Aufbau des Automaten in Abbildung 3.3.23 auf Seite 249 benötigt er einen Takt für den Zustandswechsel. Hinzu kommen jeweils ein Takt für die Wertübernahme durch die Kontrollregister und Statusregister. Dies zeigt, dass die Schleife unter normalen Bedingungen nur einmal Durchlaufen wird.

Funktion	**Kategorie (write/read)**	**Anzahl Bytes (write/read)**
GetStatus1	read	1/3
SetWriteMode	write	2/0
SetLDIVCounter	write	2/0
SetHDIVCounter	write	2/0
SetCycleCounter	write	4/0
WriteSimData	write	20/0
SetWaitMode	write	2/0
SetStartSim*	write	2/0
GetStatus1*	read	1/3
GetStatus1**	read	1/3
SetCycleEnd	write	2/0
SetReadMode	write	2/0
ReadSimData	read	6/18
SetWaitMode	write	2/0

Tabelle 5.4.5.: Das Datenaufkommen des generierten Moduls. Die Kategorie gibt an, ob es sich um eine Schreib- oder einen Lesefunktion handelt. Sterne hinter Funktionen bedeuten, dass sie in einer Schleife ausgeführt werden.

Zur Betrachtung der zweiten Schleife muss die Ausführungszeit eines IP-Emulationstaktes mit herangezogen werden. Die Frequenz der IP-Emulation berechnet sich nach Formel A.14 in Anhang A. Sie lautet

$$f_{IP} = \frac{f_{IFB}}{HDIV + LDIV + 2}$$

Die berechneten Werte des Simulationsinterfaceblock-Generators für HDIV und LDIV eingesetzt, ergibt sich die Frequenz der IP-Emulation von

$$f_{IP} = \frac{50\,\mathrm{Mhz}}{6 + 6 + 2} = 3,571\,\mathrm{Mhz}.$$

Damit beträgt die Zeit für eine Taktperiode 280 ns. Die Emulation startet sofort nach dem erfolgreichen Setzen des Startsignals durch *SetStartSim*. Anschließend wird die Funktion *GetStatus1* ausgeführt, die das Starten überprüft. Sie benötigt einen Schreibvorgang und drei Lesevorgänge. Drei Lesevorgänge wurden aus Gründen der Fehlerkorrektur eingeführt, da bei der Schnittstelle vereinzelt Lesefehler auftraten. Die drei empfangenen Bytes werden miteinander verglichen und das Byte zurüchgegeben, welches am häufigsten auftritt. Die Funktion besitzt damit eine Ausführungszeit von mindestens $4 \cdot 500\,\mathrm{ns} = 2000\,\mathrm{ns}$. Das bedeutet, dass die zweite Schleife startet, wenn die IP-Emulation schon abgeschlossen ist. Damit benötigt auch sie nur einen Durchlauf.

Nach der Untersuchung der Schleifendurchläufe soll wieder auf die Kommunikationszeit eingegangen werden. Schreibvorgänge im Simulationsinterfaceblock sind definiert

als Schreiben eines Adressbytes und Schreiben eines Datenbytes. In jedem Simulationschritt müssen damit mindestens 36 Bytes übertragen werden. Ein Lesevorgang umfasst das Schreiben eines Adressbytes und Lesen von drei Datenbytes aus Gründen der Fehlerkorrektur. Die Lesevorgänge besitzen somit ein Kommunikationsaufkommen von 40 Bytes je Simulationsschritt. Die vorliegende Simulation besteht, nach den Simulationsausgaben in der Datei *risc_cpu.log* (siehe Anhang **??**), aus 278 Schritten inklusive der Initialisierung. Als Übertragungsgeschwindigkeit werden die gemessenen Werte aus Tabelle 5.4.3 herangezogen. Für die Kommunikationszeit ergibt sich

$$\frac{36\,\text{Byte}}{618404\,\text{Byte/s}} \cdot 278 + \frac{40\,\text{Byte}}{641881\,\text{Byte/s}} \cdot 278 = 0{,}0335\,\text{s} = 33{,}5\,\text{ms}$$

Damit lässt sich ein Teil der 57 Millisekunden Zeitunterschied zwischen der Simulation der originalen und der modifizierten RISC-CPU erklären.

Ein weitere Faktor, der die höhere Simulationszeit erklären kann, ist das Signalmapping in der Datei *exec.cpp*. In jedem Simulationsschritt müssen die Eingangssignale bitweise auf das Feld für die Simulationsdaten abgebildet werden. Eine Abbildung der empfangenen Daten von der IP-Emulation auf die lokalen Variablen findet in jedem Simulationsschritt statt. Dies kostet zusätzlich Zeit.

Im Vergleich zum generierten Modul hat das originale Modul sehr wenig Anweisungen auszuführen. Aus diesem Grund besitzt es einen Vorteil hinsichtlich der Simulationszeit. Es ist wahrscheinlich, dass der jetzige Geschwindigkeitsvorteil der Originaldesigns mit zunehmender Komplexität der SystemC-Beschreibung verloren gehen würde. In diesem Fall kann der Simulationsinterfaceblock auch zur Beschleunigung der SystemC-Simulation eingesetzt werden.

Bei der Durchführung der Messung ist nicht vorhersagbar, ob der Simulation über die gesamte Laufzeit die komplette CPU-Leistung zur Verfügung steht. Auf dem simulierenden PC werden weitere Prozesse des Betriebssystems ausgeführt. Falls das Betriebssystem während der Ausführung dem Simulator kurz den Prozessor entzieht, kommen Umschaltzeiten zwischen den Prozessen zur Simulationszeit hinzu. Damit erklären sich auch die Schwankungen bei den Messergebnissen der Simulationszeit.

Die Betrachtungen zeigen, dass die Simulationsgeschwindigkeit bei der Nutzung eines Simulationsinterfaceblocks stark von der Anzahl der ausgetauschten Daten zwischen SystemC-Simulation und IP-Emulation abhängt. Die HW/SW-Schnittstelle kann hier die Simulation ausbremsen. Das FPGA ist im vorliegenden Beispiel leistungsfähig genug, um nicht geschwindigkeitsmindernd zu wirken. Zur Erhöhung der Simulationsgeschwindigkeit ist der Einsatz einer physischen HW/SW-Schnittstelle denkbar, die eine höhere Übertragungsgeschwindigkeit als die parallele Schnittstelle erlaubt. In diesem Zusammenhang sollte dann auch die Leistungsfähigkeit der FPGA neu untersucht werden und gegebenenfalls ein schnelleres Modell eingesetzt werden, um ein Gleichgewicht zwischen Schnittstellengeschwindigkeit und FPGA-Leistung zu erreichen.

5.5. Schlussfolgerungen

Am vorgestellten Demonstrator wurde nachgewiesen, dass das Modell des Simulationsinterfaceblocks leistungsfähig genug ist, um die ihm gestellten Aufgaben korrekt zu erfüllen. Der Simulationsinterfaceblock wurde erfolgreich eingesetzt, um eine SystemC-Simulation mit einer IP-Emulation über eine HW/SW-Schnittstelle zu verbinden ohne dabei Änderungen an den Komponenten vorzunehmen.

In diesem Zusammenhang erfolgte außerdem der Nachweis, dass der Simulationsinterfaceblock-Generator erfolgreich arbeitet. Das Programm erstellte den Simulationsinterfaceblock auf Basis einer Analyse der VHDL-IP und Nutzereingaben. Dieser automatisch generierte Simulationsinterfaceblock erfüllte die Aufgabe der Verbindung von SystemC-Simulation und IP-Emulation zur Zufriedenheit.

Der Einsatz eines bereits bestehenden SystemC-Designs als Teil des Demonstrators zeigt, dass die Leistung des Simulationsinterfaceblocks und des Simulationsinterfaceblock-Generators ausreicht, um auch im praktischen Einsatz Verwendung zu finden.

6. Zusammenfassung und Ausblick

Diese Kapitel bildet den Abschluss der Arbeit. Es fasst die Konzepte und Ergebnisse dieser Arbeit zusammen und erläutert daran die erfolgreiche Bearbeitung der Aufgabenstellung. Anschließend wird ein Ausblick auf Möglichkeiten der Weiterentwicklung geboten.

6.1. Zusammenfassung der Arbeit

Diese Arbeit untergliedert sich in zwei Aufgaben. Die erste Aufgabe bestand in der Untersuchung eines Co-Simulationsansatzes. Dieser Ansatz zielt darauf ab, eine synthesefähige VHDL-Komponente auf einem FPGA zu emulieren und die Emulation mit einer SystemC-Simulation zu koppeln. Die zweite Aufgabe ist eng mit der Ersten verknüpft. Dabei sollte aufbauend auf den Betrachtungen des Co-Simulationsansatzes ein Verfahren entwicklet werden, das eine automatische Adaptierung von SystemC-Simulation und IP-Emulation erlaubt.

Der erste Teil dieser Arbeit bot eine Einführung in die Problematik und stellte die Aufgabenstellung vor. Anschließend erfolgte die Betrachtung des aktuellen Standes der Techniken mit denen diese Arbeit verknüpft ist. Daran schloss sich die Vorstellung von SystemC als Entwurfssystem und die Begutachtung seiner derzeitige Leistungsfähigkeit an. Der Begriff der Intellectual Properties wurde erklärt und das IPQ-Projekt kurz vorgestellt, mit dessen Hilfe bestehende Probleme bei der IP-Nutzung überwunden werden sollen. Zuletzt wurde auf den Begriff Simulation eingegangen und verschiedene Simulationsverfahren und -konzepte vorgestellt.

Das dritte Kapitel betrachtete den Co-Simulationsansatz. Dabei wurde herausgearbeitet, dass eine Hardware/Software-Schnittstelle benötigt wird, um die SystemC-Simulation mit der IP-Emulation zu koppeln. In diesem Zusammenhang erfolgte die Herausstellung der Aufgaben der HW/SW-Schnittstelle sowie der von ihr zu erbringenden Leistungen. Für eine mögliche Umsetzung der HW/SW-Schnittstelle fand das Modell des Interfaceblocks Verwendung. Dazu wurde zunächst desses Leistungsfähigkeit analysiert, um sicherzustellen, dass es in der Lage ist, die Aufgaben zu erfüllen. Da das vorliegende

Modell des Interfaceblocks eine Aufgabe nicht abdecken konnte, erfolgte eine Erweiterung das Konzept. Das Ergebnis bestand in einem erweiterten Interfaceblock, der die Hardware/Software-Grenze überwinden kann. Um den erweiterten Interfaceblock für die Co-Simulation einsetzen zu können, wurden Untersuchungen zur praktischen Umsetzung durchgeführt. Das Ergebnis bestand im Simulationsinterfaceblock, der eine Möglichkeit der praktischen Realisierung des erweiterten Interfaceblocks darstellt. Zum Abschluss des Kapitels wurden noch die verschiedenen Einsatzmöglichkeiten des Simulationsinterfaceblocks vorgestellt.

Das vierte Kapitel beschäftigte sich mit einer Lösung, die eine automatische Erstellung eines Simulationsinterfaceblocks erlaubt. Das Ergebnis bildet das Prorgamm Simulationsinterfaceblock-Generator. Es wurde detailiert auf die interne Arbeitsweise des Programms eingegangen, um eine gute Grundlage für Weiterentwicklungen des Programms zu bieten. Für die Anwendung des Programms zur Generierung von Simulationsinterfaceblöcken wurde ein Handbuch verfasst, um Benutzer einen schnellen Einstieg zu gewähren. Abschließend erfolgte die Vorstellung von Möglichkeiten der Weiterentwicklung zur Unterstützung eines breiteren Spektrums von Implementierungsplattformen und Intellectual Properties.

Der Demonstrator, über den das fünfte Kapitel handelt, diente dem Erproben und Validieren der Konzepte aus Kapitel drei und des Programm aus Kapitel vier. Es wurde der Aufbau und die Funktionsweise des Demonstrators vorgestellt. Anschließend wurde der Nachweis der korrekten Funktion durch die Ausführung des Demonstrators und dem Vergleich mit dem Ursprungsdesign erbracht. Da benötigte Zeit einer Simulation ein wichtiges Kriterium bei der Wahl der Simulationsart darstellt, schlossen sich dem Nachweis der Funktion Untersuchungen zur Simulationsgeschwindigkeit an. In diesem Zusammenhang wurde eine Messeinrichtung entwickelt, die die Geschwindigkeit der parallelen Schnittstelle ermittelte. Die Untersuchungen ergaben, dass die Simulationsgeschwindigkeit stark von der Geschwindigkeit der physischen HW/SW-Schnittstelle abhängt. Das Ergebnis des Demonstrators bestand darin, dass die erarbeiteten Konzepte dieser Arbeit ihre Aufgaben korrekt erfüllten und erfolgreich in der Praxis eingesetzt werden konnten. Auch das Prorgamm Simulationsinterfaceblock-Generator erfüllte seinen Zweck und erlaubt die automatische Adaptierung einer IP-Emulation und einer SystemC-Simulation. Anhand des Demonstrators wurde gezeigt, dass die gestellten Aufgaben dieser Arbeit erfolgreich gelöst wurden.

6.2. Ausblick für den erweiterten Interfaceblock

Abgeleitet vom Konzept des erweiterten Interfaceblocks wurde in dieser Arbeit die Implementierung eines Simulationsinterfaceblocks durchgeführt. Er dient zur Kopplung einer SystemC-Simulation auf einem PC und einer IP-Emulation auf einem FPGA. Im Bezug auf die Cosimulation könnten weiterer Arbeiten folgende Themen untersuchen:

- Unter Nutzung eines Simulationsinterfaceblocks könnte ein Tool für das Hardwaredebugging implementiert werden. Dies bietet die Möglichkeit, das Verhalten eines Hardwaredesigns als Implementierung auf einem FPGA zu untersuchen.
- Ein weiteres Feld für Untersuchungen liegt in der Emulation von dynamisch rekonfigurierbaren Designs. Dabei kann die Kopplung mit einer SystemC-Simulation erfolgen, um dynamisch rekonfigurierbare Teile eines größeren Designs zu untersuchen oder ein Hardwaredebugging des dynamisch rekonfigurierbaren Designs durchzuführen.

Um die Einsatzmöglichkeiten des allgemeinen Konzeptes des erweiterten Interfaceblocks zu erweitern, könnten Untersuchungen auf den folgenden Gebieten behilflich sein:

- Ein Gebiet von Untersuchung könnte der Einsatz als Schnittstelle zwischen Anwendungsprogrammen und Spezialhardware sein. Der Einsatz von Spezialhardware beschleunigt dabei bestimmte Funktionen, deren Ausführung in Software zu langsam ist, um eine Steigerung der Programmperformance zu erreichen.
- Weiterhin könnte das Konzept des erweiterten Interfaceblocks in das bestehende Interface Synthese (IFS)-Format aufgenommen werden und damit auch die Möglichkeit einer automatischen Generierung eines erweiterten Interfaceblock als HW/SW-Schnittstelle entstehen.

6.3. Ausblick für den Simulationsinterfaceblock-Generator

Die Entwicklung des Simulationsinterfaceblock-Generators im Rahmen dieser Arbeit fand speziell für die automatische Adaptierung von SystemC-Simulation und VHDL-IP-Emulation statt. Um ein größeres Einsatzfeld des Programms zu erlangen und damit die praktische Einsatzfähigkeit weiter zu steigern, sind einige Erweiterungen des Programms denkbar:

- In der gegenwärtigen Version verarbeitet der Simulationsinterfaceblock-Generator nur Ports von VHDL-Entities mit den Richtungen in oder out. Zur Unterstützung einer größeren Anzahl von VHDL-IPs könnte das Programm für die Verarbeitung von bidirektionalen Ports (Richtung inout) erweitert werden.
- Mit der Analysemöglichkeit von VHDL-Generics kann das Konzept von generischen Portbreiten umgesetzt werden. Damit ist auch eine Unterstützung von parametrisierbaren IPs denkbar.
- Die Erweiterung des Programms zur Unterstützung anderer Hardwarebeschreibungssprachen, wie zum Beispiel Verilog, würde die Einsatzbreite erheblich steigern.
- Eine Unterstützung unterschiedlicher FPGA-Typen und -Boards bringt eine größere Flexibilität des Programms hinsichtlich der Emulationsplattform mit sich. In diesem Zusammenhang steht die Nutzung von unterschiedlichen Entwicklungsumgebungen und Tools zur Synthese und Implementierung.
- Andere phyische Hardware/Software-Schnittstellen können zum Einen die Simulationsgeschwindigkeit erhöhen. Auf der anderen Seite erhöht sich dadurch ebenfalls die Flexibilität gegenüber unterschiedlichen Implementierungsplattformen.

A. Herleitung der Formel zur Berechnung der Taktfrequenz der IP-Emulation

Zur Herleitung der Formel zur Berechnung der Taktfrequenz der IP-Emulation sind zunächst eine Variablendefinitionen durchzuführen. Die verwendeten Variablen definieren sich wie folgt:

f_{IFB}	...	Taktfrequenz des Interfaceblocks
T_{IFB}	...	Taktperiode des Interfaceblocks
f_{IP}	...	Taktfrequenz der IP-Emulation
T_{IP}	...	Taktperiode der IP-Emulation
t_{IP_H}	...	Zeit des Highpegels der Taktperiode der IP-Emulation
t_{IP_L}	...	Zeit des Lowpegels der Taktperiode der IP-Emulation
$HDIV$	...	Wert des Zählers HDIV
$LDIV$	...	Wert des Zählers LDIV
n_H	...	Anzahl der Durchläufe durch den Zustand Clk_gen1
n_L	...	Anzahl der Durchläufe durch den Zustand Clk_gen0

Der Interfaceblock ist mit der Taktfrquenz f_{IFB} getaktet. Die Taktperiode ergibt sich aus der Taktfrequenz nach der Formel

$$T = \frac{1}{f} \tag{A.1}$$

Somit ergibt sich für die Taktperiode des Interfaceblocks

$$T_{IFB} = \frac{1}{f_{IFB}} \tag{A.2}$$

Der endliche Automat in der $\mathrm{CU_{HW}}$ generiert den Highpegel als auch den Lowpegel des Taktsignals für die IP-Emulation mindestens einen Takt lang. Daraus folgt für die minimalen Pegelzeiten der Taktperiode der IP-Emulation

$$t_{IP_{H,min}} = T_{IFB} \qquad t_{IP_{L,min}} = T_{IFB} \tag{A.3}$$

Die Taktgenerierung erfolgt in den Zuständen *Clk_gen1* und *Clk_gen0* des Automaten der CU_{HW}. Die Verweildauer in diesen Zuständen geben die Werte der Zähler HDIV und LDIV an. Aber jeder dieser Zustände wird mindestens einmal durchlaufen. Die Anzahl der Durchläufe dieser Zustände ergibt sich aus

$$n_H = HDIV + 1 \qquad n_L = LDIV + 1 \tag{A.4}$$

Die Anzahl der Durchläufe bestimmt, für welche Zeitspanne der jeweilige Pegel generiert wird. Die Zeitspanne beträgt für den jeweiligen Pegel aber mindestens $t_{IP_{H,min}}$ bzw. $t_{IP_{L,min}}$. Für die Gesamtzeit des High- bzw. Lowpegels ergibt sich

$$t_{IP_H} = n_H \cdot t_{IP_{H,min}} \qquad t_{IP_L} = n_L \cdot t_{IP_{L,min}} \tag{A.5}$$

$$t_{IP_H} = (HDIV + 1) \cdot t_{IP_{H,min}} \qquad t_{IP_L} = (LDIV + 1) \cdot t_{IP_{L,min}} \tag{A.6}$$

Aus den Formeln A.3 und A.6 bildet sich

$$t_{IP_H} = (HDIV + 1) \cdot T_{IFB} \qquad t_{IP_L} = (LDIV + 1) \cdot T_{IFB} \tag{A.7}$$

Die Dauer der Taktperiode für die IP-Emulation ergibt sich aus der Zeit des Highpegels und der Zeit des Lowpegels.

$$T_{IP} = T_{IP_H} + T_{IP_L} \tag{A.8}$$

$$\text{mit A.7:} \qquad T_{IP} = (HDIV + 1) \cdot T_{IFB} + (LDIV + 1) \cdot T_{IFB} \tag{A.9}$$

$$T_{IP} = (HDIV + 1 + LDIV + 1) \cdot T_{IFB} \tag{A.10}$$

$$T_{IP} = (HDIV + LDIV + 2) \cdot T_{IFB} \tag{A.11}$$

$$\tag{A.12}$$

Nach Formel A.1 werden die Taktperioden in Formel A.11 durch Frequenzen ersetzt

$$\frac{1}{f_{IP}} = (HDIV + LDIV + 2) \cdot \frac{1}{f_{IFB}} \tag{A.13}$$

$$f_{IP} = \frac{f_{IFB}}{HDIV + LDIV + 2} \tag{A.14}$$

B. Hinweise zur Nutzung von SystemC in Mircrosoft Visual C++ 6.0

Um die SystemC Klassenbibliothek erfolgreich in der C/C++ Entwicklungsumgebung *Microsoft Visual C++ 6.0* einsetzen zu können, müssen einige Einstellungen vorgenommen werden. Sie betreffen den Compiler und den Linker und sind unter dem Menüpunkt `Project -> Settings...` vorzunehmen.

Die Einstellungen sind im Folgenden aufgeführt. Dabei ist als erstes die Registerkarte aufgeführt, unter dem sich die Einstellung findet. Es schließt sich der Unterpunkt und die jeweilige Parameter an. Der Wert, der für den Parameter einzutragen ist, steht darunter mit einer Bemerkung in Klammern.

```
C/C++ -> C++ Language -> Enable Run-Time Type Information (RTTI)
      aktiv (Haken eingetragen)

C/C++ -> Preprocessor -> Additional include directories
      ...\Systemc-2.0.1\src ( Pfad zu systemc.h)

Link -> General -> Object/library modules
      systemc.lib (hinzuf\"{u}gen zu vorhandenen Bibliotheken)

Link -> Input -> Additional library path
      ...\Systemc-2.0.1\msvc60\systemc\Debug
      (Pfad zur Datei systemc.lib)
```

Weitere Hinweise zur Arbeit mit SystemC enthält das Handbuch [Opeb].

Literaturverzeichnis

[AB96] Sorin A. Huss Klaus Waldschmidt Andreas Bleck, Michael Goedecke. *Praktikum des modernen VLSI-Entwurfs*. B. G. Teubner, Stuttgart, 1996.

[Alt] Altera Corporation. Altera: Leaders in FPGAs, CPLDs, and Structured ASICs. http://www.altera.com/. Homepage.

[Apt] Aptix® Corporation. www.aptix.com. Homepage.

[Apt00] Aptix® Corporation. *System Explorer™ - Reconfigurable System Prototyping for SoC Emulation*, 2000. Broschüre zum System Explorer™.

[Atm] Atmel Corporation. Atmel Corporation. http://www.atmel.com/. Homepage.

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

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

[Dig] Digilent Inc. http://www.digilentinc.com. Homepage.

[Dig02] Digilent Inc. *Digilab 2E Reference Manual*, 14. April 2002. www.digilentinc.com.

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

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

[Fla03] Marcel Flade. FPGA-basierte Fail-safe-Schnittstellen für eingebettete Systeme. Studienarbeit, Technische Universität Chemnitz, 2003.

[GL94] Manfred Selz Gunther Lehmann, Bernhard Wunder. *Schaltungsdesign mit VHDL - Synthese, Simulation und Dokumentation digitaler Schaltungen.* Franzis-Verlag GmbH, Poing, 1994.

[Har02] Dr. Wolfram Hardt. *Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme.* Shaker Verlag, Aachen, 2002.

[Har03] Prof. Dr. Wolfram Hardt. Hardware-Software Codesign. Vorlesung, Technische Universität Chemnitz, 2003.

[Har04] Prof. Dr. Wolfram Hardt. Hardware-Software Codesign II. Vorlesung, Technische Universität Chemnitz, 2003/2004.

[HVI01] Wolfram Hardt, Markus Visarius, and Stefan Ihmor. Rapid prototyping of real-time interfaces. In *Field Programmable Logic (FPL) - Poster Session*, Belfast, Northern Ireland, UK, October 2001.

[IBJK+03] Stefan Ihmor, Nilson Bastos Jr., Rafael Cardoso Klein, Markus Visarius, and Wolfram Hardt. Rapid Prototyping of Realtime Communication - A Case Study: Interacting Robots. June 2003.

[IH04] Stefan Ihmor and Wolfram Hardt. Runtime Reconfigurable Interfaces - The RTR-IFB Approach. 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 3, April 2004. Santa Fe, New Mexico, USA.

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

[Int] Intel Coporation. Intel Research - Silicon - Moore's Law. http://www.intel.com/research/silicon/mooreslaw.htm. Homepage.

[ipq] Projekt IPQ - Home Page. https://www.ip-qualifikation.de.

[IVH02a] Stefan Ihmor, Markus Visarius, and Wolfram Hardt. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. Montreal, Canada, Aug. 2002.

[IVH02b] Stefan Ihmor, Markus Visarius, and Wolfram Hardt. A Design Methodology for Application-specific Real-Time Interfaces. In *Proc. of the International Conference on Computer Design*, Freiburg, Germany, Sept. 2002.

[IVH03] Stefan Ihmor, Markus Visarius, and Wolfram Hardt. Modeling of Configurable HW/SW-Interfaces. pages 51 – 60, Feb. 2003.

[Lat] Lattice Semiconductor Corporation. FPGA, CPLD and SERDES Programmable Logic Devices by Lattice Semiconductor. http://www.vantis.com/. Homepage.

[Mar02] Norbet Schuhmann Martin Speitel. Erfahrungen mit Intellectual Property. *www.elektroniknet.de*, 2002.

[Mäd] Andreas Mäder. Vhdl kompakt. http://tech-www.informatik.uni-hamburg.de/vhdl/doc/kurzanleitung/vhdl.pdf.

[Mena] Mentor Graphics Corp. www.mentor.com. Homepage.

[Menb] Mentor Graphics Corp. High-Performance System Integration Verification: VStation. http://www.mentor.com/vstation/. Homepage.

[Menc] Mentor Graphics Corp. VStationPRO High-Performance System Verification. http://www.mentor.com/vstation/vstation_pro.html. Homepage.

[Mend] Mentor Graphics Corp. VStationTBX - High-Performance Verification Accelerator. http://www.mentor.com/vstation/vstation_tbx.html. Homepage.

[Men03] Mentor Graphics Corp. *VStationPRO High-Performance System Verification*, 2003. Broschüre zur VStationPro.

[Men04] Mentor Graphics Corp. *VStationTBX - Datasheet*, 2004. Broschüre zur VStationTBX.

[Mic04] Microsoft Corporation. Geistiges Eigentum - Definition. http://www.microsoft.com/germany/digital-mentality/definition.mspx, 2004. Homepage.

[Mül02] Prof. Dr. Dietmar Müller. Entwurfssysteme. Vorlesung, Technische Universität Chemnitz, 2002.

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

[Mod] Model Technology. www.model.com. Homepage.

[Mod03] Model Technology. *ModelSim® LE User's Manual*, Dezember 2003. Version 5.8a.

[Mon00a] Prof. Dr.-Ing. Dieter Monjau. Rechnerorganisation. Vorlesung, Technische Universität Chemnitz, 2000.

[Mon00b] Prof. Dr.-Ing. Dieter Monjau. VHDL - Einführung. Unterlagen zur Vorlesung Rechnerorganisation, Technische Universität Chemnitz, 2000.

[Nau99] Dr. Bernt Naumann. Digitaltechnik. Vorlesung, Technische Universität Chemnitz, 1999.

[Nau01] Dr. Bernt Naumann. Werkzeuge für den Systementwurf. Vorlesung, Technische Universität Chemnitz, 2001.

[Opea] Open SystemC Initiative. SystemC™ Homepage. http://www.systemc.org. Homepage.

[Opeb] Open SystemC Initiative. *SystemC™ User's Guide*. Version 2.0.

[Ope02] Open SystemC Initiative. *Functional Specification for SystemC™ 2.0*, April 2002. Version 2.0-Q.

[Pea04] Craig Peacock. Beyond Logic. http://www.beyondlogic.org/, Juli 2004. Webseite.

[Ram89] Fanz-J. Rammig. *Systematischer Entwurf digitaler Systeme*. B. G. Teubner, Stuttgart, 1989.

[Scia] Scientific Software Tools Inc. http://www.driverlinx.com/DownLoad/-DlPortIO.htm. Downloadseite von port95nt.exe.

[Scib] Scientific Software Tools Inc. SST - Data Acquisition Hardware/Software and Custom Applications. http://www.driverlinx.com. Homepage.

[Syn] Synopsys Corporate Marketing. Synopsys® Homepage. http://www.synopsys.com. Homepage.

[Syn03] Synopsys Inc. *CoCentric® SystemC™ Compiler - RTL User and Modeling Guide*, Juni 2003. Version U-2003.06, www.synopsys.com.

[Tei97] Dr.-Ing. Jürgen Teich. *Digitale Hardware/Software-Systeme - Synthese und Optimierung*. Springer-Verlag Berlin, Heidelberg, 1997.

[Thaa] Tharas Systems Inc. www.tharas.com. Homepage.

[Thab] Tharas Systems Inc. *Hammer® 100 Hardware Accelerator for Verilog, VHDL and Mixed language simulation*. Broschüre zum Hammer® 100 System.

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

[VA] VA Software Corp. SystemC™Open Source Lizenz. http://www.systemc.org/web/sitedocs/open_source_licensing.html.

[Ver96] Verein Deutscher Ingenieure. *VDI 3633 - Simulation von Logistik-, Materialfluß- und Produktionssystemen - Begriffsdefinition*, November 1996.

[VH04] Markus Visarius and Wolfram Hardt. The IPQ Format – An Approach to Support IP based Design. In *Proc. of the GI/ITG/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 106 – 115, Kaiserslautern, Germany, Feb. 2004.

[VLH+03] M. Visarius, J. Lessmann, W. Hardt, F. Kelso, and W. Thronicke. An XML Format based Integration Infrastructure for IP based Design. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI 2003)*, pages 119 – 124, São Paulo, Brazil, 08. - 10. September 2003. IEEE Computer Society.

[VLKH04] Markus Visarius, Johannes Lessmann, Frank Kelso, and Wolfram Hardt. Generic integration infrastructure for ip based design processes and tools with a unified xml format. *Integration, the VLSI journal*, 37(4):289 – 321, September 2004.

[Vra98] Hendrikus P.E. Vranken. *Design for test and debug in hardware/software systems*. Technische Universität Eindhoven, Eindhoven, 1998.

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

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

[Wika] Wikimedia Foundation Inc. Wikipedia - Die freie Enzyklopädie. http://de.wikipedia.org/wiki/Adapter. Definition: Adapter.

[Wikb] Wikimedia Foundation Inc. Wikipedia - Die freie Enzyklopädie. http://de.wikipedia.org/wiki/Schnittstelle. Definition: Schnittstelle.

[Wor] World Intellectual Property Organization. WIPO - World Intellectual Property Organization. http://www.wipo.int/. Homepage.

[Xila] Xilinx Inc. Development System Reference Guide - ISE5. http://toolbox.xilinx.com/docsan/xilinx5/pdf/docs/dev/dev.pdf.

[Xilb] Xilinx Inc. Xilinx: Design Tools Center. http://www.xilinx.com/products/-design_resources/design_tool/index.htm. Homepage.

[Xilc] Xilinx Inc. Xilinx: Programmable Logic Devices, FPGA & CPLD. http://www.xilinx.com. Homepage.

[Xild] Xilinx Inc. Xilinx Synthesis Technology (XST) User Guide. http://toolbox.xilinx.com/docsan/xilinx5/pdf/docs/xst/xst.pdf.

[Xil03] Xilinx Inc. *Spartan-IIE 1.8V FPGA Family: Complete Data Sheet*, 9. Juli 2003. Version 2.1, http://direct.xilinx.com/bvdocs/publications/ds077.pdf.

Abkürzungsverzeichnis

ASIC Application-Specific Integrated Circuit

CD Compact Disc

CPU Central Prozessing Unit

CU Controlunit

DSP Digitaler Signalprozessor

EDA Electronic Design Automation

EPP Enhanced Parallel Port

EPS Encapsulated PostScript

FPGA Field Programmable Gate Array

FPIC Field Programmable Interconnect Component

FSM Finite State Machine

HDL Hardware Description Language

HW Hardware

IEEE Institute of Electrical and Electronics Engineers

IF Interface

IFB Interfaceblock

IFS Interface Synthese

I/O Input-Output

IP Intellectual Property

ISS Instruction-set Simulator

MMX Multimedia Extension

OSCI Open SystemC Initiative

PDF Portable Document Format

PC Personal Computer

PH Protokollhandler

RAM Random Access Memory

RISC Reduced Instruction Set Computing

ROM Read Only Memory

RT Register-Transfer

RTL Register-Transfer-Level

RTOS Real-Time Operation System

SH Sequenzhandler

SimIFB Simulationsinterfaceblock

SoC System-on-Chip

SRAM Static Random Access Memory

SW Software

VDI Verein Deutscher Ingenieure

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

XML Extensible Markup Language