Prototypen-Entwicklungsumgebung zur schnellen Simulation von DSP-Algorithmen

Georg Maier
Mat. Nr.: 00 0 0102 029

In Zusammenarbeit Mit
Dem Bereich Informationstechnologien,
D.I. Gerhard Humer
Der ARC Seibersdorf Research GmbH

Betreuer:
Prof. Dr. Markus Rupp
Dr. Mario Huemer

DIPLOMARBEIT

eingereicht am
Fachhochschul-Diplomstudiengang

Hardware/Software Systems Engineering

in Hagenberg

im Juli 2004

© Copyright 2004 Georg Maier

Alle Rechte vorbehalten

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 24. Juni 2004

Georg Maier

Inhaltsverzeichnis

Erklärung **iii**

Vorwort **viii**

Kurzfassung **ix**

Abstract **x**

1 Einleitung **1**
1.1 Zielsetzung und Motivation 1
1.2 Aufbau der Arbeit 2

2 Hardwarebeschreibung **4**
2.1 ARCS RTS-DSP-Board 4
2.2 Sundance Carrier Board 5
2.3 Sundance TIM Modul SMT365 6

3 Softwarebeschreibung **7**
3.1 Matlab Schnittstellen 7
3.1.1 Matlab mex-Dateien 7
3.1.2 Simulink S-Funktionen 9
3.2 Kurzbeschreibung GenC 9
3.2.1 Aufbau einer Generic C Datei 10
3.3 Sundance SMT6025 11
3.4 ARCS LANFlasher 13

4 GenCaddOn: Erweiterung zu GenC **14**
4.1 Warum GenCaddOn? 14
4.2 Funktionsweise 15
4.2.1 Warum neue Dateien? 15
4.2.2 Vorraussetzungen 15
4.3 Umwandlungsablauf 16
4.4 Erstellte Dateien und deren Besonderheiten 22
4.4.1 Ergebnis ARCS 22

4.4.2 Ergebnis Sundance 26

5 Matlab-Hardwareinterface 29
5.1 Allgemeines 29
5.2 Aufbau 29
5.2.1 Schnittstelle zu Matlab 30
5.2.2 Datenumwandlung 30
5.2.3 Hardwarekommunikation 32
5.2.4 Globale Projektparameter 32
5.3 Mögliche Befehle 34

6 Tests und Beispiel 35
6.1 Simulationsablauf 35
6.2 Test der Funktionsweise 35
6.2.1 Matlab-Hardwareinterface 36
6.2.2 GenCaddOn 37
6.3 Analyse des Ergebnisses 38

7 Schlussbemerkungen 43

A Verzeichnisstruktur f. d. Simulation 44
A.1 Hauptverzeichnis 44
A.2 Unterverzeichnis für ARCS 44
A.3 Unterverzeichnis für Sundance 45

B Matlab-Scripte 47
B.1 Simulationsablauf für das ARCS RTS-DSP-Board `compileRTS.m` 47
B.2 Matlab-ARCS-Hardware-Interface `exampleRTS.m` 48
B.3 Simulationsablauf für das Sundance-Modul `compileSMT.m` . . 48
B.4 Matlab-Sundance-Hardware-Interface `exampleSMT.m` 49

C Generierte Dateien 50
C.1 Generic C Datei `dsp.gc` 50
C.2 Dateien für das ARCS DSP-Board 51
C.2.1 Projektdatei 51
C.2.2 DSP-Main Datei `dspRTS.c` 52
C.2.3 DSP Wrapper H-Datei `DspPcIO.h` 53
C.2.4 DSP Wrapper C-Datei `DspPcIO.c` 54
C.2.5 DSP GlobalTypes H-Datei `globalTypesDsp.h` 57
C.2.6 Simulink S-Funktion Datei `dspIO.cpp` 57
C.2.7 PC Wrapper H-Datei `PcDspIO.h` 61
C.2.8 PC Wrapper C-Datei `PcDspIO.c` 61
C.2.9 PC GlobalTypes H-Datei `globalTypesPc.h` 64
C.3 Sundance Dateien 65

C.3.1 Projektdatei `dspSMT.CFG` 65
C.3.2 Projektdatei `MAKEFILE.smt` 65
C.3.3 DSP Main-Datei `dspSMT.c` 65
C.3.4 DSP Wrapper H-Datei `DspPcIO.h` 67
C.3.5 DSP Wrapper C-Datei `DspPcIO.c` 67
C.3.6 Simulink S-Funktion Datei `dspIO.cpp` 69
C.3.7 PC Wrapper H-Datei `PcDspIO.h` 72
C.3.8 PC Wrapper C-Datei `PcDspIO.cpp` 73

D Befehle für das Hardware-Interface 76
D.1 Befehle und deren Parameter für comRTS 76
D.1.1 con2Board 76
D.1.2 send 77
D.1.3 receive 78
D.1.4 saveStruct 79
D.1.5 close 79
D.1.6 clearStruct 79
D.1.7 resetBoard 80
D.1.8 open 80
D.1.9 sendregcmd 81
D.1.10 waitforregcon 81
D.2 Befehle und deren Parameter für comSMT 82
D.2.1 init 82
D.2.2 loadProg 82
D.2.3 resettim 82
D.2.4 resetboard 83
D.2.5 cpwrite 83
D.2.6 sendpci 83
D.2.7 sendsram 84
D.2.8 cpread 84
D.2.9 receivepci 85
D.2.10 receivesram 85
D.2.11 close 86
D.2.12 saveStruct 86
D.2.13 clearStruct 86

E Inhalt der CD-ROM 87
E.1 Diplomarbeit und Dokumente 87
E.2 *LaTeX*-Dateien 87
E.3 Abbildungen 88
E.4 Sonstiges 89
E.5 Kurzfassung 89
E.6 Abstract 89
E.7 Beispiele 89

E.8 Programme 90
E.9 Sourcecodes 90

Literaturverzeichnis **91**

Vorwort

Diese Diplomarbeit entstand in Zusammenarbeit mit verschiedenen Personen und Institutionen.

An dieser Stelle möchte ich mich ganz herzlich bei Prof. Markus Rupp bedanken, der diese Arbeit erst ermöglicht hat und ich bedanke mich für seine hervorragende Betreuung während dieser Zeit.

Weiters möchte ich mich bei meinen Kollegen von ARCS für die Unterstützung bedanken.

Danken möchte ich auch dem gesamten Team der Fachhochschule Hagenberg für die gute Ausbildung.

Meiner Freundin, Anna Rainer, und meinen Freunden gebührt Dank dafür, mich immer aufgemuntert und unterstützt zu haben.

Zu guter Letzt möchte ich meinen Eltern und meiner Familie danken, die mir das Studium und damit auch diese Diplomarbeit erst ermöglicht haben.

Kurzfassung

Unter Rapid-Prototyping versteht man, eine Idee oder Algorithmus so schnell als möglich in ein Echtzeitsystem umzusetzen. Eines der größten Probleme des Rapid-Prototyping sind die unterschiedlichen Programmiersprachen, die für die Prototypenentwicklung benötigt werden. Das Umsetzen der Simulation in einen ausführbaren Code für die Hardwareplattform ist äußerst fehleranfällig, wodurch sich üblicherweise der Entwicklungszyklus erheblich verlängert. Der Einsatz von automatischen Werkzeugen zur Umwandlung von Code zwischen den einzelnen Plattformen kann daher die Entwicklungszeit deutlich verringern. Nicht nur die automatischen Werkzeuge an sich, sondern auch der Umstand, dass dafür auch nur **ein** Code notwendig ist, trägt zu einer schnelleren und fehlerfreieren Entwicklung bei.

Die Simulation auf Hardware auszulagern kann aus zwei Gründen sinnvoll sein. Erstens um den erstellten Algorithmus unter Echtzeitbedingung zu testen und zweitens um die Simulation zu beschleunigen.

Die Basis für den gesamten Umwandlungsprozess bildet eine Datei, deren Code in *Generic C*[1] beschrieben ist. Die Umsetzung in den Code für Simulink und die Echtzeitentwicklungsumgebung wird mit dem Programm *GenC*[2] durchgeführt. Diese Arbeit beschäftigt sich mit der Portierung dieser Ergebnisse auf zwei verschiedene DSP-Plattformen, sowie der Erstellung eines Interfaces zwischen Matlab oder Simulink und den beiden Hardwaresystemen. Dadurch soll es möglich sein aus einem Simulink Modell einzelne Blöcke aus der Entwicklungsumgebung auf den DSP auszulagern und unter Echtzeitbedingungen zu testen.

Durch das im Zuge dieser Arbeit entwickelte Programm *GenCaddOn* wird es nun ermöglicht, für beide Systeme Codes zu erzeugen und die Hardware in die Simulation einzubeziehen. Die Grundfunktionalität besteht in der Adaptierung der *DSP-Datei* und *S-Funktion*, sowie in der Erstellung der Schnittstellendateien für den Zugriff auf die Hardware und den Datenaustausch. Weiters bietet es die Option, Projektdateien für den DSP zu erstellen. Dadurch wird es möglich, den gesamten Entwicklungsablauf in Matlab durchzuführen.

[1] ähnlich Standard C, definiert unter COSSAP
[2] Programm zur Umwandlung von Generic C Code

Abstract

Rapid prototyping means to convert an idea or an algorithm so fast as possible into a real time system. One of the most known problems of rapid prototyping is the different programming languages which are used for development. The conversion of the simulation into an executable code for the hardware platform is extremely error-prone, whereby also the development cycle extends substantially. The use of automatic tools to convert therefore drastically between the individual platforms can reduce the development time. Errors which are possibly made by conversion done by a user can be widely avoided. Also the circumstance that only **one** code is necessary contributes to the fact that the automatic tools are a faster and error free development.

To out source the simulation on hardware can be meaningful for two reasons. The first reason is to test the developed algorithm under real time conditions and secondly to accelerate the simulation.

The basis for the entire process of transformation forms a file, whose code is described in *Generic C*[3]. The conversion to Simulink code and the real time development environment is accomplished by a program called *GenC*[4]. This thesis deals with the conversion of these results onto two different DSP platforms, as well as the production of an interface between Matlab or Simulink and the two hardware systems. Thus it should be possible to out source out an individual block from a Simulink model onto the DSP to run the test under real-time conditions.

During the development process of this thesis a program called *GenCaddOn* was developed to generate code which is able to use the hardware functionality in the simulation. Basic functionality exists in the adaptation of the *DSP file* and *s-function*, as well as in the production of the interface files for the access to the hardware and data exchange. Further more, it offers the option to generate project files for the DSP. Thereby it is possible to accomplish the entire expiration of development in Matlab.

[3]similarly standard C, defines under COSSAP

[4]Program to convert Generic C code.

Kapitel 1

Einleitung

1.1 Zielsetzung und Motivation

Das Ziel von Rapid-Prototyping ist es, eine Idee, oder einen schon vorhandenen Algorithmus, so schnell als möglich in ein Echtzeitsystem umzusetzen [2], [3]. Ein Problem mit dem man dabei konfrontiert wird, sind die unterschiedlichen Plattformen, die für die Entwicklung benötigt werden. So ist in den meisten Fällen der Code für die Simulation und der ausführbare Code in unterschiedlichen Programmiersprachen zu schreiben. Die Umsetzung zwischen den verschiedenen Plattformen durch den Entwickler ist äußerst fehleranfällig. Wünschenswert ist es, den Algorithmus in nur einer Programmiersprache zu definieren und durch automatische Werkzeuge in das benötigte Format umzuwandeln.

Das Ziel der Diplomarbeit ist es, eine Umgebung zu schaffen, die es ermöglicht C-Code für den TI C64 DSP schnell umzusetzen. Der so gewonnene Code soll

- als Simulink-Datei (S-Funktion[1]),
- weiters unter Simulink auf dem C64 und auch
- als stand-alone Datei auf dem C64 ablauffähig sein.

Weiters soll die Möglichkeit bestehen, aus einem Simulink-Modell einzelne Blöcke aus der Entwicklungsumgebung auf den DSP auszulagern und unter Echtzeitbedingungen zu testen. Für die Simulation bedeutet dies, dass ein Block den Datenaustausch zwischen Hardware und PC steuert. Die Entwicklung des Codes soll völlig unabhängig von der Zielplattform erstellt werden können.

Bei den schon öfters erwähnten Hardwaresystemen handelt es sich um ein DSP/FPGA Modul [2] der Firma *Sundance*, welches sich im PC befindet,

[1] Bietet die Möglichkeit eigene Funktionalität in Form von C-Code einzubinden.
[2] SMT 365-8-2 Modul und SMT310Q PCI Trägerkarte

und einem stand-alone DSP-Board der Firma *ARC*[3] *Seibersdorf Research*, welches über Ethernet mit dem PC verbunden ist.

Diese Umgebung soll es ermöglichen C-Code sehr schnell unter Simulink zu entwickeln. Dies würde es erlauben, ihn auf dem C64 zu testen und damit die Simulation schneller ablaufen zu lassen. Außerdem soll dieser auf dem C64 als eigenständiger Code, oder im Verbund mit anderen Codes lauffähig sein.

Dadurch, dass in der angestrebten Umgebung der Code nur durch automatische Werkzeuge verändert wird, nimmt man dem Entwickler nicht nur Arbeit ab, sondern man erreicht auch, dass keine zusätzlichen Fehler bei der Portierung des Codes von einer auf die andere Plattform hinzugefügt wird. Diese beiden Eigenschaften tragen zu einer dramatischen Verkürzung der Entwicklungszeit bei.

1.2 Aufbau der Arbeit

An dieser Stelle soll noch ein kleiner Ausblick auf die folgende Arbeit erfolgen.

Kapitel 2: Am Beginn der Arbeit steht eine kurze Beschreibung der beiden verwendeten Hardwaresysteme. Durch dieses Kapitel sollen dem Leser kurz die Funktionsweise und Besonderheiten der verwendeten Plattformen näher gebracht werden. Dies soll zu einem besseren Verständnis für Probleme und den daraus resultierenden Lösungsansätzen während der Entwicklung dienen.

Kapitel 3: Im dritten Kapitel werden verwendete Programme und spezielle Matlab Funktionen, auf denen diese Arbeit aufbaut, betrachtet. Bei den Programmen handelt es sich um das schon erwähnte *GenC* bzw. dem LANFlasher[4]. Da es sich bei diesen beiden nicht um Standardprogramme handelt, besteht die Notwendigkeit, diese dem Leser ein wenig näher zu bringen, um die darauf aufbauende Funktionalität besser verstehen zu können. Bei Matlab handelt es sich um die Funktionalität von mex-Dateien und S-Funktionen. Diese beiden Abschnitte sollen unerfahrenen Lesern deren Prinzip näher bringen, da diese die Basis für Teile dieser Arbeit bilden.

Kapitel 4: Dieses Kapitel soll die Frage beantworten, warum ein eigenes Programm entwickelt wurde, weiters wird dessen Funktionsweise erklärt. Darunter fallen auch die notwendigen Vorraussetzungen für einen fehlerfreien Umwandlungsablauf. In weiterer Folge werden die daraus resultie-

[3] Austrian Research Centers

[4] Programm zum Programmieren des DSPs auf dem RTS-Board.

renden Ergebnisse eingehend betrachtet. Auch die Möglichkeiten, die dieses Programm bietet, werden in diesem Kapitel angeführt.

Kapitel 5: Im Zuge dieser Arbeit wurde neben dem vorhin erwähnten Programm auch ein Interface zwischen Matlab und den beiden Hardwareplattformen realisiert. In Kapitel 5 wird der Grund für die Realisierung sowie der Aufbau beschrieben. Als Basis wird hier wiederum die Funktionalität von mex-Dateien verwendet.

Kapitel 6: Am Ende der Arbeit werden die erzielten Ergebnisse in Form von Tests dem Leser näher gebracht. Es wird hier ebenfalls beschrieben wie die korrekte Funktionsweise überprüft wurde. In diesem Kapitel wird auch der Simulationsablauf für den gesamten Entwicklungszyklus beschrieben.

Kapitel 7: Das eigentliche Ende der Arbeit bildet die Zusammenfassung, die neben dem erzielten Ergebnis auch Ausblicke auf Weiterführungen und Verbesserungsvorschläge geben soll.

Kapitel 2

Hardwarebeschreibung

2.1 ARCS RTS-DSP-Board

Das Board ist ein Teil des Kanalsimulators *smartsim* [4] der Firma ARCS und dient zur Simulation von Übertragungskanälen von Mobilfunksystemen.

Es setzt sich aus zwei eigenständigen Prozessoreinheiten zusammen, dem DSP-System und dem CPU-System. Beide verfügen über Anbindungen an SDRAM und Flash-Speicher. Die Verbindung zwischen den beiden Systemen stellt ein FPGA[1] her. Dieser dient zum Austausch von Daten und Steuersignalen zwischen den beiden Einheiten. Beide, das FPGA und der DSP [2], sind für Simulationsaufgaben vorgesehen. Die CPU [3] stellt die Kommunikation über Ethernet zum PC her.

Wie aus Abb. 2.1 ersichtlich ist, werden die beiden External Memory Interfaces (EMIFs) für verschieden Aufgaben genutzt. Das EMIFA wird zum Ansprechen des 32 MByte großen SDRAM verwendet. Wie aus dem Blockschaltbild hervorgeht, besteht außerdem die Möglichkeit ein SODIMM SDRAM Modul über das EMIFA anzusprechen. Es kann aber immer nur eines der vorhin genannten Speichermodule verwendet werden. Der Datenaustausch bzw. die Kommunikation mit dem FPGA und dem Flash-Speicher wird über das EMIFB durchgeführt.

Das FPGA erfüllt zwei verschiedene Aufgaben. Einerseits erledigt es einen wesentlichen Teil der Signalverarbeitung (Simulation des Kanalmodells) und andererseits findet es als Kommunikationsschnittstelle zwischen DSP und Coldfire Verwendung. Für diese Arbeit ist nur der Kommunikationsteil von Bedeutung. Dieser ist als Dual Ported RAM (DPRAM) ausgeführt und wird vom DSP für den Datenaustausch genutzt.

[1] Xilinx Virtex2 XC2V1000-E [11]
[2] Texas Instruments TMS320C6416 [10]
[3] Motorola Coldfire MCF5272 [1]

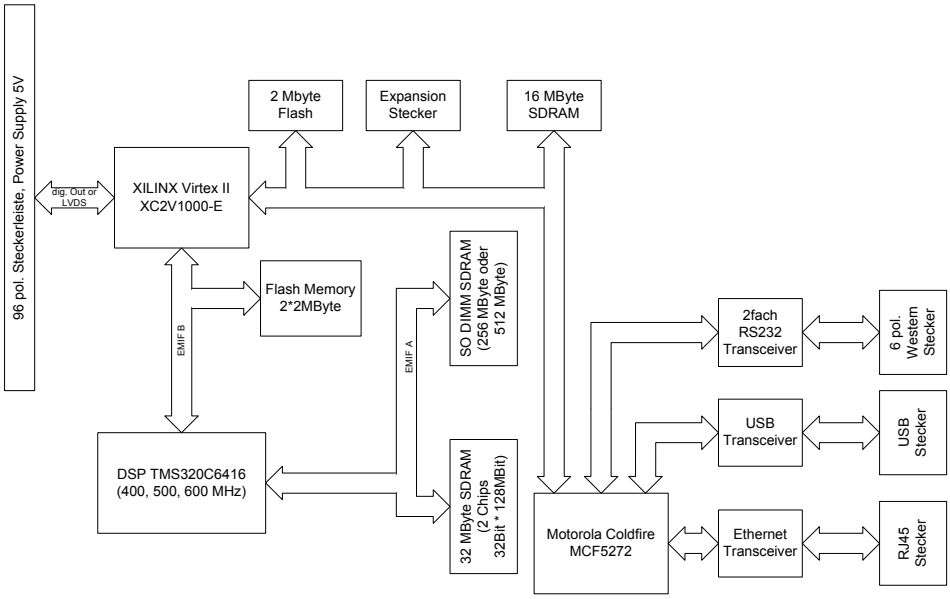


Abbildung 2.1: Blockschaltbild ARCS-RTS-Board.

Um Daten zum DSP zu transportieren, wird einerseits das FPGA benötigt und andererseits der Prozessor. Dieser stellt die Schnittstelle zur Außenwelt dar. Um diese Aufgabe erfüllen zu können, läuft auf ihm ein μCLinux, das speziell für dieses Board konfiguriert wurde. Für die Ethernetschnittstelle wird weiters noch ein Transceiver benötigt, der die physikalische Schicht des ISO-OSI Modells abbildet.

2.2 Sundance Carrier Board

Das SMT310Q [9] ist eine PCI Trägerkarte und bietet Platz für bis zu vier TIMs[4]. Diese Trägerkarte stellt die Verbindung zwischen dem PC und den Modulen dar. Weiters ermöglicht es die Kommunikation über die so genannten comports der Prozessormodule untereinander. Es bietet auch dem PC die Möglichkeit direkt auf die comports sowie den SDB[5] zuzugreifen. Die Kommunikation zwischen PC und Träger wird über eine PCI-Bridge durchgeführt. Über sie können DMA-Transfers, Mailbox- und Interruptzugriffe vom Host aus stattfinden. Auf der Karte erfolgt der Datenaustausch über den lokalen Bus. Der verfügbare Speicher beträgt 1 MByte shared SRAM und nur der Master TIM, auf Steckplatz eins, hat direkten Zugriff auf diesen.

[4]Texas Instruments Modules
[5]Sundance Digital Bus

2.3 Sundance TIM Modul SMT365

Das SMT 365-8-2 [8] ist ein DSP/FPGA TIM Modul der Firma Sundance. Als DSP kommt hier auch ein Texas Instruments DSP (TMS320C6416) zum Einsatz. Es bietet 8 MByte ZBTRAM[6] sowie 8 MBytes Flash ROM. Das ROM enthält boot Code und Daten zur Programmierung des FPGA. Über die sechs comports und die vier SDBs bzw. zwei SHBs[7] können Datentransfers durchgeführt werden. Diese Schnittstellen werden, wie aus Abb. 2.2 ersichtlich, über den FPGA angesprochen. Es handelt sich hier um einen Virtex2 XC2V2000E-4 der Firma Xilinx. Das Konfigurieren des FPGA übernimmt der DSP nach Einschalten der Spannungsversorgung. Der FPGA steuert weiters den globalen Bus und enthält, wie schon erwähnt, die comports und SDBs bzw. SHBs.

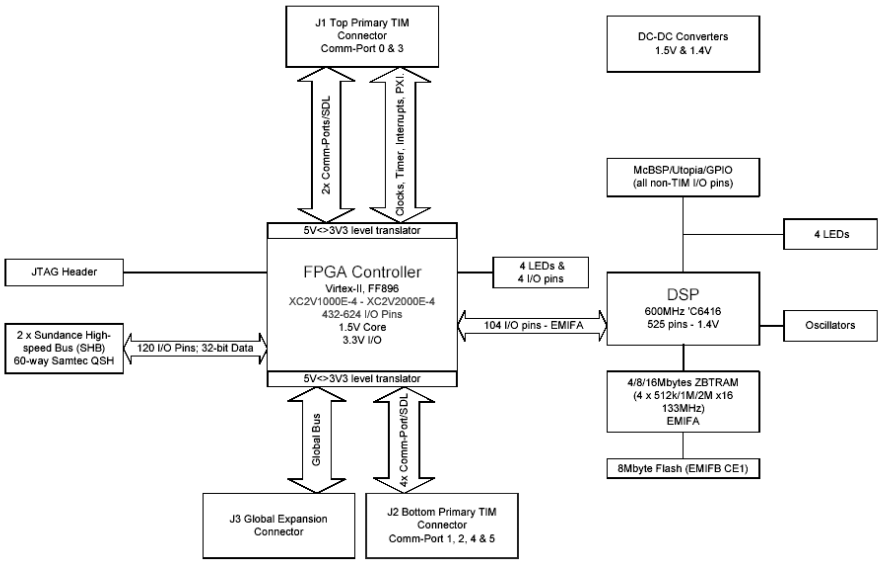


Abbildung 2.2: Blockschaltbild TIM SMT365-8-2.

[6] Zero Bus Turnaround RAM
[7] Sundance Highspeed Bus

Kapitel 3

Softwarebeschreibung

3.1 Matlab Schnittstellen

3.1.1 Matlab mex-Dateien

Um C-Routinen nutzen zu können, bietet Matlab die Möglichkeit, diese mittels mex-Dateien einzubinden. Diese werden häufig zur Steigerung der Effizienz, wie z. B. bei Schleifen, eingesetzt oder um bestehende Codes in Matlab zu integrieren.

Mathworks stellt für C ein *Application Program Interface*, kurz *API*, unter Matlab zur Verfügung. Mit dessen Hilfe können C-Routinen aus Matlab aufgerufen werden, als wären sie schon direkt in Matlab integriert. Dieses API gliedert sich in zwei Bibliotheken und zwar den *mx-Routinen* und den *mex-Routinen*. Mittels *mx-Routinen* können *mxArrays* erzeugt und verarbeitet werden und beinhaltet weiters Funktionen für das Speichermanagement. Der Zugriff auf Matlab-Funktionen, die Textausgabe und die Fehlerbehandlung, sind in Form von *mex-Routinen* in der *API* abgelegt.

Unter Microsoft Windows liegt eine mex-Datei als *Dynamik Link Library (DLL)* vor, und wird durch die Endung `.dll` gekennzeichnet. Es handelt sich hierbei um dynamisch gelinkte Routinen, die der Matlab Interpreter automatisch ausführen und laden kann. Diese Routinen müssen natürlich bestimmte Konventionen erfüllen, um von Matlab aufgerufen werden zu können. Der Aufruf unter Matlab unterscheidet sich nicht von m-Dateien, d. h. mit dem Namen der mex-Datei. Nachfolgend ist ein Programmrumpf mit den Hauptbestandteilen angeführt.

```
#include "mex.h"
void mexFunction( int           nlhs,
                  mxArray       *plhs[],
                  int           nrhs,
                  const mxArray *prhs[]
                  )
{
  /* C code ... */
}
```

Den Einstiegspunkt jeder mex-Datei bildet die *mexFunction*. Bei jedem Aufruf werden vier Parametersätze übergeben. Mit *nlhs* werden die Anzahl der erwarteten Ausgangswerte und mit *nrhs* die Anzahl der Eingangsparameter bezeichnet. Über *plhs* und *prhs* werden die eigentlichen Aus- und Eingangsparameter zwischen Matlab und der mex-Datei ausgetauscht. Unter Matlab werden alle Variablen als Matlab Arrays abgespeichert. Diese werden in *mxArrays* unter C abgebildet. Es enthält außer den Daten auch noch Informationen über deren Typ und Dimensionen.

Um eine C-Datei unter Matlab nutzen zu können muss sie, wie schon erwähnt, in eine mex-Datei übersetzt werden. In Abbildung 3.1 ist der Ablauf zum Erstellen einer DLL schematisch dargestellt. Weiters ist daraus ersichtlich, dass auch aus mehreren C-Dateien eine DLL erstellt werden kann. Hierbei ist nur zu beachten, dass die *mexFunction* nur in einer Datei enthalten ist. Wird unter Matlab das erste Mal eine mex-Datei erzeugt, sollte mittels des Befehls `mex -setup` der zu verwendende Compiler ausgewählt werden.

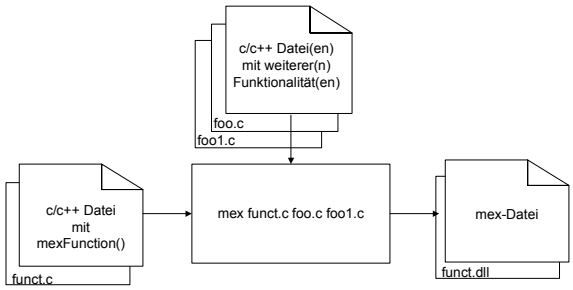


Abbildung 3.1: Erstellen einer Matlab mex-Datei unter Microsoft Windows.

Unter Microsoft Windows können mex-Dateien in einem Microsoft *VISUAL STUDIO 6.0* Projekt erstellt werden. Dies bietet die Möglichkeit, auf die gewohnten Entwicklungsfunktionalitäten dieses Tools, wie z. B. das Debuggen, zurückzugreifen.

3.1.2 Simulink S-Funktionen

System-Funktionen, oder kurz S-Funktionen, stellen eine Erweiterung zu den Standardblöcken unter Simulink dar. Mit ihrer Hilfe kann eigene Funktionalität einfach in die Simulation integriert werden. Die Beschreibung kann als Matlab m-Datei oder in Form von C, C++, Fortran oder Ada Codes erfolgen. Eine S-Funktion wird als mex-Datei kompiliert und bei Bedarf dynamisch in Matlab gelinkt. Eine S-Funktion kann aus den verschiedensten Gründen zum Einsatz kommen, wie z. B. Erstellen von nicht Standardfunktionalität, existierenden Code in Matlab einbinden und/oder um Schnittstellen zur Hardware zur realisieren.

Wie bei mex-Dateien sind auch bei S-Funktionen bestimmte Elemente erforderlich, um richtig ausgeführt werden zu können. Wichtig ist, dass der verwendete *Blockname* mit dem Dateinamen, ohne Endung, übereinstimmt. In Abbildung 3.2 ist ein grober Simulationsablauf dargestellt. Dieser kann in drei Abschnitte gegliedert werden. Der Erste, *Start und Initialisierung*, ist z. B. für die Initialisierung der Ein- und Ausgänge oder auch der Hardware verantwortlich. Einen wesentlichen Teil stellt die *mdlUpdate* Funktion dar, hier werden diskrete Zustände aktualisiert. Weiters ist sie für das Durchführen verschiedenster Aufgaben nützlich, die nur einmal pro Integrationsschritt stattfinden sollen. Danach folgt der eigentliche Algorithmus in der *simulation loop (mdlOutputs)*. Hier werden die Ausgangswerte und Folgezustände berechnet. Den Abschluss bildet der *Terminierungsteil*. Dieser wird, wie sein Name schon ahnen lässt, am Ende bzw. beim Verlassen der Simulation aufgerufen. Hier können allozierte Speicher wieder freigegeben oder eine bestehenden Verbindung zur Hardware geschlossen werden.

3.2 Kurzbeschreibung GenC

Das Programm *GenC* [7] bietet die Möglichkeit, einen Source Code, der in *Generic C* erstellt wurde, in Simulink, 3L_Diamond und A|RT-Builder Code umzusetzen. Mit diesem Programm ist es möglich, vier verschiedene C-Formate herzuleiten. Als Basis dient immer ein und der selbe Source Code. Die Umwandlung erfolgt ohne den ursprünglichen Code zu verändern, und weiters ist kein Anpassen an die jeweilige zu erstellende Umgebung notwendig. Der Aufruf sieht folgendermaßen aus:

```
GenC progname.gc SL
```

Durch den dritten Parameter wird die Zielplattform ausgewählt. Es werden folgende Parameter unterstüzt:

```
SL, SL2    Simulink
3L         3L_Diamond
AL, AL2    A|RT
RS         Echtzeitumgebung unter Simulink
```

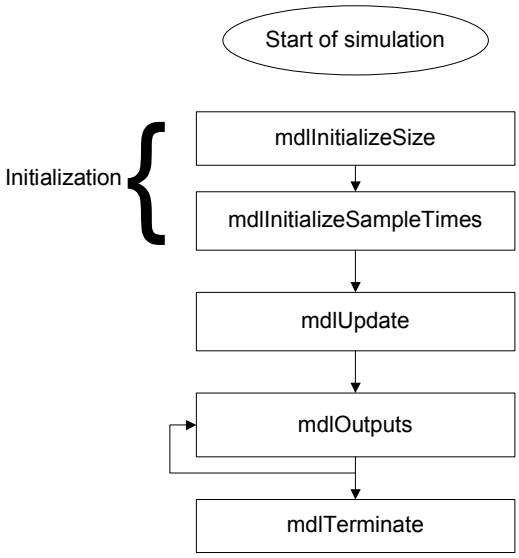


Abbildung 3.2: Einfacher Simulationsablauf einer Simulation unter Simulink mit einer S-Funktion.

Im Zuge der Diplomarbeit sind die beiden Parameter `SL` und `3L` am wichtigsten. Mittels dieser beiden Parameter werden aus der Datei `progname.gc` eine `progname.cpp` für Simulink und eine `progname.c` für den DSP. Diese drei Dateien bilden den Ausgangspunkt für das im nächsten Kapitel beschriebene Programm *GenCaddOn*.

3.2.1 Aufbau einer Generic C Datei

Im nachfolgenden Abschnitt sollen die wesentlichen Bestandteile einer `*.gc` Datei sowie deren wichtigsten Befehle und Eigenheiten näher beschrieben werden. Im Anhang C, Abschnitt C.1 ist ein Beispiel angeführt.

Eine *Generic C* Datei setzt sich aus drei Teilen zusammen. Diese sind der *Header*, die *Initialisierung* und der *Hauptteil*.

Der Header beinhaltet die Definitionen für *includes*, *defines*, *inputs*, *outputs* und deren *rates* und die Keywörter *STATE*, *RAM*, *DRAM* und *PARAMETER*. Compiler statements wie z. B. includes, defines werden ohne Änderung in Kleinschrift übersetzt.

Bis auf den Parameter *STATE*, können alle anderen Anweisungen öfter verwendet werden. Durch *GenC* wird nur eine vereinfachte Verwendung dieses Parameters unterstützt. Prinzipiell kann auch mittels *static* Variablen die selbe Funktionalität erreicht werden. Es gibt aber auch hier Einschränkungen in der Funktionalität. Ein Problem tritt bei Simulink auf, und zwar dann, wenn die S-Funktion öfter instanziert wird. Dies kann dadurch

vermieden werden indem man die Datei, die den *Generic C* Code enthält, unter verschiedenen Dateinamen speichert.

Nach den Headerdefinitionen folgt die Initialisierung, sie wird durch *init_progname* eingeleitet. Wobei `progname` der *Generic C* Datei ohne der Erweiterung `.gc` entspricht. Hier erfolgt die Initalisierung der *STATE* und *static* Variablen. Es ist zu beachten, dass sich die hier verwendeten temporären Variablennamen von jenen im Hauptprogramm unterscheiden. Weiters sollte vermieden werden, die Variable *first_run* weder hier noch im Hauptprogramm zu verwenden, da es sich bei dieser um eine interne Variable, die durch *GenC* erzeugt wird, handelt. Die Initialisierung muss mit einem Standarddatentyp beendet werden, um fehlerfrei übersetzt werden zu können.

```
init_progname()
{
  /* Initialisierung der STATE und static Variablen */
}
```

Den letzten Teil stellt das Hauptprogramm dar. Wobei `progname` wiederum der Dateiname der *Generic C* Datei ist. Die Abarbeitung erfolgt je nach erstelltem Code, sample-by-sample oder block-by-block.

```
progname()
{
  /* eigentliches Programm */
}
```

3.3 Sundance SMT6025

Wie schon in Abschnitt 2.3 erwähnt, kommt unter anderen ein DSP/FPGA Modul von Sundance zum Einsatz. Um dieses über den PC zu steuern bzw. Daten auszutauschen stellt die Firma Sundance ein *Software Development Kit*, das sogenannte SMT6025, zur Verfügung. Es sollen nun kurz die wichtigsten Funktionen erklärt werden.

Das SDK stellt die Schnittstelle zwischen der Hardware und dem eigenen Programm her. Die Funktionsweise ist in Abbildung 3.3 dargestellt. Es ist daraus ersichtlich, dass das eigene Programm nur über das SDK eine Verbindung zur Hardware herstellen kann. Der Vorteil, der dadurch entsteht, ist, dass das SDK die gesamte Zugriffssteuerung auf das Modul übernimmt. Der Entwickler kann so mehr Zeit für die Umsetzung seiner Anwendung aufwenden.

Die Funktionalitäten, die das SMT6025 zur Verfügung stellt, umfassen den Datentransfer, die Abfrage von Kontroll- und Allgemeininformationen und die Programmierung des Moduls. Es stellt weiters ein einfach zu benutzendes Interface dar, um auf *comports*, *mailboxes*, *HSC*[1] und auf

[1] High Speed Channel

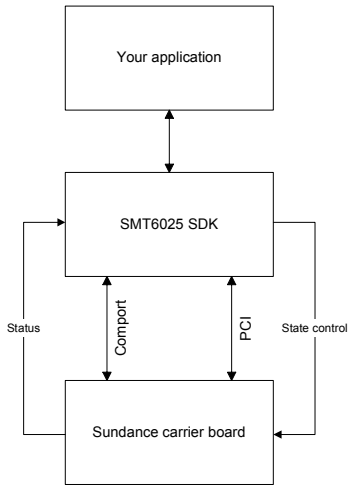


Abbildung 3.3: SMT6025 Schnittstelle zur Sundance Hardware.

DSP-Interrupts vom Host aus zuzugreifen. Das SDK stellt auch den Zugriff auf Low-Level Funktionen, wie z. B. Zugriff auf DSP-Register, zur Verfügung.

Der Transfer zwischen PC und Modul erfolgt über die PCI-Bridge des PC. Dieser kann auf zwei verschiedene Arten erfolgen. Die möglichen Transferraten sind in Tabelle 3.1, jeweils für Schreib- und Lesezugriffe, angeführt. Die langsamere Variante über den *Host comport* wird z. B. für die Programmierung des Modules verwendet und ist auch einfacher anzuwenden. Das SDK bietet die Möglichkeit auf den HSC auf zwei verschiedenen Arten zuzugreifen. Die Basis bildet bei beiden ein *HSC Handler Objekt*. Das Austauschen von Daten kann hier entweder über einen PCI- oder SRAM-Speicherbereich erfolgen.

Bei dieser Arbeit kamen beide Übertragungsarten zum Einsatz. Der Transfer über comports wurde zum Austausch von Statusmeldungen und der HSC von großen Datenmengen zwischen PC- und DSP-Programm verwendet.

Transfer Type	*Read [MB/s]*	*Write [MB/s]*
HSC with contiguous memory.	46	40
HSC with Scatter/Gather memory.	40	32
Host comport	2	2

Tabelle 3.1: Typische Datentransferraten bei unterschiedlichen Transfermöglichkeiten.

3.4 ARCS LANFlasher

Der Programm LANFlasher [5] ermöglicht es neue Codes für den DSP oder den FPGA in den Flash-Speicher des ARCS RTS-DSP-Board zu laden. Die Programmierung erfolgt über die Ethernetschnittstelle und nicht mehr über JTAG. Die Option dieser Art des Programmierens besteht nicht bei neuen Boards, sondern nur für jene, die schon mit einem speziellen Programm im Flash versehen worden sind. Dies bedeutet, dass mindestens einmal mit JTAG das Board geflasht werden muss. Es kann jedes Programm verwendet werden, es müssen nur folgende Änderungen vorgenommen werden: Auf der PC Seite ist das Programm `LANFlasher.exe` und eine `*.out` Datei notwendig. Es müssen auch einige neue Module in das DSP-Programm eingebaut werden und beim Empfang einer Nachricht auf die jeweiligen Handler verzweigt werden, wie nachfolgender Code zeigt:

```
switch (VarType){
  case SMRT_MEMCOPY_ID:
  case SMRT_MEMSET_ID:
    RecvBinaryData(VarType);
    break;
  case SMRT_COMMAND_ID:
    RecDataTFromNet(SMRT_COMMAND_ID, (int*) &Cmd);
  switch( Cmd.commandcode ){
    case SMRT_COMMAND_FLASH:
      ProgramFlash((Byte*) Cmd.data[2],(Byte*) Cmd.data[0],
                   Cmd.data[1], Cmd.data[3]);
      break;
    // andere Kommandos
  }
  // andere Datentypen
}
```

Durch diese Erweiterung wird die Programmierung über JTAG überflüssig. Dadurch kann der Entwicklungszyklus beschleunigt werden, da das TI CCS[2] überflüssig wird. Es wird nur noch die Möglichkeit der Kompilierung über die Commando Zeile genutzt. Dies kann auch direkt aus Matlab erfolgen. Um dies durchführen zu können, müssen noch die Umgebungsvariablen gesetzt werden. Dies kann mittels *Batch-Datei* `Dosrun.bat` im TI-Installationsverzeichnis oder händisch erfolgen.

[2]Texas Instruments Code Composer Studio

Kapitel 4

GenCaddOn: Erweiterung zu GenC

4.1 Warum GenCaddOn?

Wie in Abschnitt 3.2 schon erwähnt bietet das Programm *GenC* die Möglichkeit der Umwandlung von *Generic C* Dateien in Simulink S-Funktionen und DSP-Code. Diese erzeugten Dateien können aber nur unter ihren Zielumgebungen ausgeführt werden. Um unter Matlab und Simulink auf die in Kapitel 2 beschriebenen Hardwaresysteme zuzugreifen müssen die von *GenC* erzeugten Dateien um Initialisierungs-, Sende- und Empfangsroutinen erweitert werden. Dies erlaubt es, einzelne Simulationsblöcke auf die Hardware auszulagern und unter realen Bedingungen zu simulieren bzw. zu testen. Es kann hier also für die Simulation und für Funktionstests die gleiche Testumgebung verwendet werden.

Um diese zusätzliche Funktionalität zu unterstützen standen zwei verschiedene Ansätze zur Auswahl. Es kann das Programm *GenC* erweitert werden oder ein neues Programm, das diese Erweiterung ermöglicht, entwickelt werden. Im ersten Moment mag es sinnvoller erscheinen, einfach die noch benötigten Routinen *GenC* anzufügen. Da es sich aber um keinen eigenen Source Code handelt ist eine genau Analyse erforderlich, um durch die Erweiterungen die ursprüngliche Funktionalität nicht zu verfälschen. Bei einem eigenen Programm ist es nur wichtig, die Inputs, in diesem Fall die von *GenC* erzeugten Dateien, zu kennen und weniger wichtig ist, wie diese generiert werden. Nicht vergessen werden dürfen auch die Tests. Bei einem erweiterten Code sollte auch nach den Änderungen gewährleistet sein, dass dieser noch seine alte Funktionalität erfüllt. Dies macht es notwendig, neben den neuen Funktionen, auch noch die bestehenden Funktionen zu testen. Bei der zweiten Variante ist dies nicht der Fall, da nichts am ursprünglichen Programm verändert wurde. Diese Zeitersparnis kann für ein intensiveres Testen der eigenen Funktionalität verwendet werden.

Der Nachteil ist allerdings die Abhängigkeit von *GenC*. Eine Änderung in *GenC* kann u. a. zu einer Fehlfunktion im eigenen Programm führen. Dieses Problem besteht in gewisser Weise auch beim Hinzufügen von Codes, da hier auch die neuen Routinen erneuert werden müssen.

4.2 Funktionsweise

Das Programm *GenCaddOn* ist ein Konverter, der die von *GenC* erzeugten Dateien, zu hardwarespezifischen Dateien umwandelt. Wie aus Kapitel 2 hervorgeht erfolgt die Umwandlung auf zwei verschiedene Hardwareplattformen. Der Programmaufruf von der Kommandozeile aus sieht wie folgt aus:

```
GenCaddOn plattform progname -p progname1.mdl
```

Mittels des Parameters `plattform` wird das Zielsystem ausgewählt. Gültige Werte sind *SMT* für die Sundanceplattform und *RTS* für die Hardware von Seibersdorf. Der dritte Parameter gibt den Namen der *Generic C* Datei an, aber ohne der Dateiendung `.gc`. Mit dem Parameter `-p` (optional) kann eingestellt werden, ob Projektdateien für den DSP erstellt werden sollen. Der letzte Parameter gibt eine Datei an, die ein Simulink-Modell enthält, und ist optional. Der Hardwareblock wird in ein bestehendes Modell eingefügt.

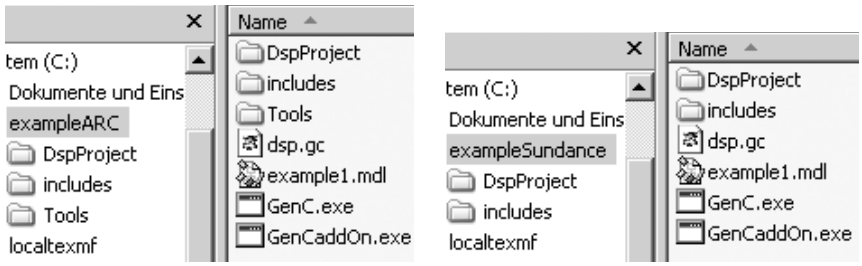
4.2.1 Warum neue Dateien?

Es stellt sich auch die Frage warum neue Dateien erstellt werden und die Änderungen nicht in den bereits bestehenden Dateien gemacht werden.

Die Modifikationen in den bestehenden Dateien durchzuführen ist programmiertechnisch aufwendiger, als nur die benötigten Code Zeilen zu übernehmen. Weiters kann dadurch, je nach Bedarf, die Simulation am PC oder auf der Hardware durchgeführt werden, ohne jedes Mal die Dateien neu zu erstellen. Außerdem handelt es sich nur um zwei zusätzliche Dateien. Aus diesen Gründen fiel die Entscheidung, dass es sinnvoller ist, diese zu erstellen.

4.2.2 Vorraussetzungen

Wie schon erwähnt werden die von *GenC* erzeugten Dateien eingelesen und daraus neue plattformabhängige Dateien erzeugt. Dafür sind einige Vorraussetzungen notwendig um ein fehlerfreies Konvertieren zu gewährleisten. Dazu benötigt *GenCaddOn* drei Dateien, und zwar `progname.gc`, `progname.c` und `progname.cpp`. Die letzten beiden Dateien können mittels *GenC* und den Parametern `3L` bzw. `SL` generiert werden. Aus diesen Dateien ermittelt dann das Programm die benötigten Informationen, um die neuen Dateien zu erstellen. Für eine fehlerfreie Umsetzung wird außerdem eine



(a) (b)

Abbildung 4.1: Exemplarische Verzeichnisstruktur für (a) ARCS und (b) Sundance.

gewisse Verzeichnisstruktur benötigt bzw. sollte diese eingehalten werden. Diese beiden Strukturen sind in Abbildung 4.1 dargestellt. Dies soll eine gewisse Übersichtlichkeit bieten, da hier die Dateien je nach Art der Verwendung, in Unterverzeichnissen gespeichert werden. Ein weiterer Grund für diese Aufteilung ist die Includestruktur und projektspezifische Dateien. Unter projektspezifische Dateien sind jene zu verstehen, die nicht immer wieder neu erstellt werden müssen, oder von den Hardwareherstellern schon zur Verfügung gestellt werden. Im Anhang A sind die Verzeichnisstrukturen dargestellt. Weiters sind benötigte Programme und Dateien darin festgehalten. Man kann hier erkennen, dass viele Dateien notwendig sind um die Hardwarefunktionalität einzubinden. Diese Auflistung soll nochmals verdeutlichen, warum eine Aufteilung in Verzeichnisse durchgeführt wird.

4.3 Umwandlungsablauf

In diesem Abschnitt wird der grundlegende Umwandlungsalgorithmus beschrieben. Aus Gründen der Übersicht wird hier auf einige Details, vor allem das Umwandlungsergebnis betreffend, nicht näher eingegangen, denn diese werden im folgenden Abschnitt ohnehin genauer betrachtet.

Die Fehlererkennung wird größtenteils beim Öffnen und Anlegen von neuen oder beim Öffnen von bestehenden Dateien angewendet. Der Grund dafür liegt darin, dass die benötigten Eingangsdateien durch *GenC* erstellt wurden und deshalb von weitgehend fehlerfreien Dateien ausgegangen wird.

Zu Beginn wird die Anzahl der übergebenen Parameter überprüft, da diese entweder drei, vier oder fünf betragen kann. Anschließend erfolgt das Einlesen des zweiten Parameters und dessen Auswertung. Je nach angegebener Zielhardware wird ein Flag auf einen bestimmten Wert gesetzt, der der jeweiligen Hardware zugeordnet ist. Den nachfolgenden Routinen wird nur mehr dieses Flag übergeben, um zwischen den beiden Hardwaresystemen zu

unterscheiden. Weiters werden die benötigten Dateinamen erzeugt, die sich von dem dritten Parameter, dem Projektnamen, ableiten. Die Hauptdateien für den DSP erhalten entweder die Bezeichnung `ProjektnamenRTS.c` oder `ProjektnamenSMT.c`. Die Simulink S-Funktion erhält den Dateinamen `ProjektnameIO.cpp`.

Nach dem Einlesen und Überprüfen der Eingangsparameter wird nach Ein- und Ausgängen gesucht. Das Einlesen dieser Parameter wird hauptsächlich für das RTS-Board benötigt und in Abschnitt 4.4.1 erklärt. Die Suche wird in der `progname.gc` Datei durchgeführt. Der Grund hierfür liegt darin, dass in dieser Datei die Suche am einfachsten durchgeführt werden kann. Dies liegt an den Keywörtern *INPUT_PORT* und *OUTPUT_PORT*. Wird eines dieser beiden gefunden, werden die benötigten Informationen ausgelesen und in ein Array vom Typ *DataStorage* abgelegt.

```
typedef struct {
 char datatyp[30];
 char width[30];
 char ins[30][30];
 char outs[30][30];
 bool isStruct;
 int structNr;
} DataStorage;
```

Zuerst wird der Datentyp eingelesen und überprüft, ob es sich um einen Standardtyp handelt. Handelt es sich nicht um einen Standardtyp, wird die *isStruct* Variable auf *true* gesetzt und der Wert von *structNr* um eins erhöht. Danach wird die nächste Zeile eingelesen um daraus die Größe des Ein- oder Ausganges zu ermitteln. Diese wird in der Variable *width* gespeichert. Wie aus der Datei `dsp.gc` im Abschnitt C.1 ersichtlich ist, folgt *INPUT_PORT* und *OUTPUT_PORT* in der darauf folgenden Zeile die zugehörige *RATE*. In den Variablen *ins* und *outs* wird gespeichert, welche Ein- und Ausgänge den gleichen Datentyp und die Größe aufweisen. Die Suche wird beendet, wenn in einer Zeile *init* gefunden wird oder das Ende der Datei erreicht ist.

Wie in Abbildung 4.2 ersichtlich ist spaltet sich hier der weitere Programmverlauf je nach verwendeter Hardware auf. Zunächst wird mit der weiteren Vorgehensweise der Umsetzung für das RTS-Board fortgesetzt.

Nachdem die Ein- und Ausgänge erfasst wurden, wird nach allen *defines* gesucht und werden diese in einem Array mit maximal 100 Einträgen gespeichert. Im Beispiel im Abschnitt C.1 ist ersichtlich, dass sich eine define Anweisung (in Zeile 1) direkt auf die Größe der Ein- und Ausgänge bezieht. Wie schon erwähnt werden die Ein- und Ausgänge in Strukturen umgewandelt, die sich in `globalTypes.h` befinden. Da diese Datei sowieso inkludiert werden muss, werden alle *defines* in diese Datei kopiert.

Vorweg muss hier erwähnt werden, dass das Programm *GenCaddOn* nur unter Microsoft Betriebssystemen kompiliert und getestet wurde. Theoretisch ist das Programm aber auch unter anderen Betriebssystemen, wie z. B.

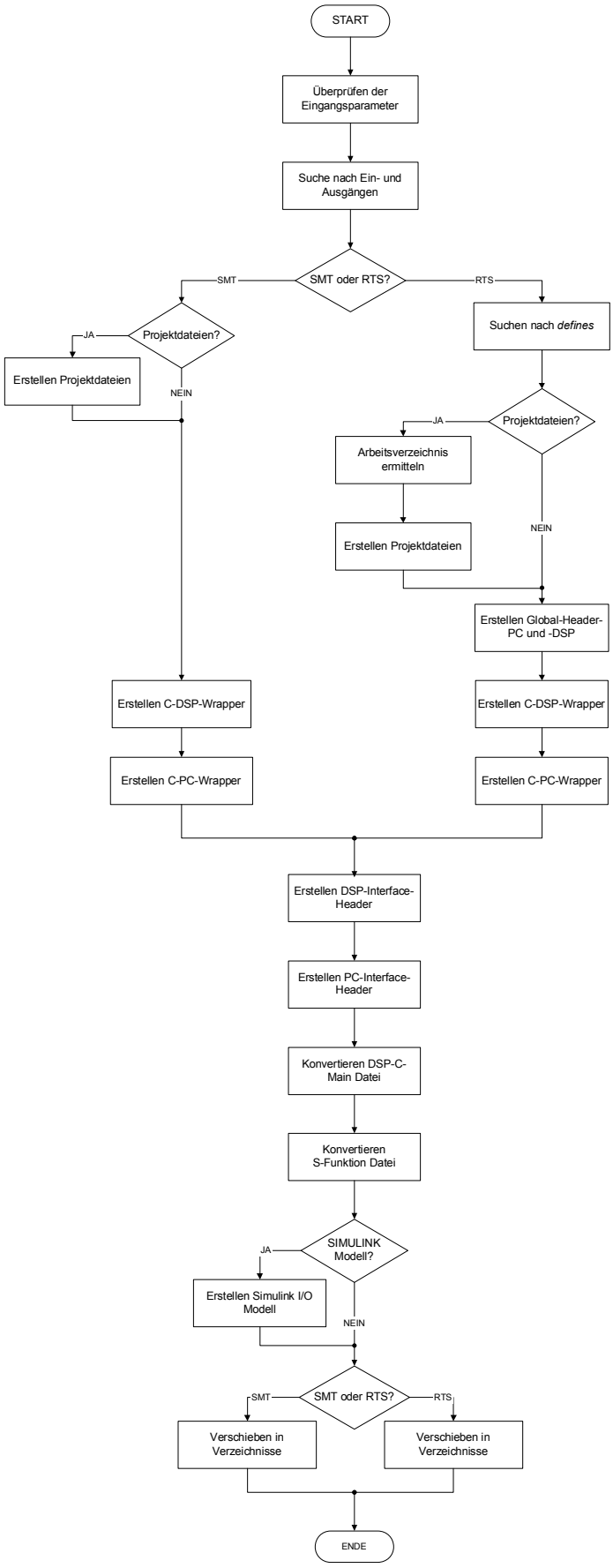


Abbildung 4.2: Umwandlungsablauf von GenCaddOn.

Linux oder Unix, ablauffähig. Es muss nur unter dem jeweiligen Betriebssystem kompiliert werden.

Als letzter Parameter, vor den eigentlichen Umwandlungen, wird das aktuelle Arbeitsverzeichnis ermittelt. Dies wird für das Erstellen der Projektdatei benötigt. Die einfachste Möglichkeit wäre es, den ersten Parameter der übergeben wird zu nutzen. Dies ist aber nicht möglich, da je nach Art des Aufrufes nicht immer der gesamte Pfad als Programmargument übergeben wird. Unter Art des Aufrufes wird Verstanden, wie ein Programm aufgerufen wird, wie z. B. Kommando Zeile, Matlab oder Explorer. Deshalb wird der Umweg über den Befehl

```
system("cd > dir.txt");
```

unter C in Kauf genommen. Mit Hilfe dieses Befehles ist es möglich Systembefehle auszuführen. In diesem Fall wird der `cd` (change directory) Befehl verwendet, der mit diesen Parametern das aktuelle Arbeitsverzeichnis in die Datei `dir.txt` schreibt. Das Schreiben in eine Datei ist deshalb notwendig, da der Befehl nur eine Statusmeldung zurückliefert, nicht aber den Pfad. Aus dieser Datei wird nun das Verzeichnis ausgelesen und intern gespeichert.

Das aktuelle Arbeitsverzeichnis wird zur Erstellung der Projektdatei für das Code Composer Studio von Texas Instruments benötigt. Als Vorlage wurde ein bestehendes Projekt verwendet und es werden nur das Verzeichnis sowie der Name der DSP Datei eingefügt.

Fortgesetzt wird mit dem Erstellen der Globalen Header Dateien für PC und DSP. In diese werden alle gefunden *defines* und Datenstrukturen geschrieben.

Datenstruktur: Nachfolgend ist ein Beispiel für eine Datenstruktur, wie sie für die ARCS Hardware verwendet wird, angeführt.

```
typedef struct {
  unsigned int var_id;
  int data[2*M];
} MATLAB_DATA1;

#define MAKE_ID( type, var)
        ((((unsigned int)(type))<<24)+sizeof(var)/4)

#define MATLAB_DATA1_ID MAKE_ID( 50, MATLAB_DATA1 )
```

Beim RTS-Board erfolgt der Datentransfer wie in Abbildung 4.3 dargestellt. Die Übermittlung der Daten, über Ethernet, erfolgt mittels UDP Protokoll. Wie in der Abbildung ersichtlich ist gelangen die Daten nicht direkt vom PC an den DSP und umgekehrt, sondern über die CPU und das Dual Ported RAM (DPRAM). Deshalb ist es nicht möglich, die Daten alleine zu versenden, da z. B. die CPU nicht erkennen würde, wie viele Bytes sie weiterleiten müsste. Deshalb werden die Daten in eine Struktur verpackt.

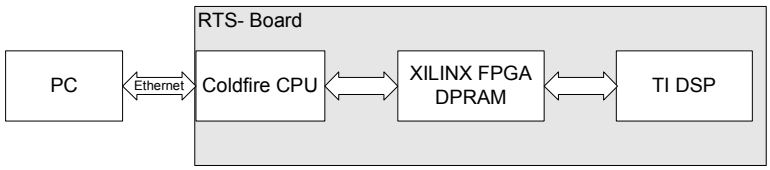


Abbildung 4.3: Schematische Dartstellung des Datentransfers zwischen PC und ARCS-RTS-DSP-Board.

Wobei der erste Wert, vom Typ unsigned int, Informationen über Länge und Typ zu dieser Struktur enthält. Diese Information werden über das Makro *MAKE_ID* erstellt. Der erste Parameter gibt den Typ an und sollte einmalig sein. *GenCaddOn* vergibt diesen aufsteigend, beginnend bei 50. Dieser Wert wurde gewählt, da bereits in `global.h` Strukturen definiert sind, und deren Typ bei 0 beginnt. Als zweiter Parameter wird die Variable übergeben. In der Struktur folgen nach der Variable *var_id* die eigentlichen Daten.

Wie in Abbildung 4.2 ersichtlich ist werden danach die Wrapper C-Dateien für den DSP und PC erstellt. Es werden hier für jeden Ein- und Ausgang, die sich in dem *inout* Array befinden, die Sende- bzw. Empfangsroutinen erstellt.

Ist die Umwandlung für die Sundance Hardware durchzuführen, werden zwei Dateien, die Konfigurationsdatei und das `MAKEFILE` für das DSP-Projekt erstellt. Um diese erstellen zu können wird nur der `progname` benötigt. Auch für diese Hardware werden Wrapper Dateien benötigt. Diese werden nach dem selben Muster wie für die ARCS Hardware erstellt.

Die nachfolgenden Abläufe werden wieder auf beide Plattformen angewendet. Dies bedeutet nicht, dass hier immer die gleichen Änderungen gemacht werden. Die Unterschiede können hier mittels einfachen Abfragen und wenig Code-Änderung implementiert werden.

Um die vorhin erwähnten Dateien nutzen zu können, müssen noch die entsprechenden Header Dateien erstellt werden. Diese enthalten die jeweiligen Funktionsprototypen.

Danach erfolgt das Anpassen der DSP main Datei. Für die Hardware von Sundance muss nur eine zusätzliche *include* Datei und der Aufruf der Initialisierungsroutine eingefügt werden. Diese Routine wird vor der Abfrage `if (first_run==0)` eingefügt, da an dieser Stelle die Variablendefinition beendet ist und noch keine Ein- und Ausgangsoperationen durchgeführt werden.

In der Hauptdatei des RTS-Boards müssen mehrere Veränderungen durchgeführt werden. Zu Beginn wird hier nach *define* Anweisungen gesucht, da diese nicht in die neue Datei übernommen werden. Weiters wird zu

Beginn noch eine *include* Anweisung hinzugefügt und die Kennzeichnung des Beginns der Hauptfunktion auf `void main(void)` geändert. Danach erfolgt die Neudekleration von Ein- und Ausgängen sowie deren temporärer Werte. Die Definitionen, die von *GenC* erzeugt wurden, werden nicht in die neue Datei übernommen. Weiters wird noch eine Variable, *size* vom Typ integer, eingefügt. Der Grund hierfür wird in Abschnitt 4.4.1 angeführt.

Weiters wird noch, vor der Abfrage `if (first_run==0)`, die Initialisierungsroutine *Init()* eingefügt. Der nachfolgende Code, der eigentliche Algorithmus, wird unverändert in die neue Datei übernommen.

Als letzte Umwandlung erfolgt die der S-Funktion. Hier sind unabhängig von der Hardware die größten Änderungen nötig. Es wird auch hier eine neue Datei erstellt und unveränderte Codezeilen kopiert. Zuerst muss der Blockname mit der Erweiterung *IO* versehen werden. Es wird hier nach der entsprechenden *define* Anweisung gesucht und anschließend manipuliert. Der nächste Schritt ist es nach den *include* Anweisungen zu suchen und je nach Hardware die benötigten einzufügen. Weiters wird eine static Variable *isInit* vom Typ boolean eingefügt. Diese soll später Auskunft geben ob die Hardware schon initialisiert wurde oder nicht.

Wie bei der DSP Hauptdatei des RTS-Boards ist es auch hier notwendig die *define* Anweisungen nicht in die neue Datei zu übernehmen.

Der Aufruf der jeweiligen Initialisierungsroutine erfolgt in der Funktion *mdlInitializeConditions*. Der Aufruf erfolgt vor der Zuweisung an den *xdisc* Parameter. Hier erfolgt auch die Abfrage, ob die Hardware schon initialisiert ist. Im Falle des RTS-Boards ist diese aber auskommentiert und kann bei Bedarf einkommentiert werden. Der Grund für das Auskommentieren wird in Abschnitt 4.4.1 erklärt.

Der eigentliche Algorithmus befindet sich in der Funktion *mdlOutputs* und deshalb sind auch hier die meisten Änderungen notwendig. Am Beginn der Funktion werden temporäre Eingangsvariablen deklariert, in diesen werden die empfangenen Werte zwischengespeichert. Weiters wird eine Zählervariable definiert, die später dazu verwendet wird die Ausgänge zuzuweisen. Dort wo sich der eigentliche Algorithmus befindet werden in der neuen Datei die benötigten Aufrufe für die Sende- und Empfangsroutinen eingefügt, sowie die Zuweisung der temporären auf die eigentlichen Ausgänge. Danach muss nur noch in die *mdlTerminate* Funktion das Trennen der Verbindung eingefügt werden. Dies geschieht nur bei S-Funktionen für das RTS-Board. Der restliche Code wird eins zu eins in die neue Datei übernommen.

Wie schon erwähnt, unterstützt *GenCaddOn* die Möglichkeit auch eine Simulink Modell Datei umzuwandeln. Wird dieser Parameter übergeben, wird eine neue Datei erzeugt und als `ModellnamenIO.mdl` gespeichert. Im Großen und Ganzen kann die Information beibehalten werden und eins zu eins in die neue Datei übernommen werden. Die Änderungen betreffen den Namen des Modells und des Blockes. Diese beiden werden in der gesamten Datei mit der Endung *IO* versehen. Durch diese Möglichkeit kann in einem

Matlab Script, ohne Ändern des Modells unter Simulink, eine Simulation auf den PC und der Hardware miteinander verglichen werden.

Zum Abschluss der Umsetzung werden die neu erzeugten Dateien noch in die entsprechenden Verzeichnisse verschoben. Die Gründe dafür wurden schon in Abschnitt 4.2.2 angeführt. Das Verschieben erfolgt mittels des Befehles `system("move name.ext dir")`. Als Name wird der jeweilige Dateiname übergeben, `ext` stellt die Erweiterung `c`,`h` oder `cpp` und `dir` das Verzeichnis dar.

4.4 Erstellte Dateien und deren Besonderheiten

Das Ziel von *GenCaddOn* ist es, dem Benutzer so viel Arbeit wie möglich abzunehmen bzw. diese zu automatisieren. In diesem Abschnitt werden die erstellten Dateien näher betrachtet. Es sollen hier vor allem die Gründe warum diese Dateien erzeugt werden und welche Aufgabe sie erfüllen geklärt werden. Im Anhang C ist jeweils ein Beispiel für die jeweilige Hardware angeführt. Als Basis für die Umwandlung dient die Datei `dsp.gc` und alle Ergebnisse beziehen sich auf diese Datei.

4.4.1 Ergebnis ARCS

dspRTS.c: *GenC* erzeugt aus den Ein- und Ausgängen temporäre Character Arrays und Zeiger des angegebenen Datentyps. Diese Zeiger werden mit der Anfangsadresse der Arrays initialisiert.

Es wurde bei Tests festgestellt, dass es bei größeren Arrays zu Überschneidungen bei diesen kommt. Die Gesamtgröße der Arrays bei den Tests betrug maximal 36000 Bytes. Um diesen Fehlern vorzubeugen werden nur mehr temporäre Character Zeiger definiert und die Daten werden in den externen 32 MByte großen SDRAM abgelegt. Die Speicherbelegung ist in Abbildung 4.4 dargestellt.

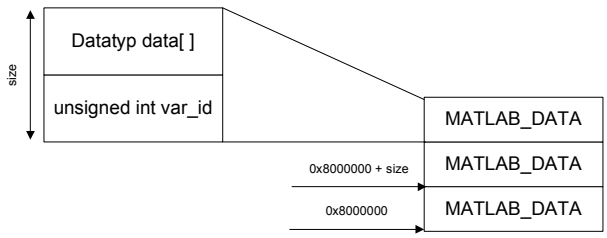


Abbildung 4.4: Speicherbelegung des externen Speichers bei der ARCS Hardware.

Die Initialisierung der Zeiger erfolgt nachdem alle benötigten Variablen definiert wurden. Es wird weiters am Ende der Variablendefinition eine Va-

riable mit der Bezeichnung *size* hinzugefügt. Sie gibt den Offset für den Speicherplatz an. Der Initialisierungswert hängt davon ab ob es sich um einen Standarddatentyp oder eine eigene Struktur handelt. Bei Standardtypen wird sie mit dem Wert *4* initialisiert. Dieser Wert kommt daher, dass jeder Standardtyp, wie in Abschnitt 4.3 beschrieben, in eine Struktur umgewandelt wird und die ersten *vier* Bytes den Variablenidentifier beinhalten. Handelt es sich um einen eigene Datentyp, wird *size* mit Null initialisiert.

Dem ersten temporäre Zeiger wird die erste Speicheradresse, die sich an der Stelle *0x8000000+size* befindet, zugewiesen. Danach wird *size* um die Größe der Datenstruktur erhöht. Handelt es sich hier wieder um einen bekannten Datentyp, wird *size* um weitere vier Bytes erhöht.

Nachdem alle Ein- und Ausgänge mit ihren temporären Variablen verknüpft wurden, wird die Initialisierungsroutine aufgerufen. Diese Routine befindet sich in `DspPcIO.c` und wird später noch erklärt. Dasselbe gilt für die Funktion *LookVar()*, die am Beginn der Endlosschleife aufgerufen wird.

DspPcIO.c und DspPcIO.h: Wie vorhin erwähnt, wird eine Initialierungsroutine benötigt. Diese Routine befindet sich in der Wrapper-Datei. Hier wird die Hardware für den Betrieb vorbereitet. Dies beinhaltet das Resetieren des FPGA's und Initialisieren des Cycle Buffer's.

Diese Wrapper-Datei beinhaltet weiters die Routine *LookVar()*. Sie wird benötigt um die Funktion des Programmierens über Ethernet zu ermöglichen. Dafür ist es notwendig, den Empfangsbuffer zu pollen bis Daten empfangen werden. Je nach empfangenen Daten, insbesondere der Typ ist wichtig, wird eine bestimmte Aktion durchgeführt. Mit dem eigentlichen Programm wird fortgefahren, wenn dem Variablentyp keine explizite Funktion zugeordnet ist.

Im Hauptprogramm werden die Variablen mittels *call_VariablenName* gesendet bzw. empfangen. Im Wrapper existiert für jede dieser *call* Funktionen eine eigene Routine. Diese rufen im Wesentlichen die Funktionen

- SendDataToNet: Sendet Daten an den Cyclebuffer.
- RecDataTFromNet: Auslesen einer bestimmten Datenstruktur aus dem Buffer.
- RecDataFromNet: Auslesen aller vorhandenen Daten aus dem Cyclebuffer.

auf. Den *call* Funktionen muss die Größe der Daten in Bytes und der Zeiger auf die Speicherstelle übergeben werden.

Beim Empfangen wird zunächst ein Zeiger der jeweiligen Datenstruktur angelegt. Diesem wird der Eingangssdatenzeiger, um vier verringert, zugewiesen. Diese Reduktion muss durchgeführt werden, da der Eingangszeiger nur auf die Nutzdaten zeigt. Empfangen wird aber die Datenstruktur mit *var_id*. Der Funktion *RecDataTFromNet* wird der Identifier der Datenstruktur und der temporäre Zeiger auf die Datenstruktur übergeben. Es wird

solange in der Funktion verblieben bis die Daten empfangen werden, die den richtigen Identifier aufweisen.

Das Senden erfolgt nach ähnlichem Prinzip. Auch hier werden die Größe und der Zeiger auf die Daten übergeben. In der Funktion wird wiederum ein temporärer Zeiger auf eine Datenstruktur angelegt. Diesem wird die Adresse des Eingangszeigers, um vier verringert, zugewiesen. Bevor die Daten gesendet werden wird die entsprechende *var_id* gesetzt. Die Daten werden durch die Funktion *SendDataToNet* an den CycleBuffer weitergeleitet. Nachdem dies erfolgt wird je nach Rückmeldung der Buffer gelöscht.

In der zugehörigen Headerdatei sind neben den Prototypen, der eben beschrieben Funktionen, die benötigten Includedateien aufgelistet. Weiters enthält sie die Definition der Zeiger auf den Cyclebuffer und Konstanten für den DSP Zustand.

globalTypesDsp.h: Hier befinden sich die Definition der verwendeten Datenstrukturen. Weiters werden hier die *defines* und Makros bereitgestellt.

dspIO.cpp: Nicht nur auf Seite des DSP's, sondern auch PC-seitig sind Änderungen und Erweiterungen in den von *GenC* erzeugten Dateien notwendig. In der S-Funktion Datei muss z. B. der Zugriff auf die Hardware realisiert werden. Die Initialisierungs-, Sende- und Empfangsroutinen sind wieder in einer Wrapperdatei zusammengefasst. Dies wird später noch genauer erklärt.

Die erste Änderung bezieht sich auf den *S_FUNCTION_NAME*, der um *IO* erweitert wird. Diese Erweiterung soll andeuten, dass es sich hier um einen Simulationsblock mit Hardwarezugriffen handelt. Weiters werden die nötigen Includedateien angeführt, um auf die Hardware zugreifen zu können.

Die Initialisierungsroutine wird in der *Funktion mdlInitializeConditions* aufgerufen. Sollte die Verbindung nicht hergestellt werden können, wird hier die Ausführung abgebrochen.

Wird die Simulation am PC durchgeführt erfolgt die Abarbeitung des Algorithmus sowie Einlesen und Ausgabe der Daten in *mdlOutputs*. Diese Routine wird dahingehend geändert, dass der ursprüngliche Simulationsalgorithmus durch die benötigten Sende- und Empfangsroutinenaufrufe ersetzt wird.

Am Begin der Funktion werden temporäre Ausgangsvariablen sowie eine Zählervariable definiert. Die nachfolgenden Deklarationen und Zuweisungen sind direkt aus der Originaldatei übernommen. An die Stelle des ursprünglichen Algorithmus werden die benötigten Sende- und Empfangsroutinenaufrufe eingefügt. Vor jedem dieser Aufrufe wird eine 20 μs lange Wartezeit[1]

[1] Bei Tests wurde bei schnellen Sende- und/oder Empfangsabläufen Fehler in der Übertragung festgestellt. Um diese zu eliminieren wurde die Wartezeit eingeführt.

eingefügt. Am Ende werden noch den Ausgangsvariablen die temporären Werte zugewiesen. Hierfür wird die vorher erwähnte Zählervariable benötigt.

Am Ende der Simulation wird die Funktion *mdlTerminate* ausgeführt. In dieser Funktion wird die bestehende Verbindung zur Hardware getrennt.

Soll die Simulation öfters, wie z. B. in einer Schleife, ausgeführt werden, wird bei jedem Simulationsdurchlauf die Verbindung neu hergestellt. Dies verlangsamt den Simulationsablauf und senkt die Datenübertragungsgeschwindigkeit.

Es wurde eine Möglichkeit vorgesehen die Initialisierungsroutine nur einmal aufzurufen. Hierfür muss die Variable *isInit* am Beginn der Datei einkommentiert werden. Weiters muss auch die Abfrage auf diese Variable in der Funktion *mdlInitializeConditions* einkommentiert werden. Damit die Verbindung bestehen bleibt muss noch der Aufruf der *Close* Routine in *mdlTerminate* auskommentiert werden. Somit wird die Verbindung nur einmal hergestellt und bleibt die gesamte Simulationsdauer erhalten.

Der Nachteil der sich hier ergibt sind die verwendeten Ressourcen, die die Kommunikation verwendet. Diese werden beim Beenden der Verbindung normalerweise wieder freigegeben. Da dies aber nun auskommentiert ist, können die Ressourcen nur durch Beenden von Matlab wieder freigegeben werden. Dieses Problem kann nur beseitigt werden, wenn die Anzahl der Simulationsdurchläufe bekannt ist. Dies Erfolgt durch Einfügen von nachfolgendem Code in *mdlTerminate*.

```
static int cntTerminate = 0;

if(cntTerminate == DurchlaufAnzahl) {
  if( Close() != 0 ) {
 mexErrMsgTxt("\nconnetion could not be closed!");
 }
} else
  cntTerminate ++;
```

Mittels einer statischen Zählervariable werden die Aufrufe mitgezählt. Ist der Zählerstand gleich mit der Sollanzahl von Durchläufen wird die Verbindung beendet.

PcDspIO.c und PcDspIO.h: Auch auf der PC Seite wird eine Wrapperdatei erstellt um den Datenaustausch zu steuern.

Für jede zu sendende Datenstruktur wird eine Funktion mit dem Namen *SendData_VariablenNamen* erstellt. Bevor die Daten mittels *SendVariable* gesendet werden können muss eine gültige Datenstruktur erstellt werden. Dafür wird eine temporäre Datenstruktur vom benötigten Typ angelegt. Danach wird sie mit einer gültigen *var_id* versehen. Weiters müssen noch die Daten von der übergebenen Speicherstelle kopiert werden. Dafür ist die Größe, die ebenfalls übergeben wird, notwendig. Die Funktion *SendVariable* befindet sich in der Datei `moduleIO.c` und stellt die Verbindung zu den

Low-Level Funktionen her. Deren Rückgabewert wird an die S-Funktion weitergeleitet und dort ausgewertet.

Beim Empfangen wird ebenfalls eine temporäre Datenstruktur angelegt. Wurden die Daten mittels *ReceiveVariable*, ebenfalls in `moduleIO.c`, richtig empfangen, werden sie an die übergebene Speicherstelle kopiert. In der Funktion *ReceiveVariable* werden die Daten mittels der Low-Level Funktion *RTS_Receive* empfangen. Diese Funktion blockiert den Ablauf solange, bis gültige Daten empfangen wurden. Um zu vermeiden, dass dies geschieht wird vorher mittels *RTS_DataAvail* auf gültige Daten geprüft. Die Zeit die gewartet wird kann der Funktion als Parameter übergeben werden. Der Wert wird standardmäßig auf 5000000 μs gesetzt. Kommt es während der Simulation zu einem receive error, sollte zuerst diese Zeit erhöht werden.

globalTypesPc.h: Es sind hier im Wesentlichen dieselben Informationen wie in der DSP-Variante enthalten. Einzig die Makrodefinition *MAKE_ID*, zum Erstellen der *var_id* kommt hier dazu. In der DSP Version ist diese schon in der Datei `global.h` enthalten und wird dort, wegen Mehrfachdeklaration, nicht verwendet.

dsp.pjt Um die Möglichkeit zu nutzen eine `.out` Datei über die Kommando Zeile mittels `timake` zu erstellen wird die Projektdatei benötigt. Diese kann mittels Code Composer Studio von Texas Instruments oder mittels *GenCaddOn* erstellt werden. Der Vorteil von letzt genannter Möglichkeit ist es, dass der gesamte Umwandlungs- und Simulationsablauf aus Matlab bzw. aus einem Matlab-Script gesteuert werden kann. Als Nachteil kann hier gesehen werden, dass Compileroptionen händisch geändert werden müssen. Die Option des Projektdateierstellens kann mittels Parameter `-p` aktiviert werden. Sind weitere nicht Standarddateien notwendig, müssen diese unter `[Source Files]` eingetragen werden.

4.4.2 Ergebnis Sundance

dspSMT.c: Es müssen hier zwei Änderungen (siehe Anhang C.3.3) gegenüber der von *GenC* erstellten Datei gemacht werden. Der Datenaustausch wird auch hier mittels eines Wrappers durchgeführt. Um diese Funktionalität nutzen zu können, wird eine zusätzliche *include* Anweisung eingeführt und nach der Variablendeklaration die Hardwareinitialisierungsroutine aufgerufen.

DspPcIO.c und DspPcIO.h: In der Initialisierungsroutine wird, wie der Name schon erahnen lässt, die Hardware initialisiert. Zum einen werden die Grundfunktionen und andererseits der High Speed Channel initialisiert. Konnte dies erfolgreich durchgeführt werden wird dies dem PC mitgeteilt. Dazu wird über das *comport* ein *OK*, entspricht Wert *0*, gesendet, ansonst

der jeweilige Fehlercode. Gültige Fehlercodes sind *2* für „Grundfunktion konnte nicht geöffnet werden“ und *3* für einen Fehler beim Initialisieren des HSC.

Für jede Leseoperation wird eine eigene Funktion, *call_VariablenNamen*, bereitgestellt. Der Datenaustausch zwischen DSP und PC erfolgt über den *HSC*, dies wegen der höheren Datenraten. Bevor Daten gelesen werden können muss der betreffenden PCI-Bereich geöffnet und danach wieder geschlossen werden. Dies erledigen die Funktionen *OpenPci()* und *ClosePci()*. Der eigentliche Lesezugriff erfolgt mittels *Smt_HscPciRead* und benötigt unter anderem die Größe der zu lesenden Daten und einen Zeiger auf den Speicherbereich für die empfangenen Daten. Diese Funktion liefert die tatsächlich erfolgreich gelesenen Bytes zurück. Diese werden vor Verlassen der Funktion an den PC über ein *comport* gesendet. Dies soll zu einer besseren Fehlererkennung am PC dienen.

Die Übertragung an den PC erfolgt nach demselben Prinzip wie beim Empfangen. Auch hier muss der PCI-Bereich geöffnet und nach erfolgtem Senden wieder geschlossen werden. Dies geschieht über die Funktionen *OpenPci* und *ClosePci*. Die Senderoutine *Smt_HscPciWrite* liefert die tatsächlich geschriebenen Bytes zurück. Diese werden zur Kontrolle an den PC mittels *comport* übertragen.

MAKEFILE und dspSMT.cfg: Um die benötigte `.app` Datei zu erstellen wird von Sundance das Programm `NMAKE` bereitgestellt. Dieses benötigt neben den Sourcedateien eine Konfigurationsdatei und ein `MAKEFILE`. Die Konfigurationsdatei beinhaltet Informationen über das Modul, Namen des Prozesses und Speicheraufteilung. Im `MAKEFILE` sind Informationen für den Kompiliervorgang abgelegt. Dies beinhaltet hauptsächlich die Namen der Sourcedateien für das Projekt. *GenCaddOn* erstellt die Projektdateien automatisch falls der Parameter (`-p`) angegeben wird. Zu beachten ist, dass die Konfigurationsdatei immer für das Modul *TIM SMT365_8_2* erstellt wird.

dspIO.cpp: Am Dateianfang wird auch hier der *S_FUNCTION_NAME* um *IO* erweitert. Weiters folgt die Includeanweisung für das Einbinden der Wrapperdatei und die Definition der statischen Variable *isInit* für die Initialisierungsroutine. Diese wird in der Funktion *mdlInitializeConditions* aufgerufen.

Die Initialisierung wird nur einmal ausgeführt. Dies wird durch die Abfrage der *isInit* Variable realisiert. Beim ersten Mal wird diese auf den Wert *true* gesetzt und somit ein weiterer Aufruf verhindert.

Der ursprüngliche Algorithmus wird durch die benötigten Sende- und Empfangsroutinen ersetzt. Für die Empfangsroutinen werden temporäre Variablen sowie eine Zählervariable angelegt. Letztere wird für die Zuweisung der temporären auf die Ausgangsvariablen benötigt.

PcDspIO.cpp und PcDspIO.h: Die Kommunikationssteuerung ist in der Wrapperdatei enthalten. In der Initialisierungsroutine wird die Verbindung zum Board hergestellt und der *HSC* konfiguriert. Weiters wird das Modul resetiert und mit dem Programm geladen. Das Programm muss als `.app` Datei im Verzeichnis *DspProject* vorliegen.

Für jeden Ein- und Ausgangswert ist eine eigene Funktion mit den Bezeichnern *SendData_VariablenName* bzw. *ReceiveVariable_VariablenName* vorgesehen. Diese rufen jeweils die Funktion *SendVariable* oder *ReceiveVariable* auf und leiten deren Statusmeldung zurück.

Beim Senden werden am Beginn Daten in ein temporäres Array kopiert. Danach wird dieser Buffer für nicht berechtigte Zugriffe mittels der Funktion *PciOpenSync* gesperrt. Sollte dies nicht möglich sein wird der Datenbuffer gelöscht und eine Fehlermeldung zurückgeliefert. Die Daten werden über einen PCI Speicherbereich ausgetauscht. Zuerst wird eingelesen wie viele Bytes der DSP lesen möchte. Stimmen diese nicht mit denen des PCs überein, wird der Datenaustausch mit einem Fehler abgebrochen. Der PC gibt dem DSP die Anzahl der verfügbaren Bytes bekannt und schließt danach mittels der Funktion *PciCloseSync* die Datenübertragung. Wie beim DSP Wrapper erwähnt schickt der DSP nach dem *ClosePci* die Bytes, die er tatsächlich gelesen hat. Diese werden am PC mittels *CpRead* eingelesen und mit der zu sendenden Anzahl verglichen. Stimmen diese nicht überein wird der Programmablauf unterbrochen.

Das Empfangen von Daten funktioniert nach demselben Schema. Der Unterschied ist nur, dass hier die Daten vor dem Schließen der Synchronisation kopiert werden.

Kapitel 5

Matlab-Hardwareinterface

5.1 Allgemeines

Da es nicht immer zweckdienlich ist eine Simulation unter Simulink durchzuführen, wurde auch eine Schnittstelle zwischen Matlab und den beiden Hardwareplattformen erstellt. Wie schon in Abschnitt 3.1.1 beschrieben bieten mex-Dateien die Möglichkeit, C-Routinen in Matlab zu integrieren. Da die Low-Level Funktionen für die Kommunikation schon als C-Code vorliegen wurden auch die Routinen, die auf diese Funktionalität aufsetzen, in C programmiert. Die Schnittstelle zu Matlab bildet die mexFunction, die sich in `comRTS.c` bzw. `comSMT.cpp` befindet und dadurch den Funktionsaufruf bestimmt. Unter Matlab kann auf die Funktionalität mittels

```
comRTS('Kommando',Argument1,Argument2,..., ArgumentX);
```

oder

```
comSMT('Kommando',Argument1,Argument2,..., ArgumentX);
```

zugegriffen werden. Der Parameter `'Kommando'` stellt den auszuführenden Befehl dar. Je nach Befehl ist eine bestimmte Anzahl von Argumenten notwendig. Die Anzahl kann bei gleichen Befehlen variieren und ermöglicht dadurch eine flexiblere Programmgestaltung. Weiters wurden auch Funktionalitäten zu einem Befehl, wie z. B. `con2board`, zusammengefasst. Der vorher genannte Befehl stellt eine Verbindung zum RTS-Board her.

5.2 Aufbau

Bei der Erstellung des Konzeptes wurde versucht ein möglichst einfaches, aber dennoch flexibles Interface zu realisieren. Dieses Vorhaben ist in Abbildung 5.1 skizziert. Die Schnittstelle zwischen Matlab und der Hardware wurde in vier Module aufgeteilt. Wobei die Module mit den Low-Level Hardwarefunktionen von den Herstellern schon bereitgestellt werden.

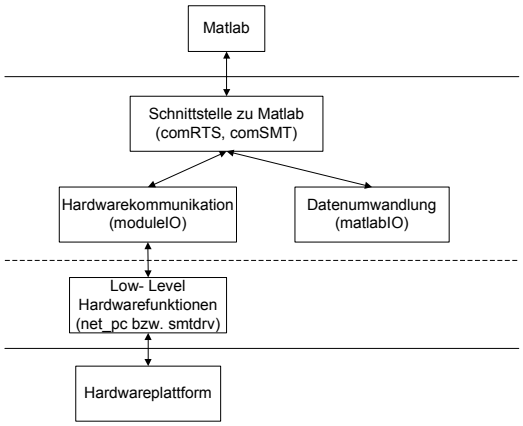


Abbildung 5.1: Konzeptioneller Aufbau des Matlab-Hardware-Interfaces.

5.2.1 Schnittstelle zu Matlab

Hier wird nach der Überprüfung der Kommandos und Programmargumente auf die jeweiligen Unterprogramme verzweigt. Es wird hier der gesamte Ablauf gesteuert. Die unterstüzten Kommandos und deren Bedeutung und möglichen Argumente werden in Abschnitt D.1 und D.2 noch näher beschrieben. Weiters beinhaltet sie die schon öfters erwähnte *mexFunction*. Der Aufbau der beiden Dateien weicht nur wenig voneinander ab. Der Unterschied besteht im Aufruf der Unterprogramme für die verschiedenen Hardwaresysteme. Ein weiterer Grund für die Aufteilung in zwei verschiedene Dateien dient der besseren Unterscheidung.

5.2.2 Datenumwandlung

Ein wesentlicher Teil des Interfaces ist das Einlesen, das Umwandeln und die Ausgabe der Daten. Weiters ist es möglich Datenstrukturen zu speichern und empfangene Daten in diese Struktur umzuwandeln. Das Speichern ist beim Senden sowie mittels eines eigenen Befehls möglich.

Es können maximal 100 verschiedene Strukturen gespeichert werden. Die Informationen werden in einem Array des Types *TStruct* gespeichert. Diese Struktur enthält alle relevanten Informationen um empfangene Daten in eine bestimmte Matlab Struktur umzuwandeln.

Eine zu speichernde Struktur muss einige Anforderungen erfüllen um gespeichert werden zu können. Nachfolgend ist die Strukturvereinbarung mit dem Etikett `TStruct` angeführt.

```
typedef struct {
  char fieldname[30][30];
  char dimensions[30][20];
  char datatype[30][10];
  char className[30];
  bool isComplex[30];
  int  sizeInBytes;
} TStruct;
```

Die Liste der Deklarationen besteht aus sechs Variablen, die die zu speichernden Informationen aufnehmen. Die maximale Anzahl von Datenfeldern, die eine Struktur enthalten darf, wurde mit 30 festgelegt. Gespeichert wird der Name jedes Feldes (*fieldname*), der auf eine Länge von 30 Zeichen limitiert wurde. Jedes Feld kann eine unterschiedliche Dimension und einen anderen Datentyp aufweisen, deshalb ist es notwendig, diese Informationen auch für jedes Feld festzuhalten. Dies geschieht mittels den Variablen *dimensions* und *datatype*. Mittels der Variable *isComplex* wird festgehalten, ob es sich im jeweiligen Datenfeld um komplexe Daten handelt oder nicht. Als zusätzliche Informationen werden noch der Name der Klasse (*className*) und die Größe *sizeInBytes*, in Bytes, der gesamten Struktur gespeichert.

An Matlab wird die Nummer, an welcher Stelle die Struktur gespeichert wurde, beginnend mit *1*, zurückgeliefert. Diese Zahl muss, wenn die empfangenen Daten in eine Struktur gewandelt werden sollen, beim Befehl *receive* übergeben werden.

Weiters besteht die Möglichkeit, alle gespeicherten Strukturen zu löschen. Das Löschen einer einzelnen Struktur wurde nicht realisiert. Der Grund dafür liegt im hohen Realisierungsaufwand des dafür notwendigen Speichermanagements. Es wurde außerdem angenommen, dass 100 Speicherplätze ausreichen sollten. Ist dies nicht der Fall, kann diese Anzahl einfach erhöht werden, indem der Paramter `#define NOOFSTRUCTS 100`, der sich in der Datei `matlabIO.h` befindet, entsprechend verändert wird.

Das Einlesen der Daten und Umwandeln in C-Format wird von der Funktion

```
int GetInputStructure(const mxArray *pStrArr,int *size,char *data,
                      int save, int *saveVal);
```

gesteuert. Zuerst wird die Klasse der von Matlab übergeben Daten bestimmt und zur jeweiligen Analyseroutine weitergeleitet. Es existiert für jeden mögliche Datentyp eine eigene Routine, die die Daten ausliest und abspeichert. Das Speichern erfolgt in einem Character Array der Länge *DATAMAXLEN*, und wird der Funktion, in Form eines Zeigers, übergeben. Der Parameter *DATAMAXLEN* wird in Abschnitt 5.2.4 noch näher erklärt.

Die empfangenen Daten können auf zwei verschiedene Arten zurückgeliefert werden. Einmal als Array oder in Form einer gespeicherten Struktur. Wird die erste Variante gewählt werden alle empfangenen Daten in ein Array

eines von Matlab unterstützten Datentyps umgewandelt. Dieser Datentyp muss mitübergeben werden. Der Funktionsaufruf sieht wie folgt aus:

```
mxArray *Convert2Output(void *data, char *datatyp, int bytes,
                        int boardType);
```

Wie schon erwähnt sind die Funktionen von *matlabIO* hardwareunabhängig. Intern muss aber, je nach Plattform, eine andere Art der Umwandlung vollzogen werden. Mittels des Parameters *boardType* wird die Zielplattform ausgewählt wobei *1* dem ARCS-RTS-Board und *0* dem Sundance SMT Modul entspricht. Dem Nutzer entsteht aber dadurch kein Mehraufwand, da dies intern geregelt wird. Der Paramter *bytes* wird bei RTS auf den Wert *0* gesetzt, da dieser hier nicht benötigt wird. Die Information über die Länge ist hier im Wert der Variable *var_id* enthalten. Dieser Wert wird an Matlab zurückgeliefert. Dies geschieht in Form einer Struktur, wobei das erste Element die *var_id* und das zweite Element das eigentliche Datenarray enthält.

Sollen die Daten in eine Struktur umgewandelt werden wird dies an die Funktion

```
mxArray *Convert2OutputStruct(void *data, int structNo,
                              int noOfElem);
```

weiter delegiert. Vorraussetzung hierfür ist, dass die Struktur bekannt ist. Wird das RTS-Board verwendet, muss der Benutzer dafür sorgen, dass die von Matlab übergebene Struktur die Variable *var_id* enthält.

5.2.3 Hardwarekommunikation

Um auf die jeweilige Hardware zugreifen zu können stellen die beiden Hersteller Low-Level Funktionen zur Verfügung. Auf diesen baut *moduleIO* auf um den Datentransfer durchzuführen. Neben der Sende- und Empfangsroutine sind auch der Verbindungsaufbau und das Trennen der Verbindung hier enthalten. Der Aufbau der Low-Level Funktionen macht es notwendig für die jeweilige Hardware eine eigene Datei anzulegen.

5.2.4 Globale Projektparameter

Hier werden allgemeingültige Definitionen und Macros sowie *includes* und *Bibliotheken* bereitgestellt. Nachfolgend sind diese für `comRTS` angeführt. Bei `comSMT` ist nur ein Parameter, und zwar *DATAMAXLEN* vorgesehen. Dies liegt an der unterschiedlichen Datenübertragung und dem Hardwareaufbau.

```
#define DATAPORT  5000
#define REGPORT   5001
#define RACKID    10
#define RECEIVEBUFFER 10240*10240
#define TIMEOUT_DATAAVAIL 100000
```

```
#define DATAMAXLEN 10240

#define ID_TO_SIZE( ID ) (((ID)&0x00FFFFFF)*4)
#define MAKE_ID( type, size)((((unsigned int)(type))<<24)+size/4)
```

Die ersten fünf Werte stellen die *default* Werte bei einigen im nächsten Abschnitt beschriebenen Befehlen dar. Es ist daher nicht erforderlich, diese jedesmal in `global.h` zu ändern und das Projekt neu zu kompilieren.

Der Parameter *DATAMAXLEN* wird für temporäre Variablen benötigt. Sollten die Sende- oder erwartenden Empfangsdaten größer als *10240* Bytes (C Speicherbelegung) sein, muss dieser Parameter erhöht werden.

Mit dem *RECEIVEBUFFER* Parameter wird die Größe des Empfangsspeichers für die empfangenen UDP-Pakete angegeben. Ist mit Datenmengen größer 1 MByte zu rechnen, sollte dieser Wert entsprechend geändert werden.

Da die Funktion *RTS_Receive* blockierend ist, wird in *net_pc*[1] [6] eine Funktion zur Verfügung gestellt, die eine gewisse Zeit lang überprüft, ob Daten vorhanden sind. Diese Zeit kann beeinflußt werden, indem der Parameter *TIMEOUT_DATAAVAIL* (in μs) geändert wird oder als optionales Argument bei der *receive* Funktion übergeben wird.

[1] Enthält die Low-Level Funktionen für das RTS-Board.

5.3 Mögliche Befehle

In der nachfolgende Tabelle sind alle möglichen Befehle für die beiden Hardwaresysteme aufgelistet. Eine detailiertere Beschreibung befindet sich im Anhang D.

Befehl	Kurzbeschreibung
für RTS-DSP-Board	
open	Öffnet eine Verbindung zur Hardware.
close	Beendet eine bestehende Verbindung zur Hardware.
con2Board	Öffnet und stellt eine Verbindung zur Hardware her.
resetBoard	Schließen und erneutes Öffnen einer Verbindung.
send	Senden von Daten.
receive	Empfangen von Daten.
sendregcmd	Sendet Registrierungskommando an die Hardware.
waitforregcon	Wartet auf die Registrierung der Hardware.
saveStruct	Speichern einer Struktur.
clearStruct	Löschen aller gespeicherten Strukturen.
für Sundance Modul	
init	Initialisiert eine Verbindung zur Hardware.
loadProg	Programmiert eine `.app` Datei.
resettim	Resetiert die vorhandenen TIMs.
resetboard	Resetiert die gesamte Hardware.
cpread	Iniziert einen comport Lesevorgang.
cpwrite	Schreibt Daten auf das comport.
sendpci	Datenaustausch über PCI-Bus.
sendsram	Datenaustausch über SRAM.
receivepci	Datenaustausch über PCI-Bus.
receivesram	Datenaustausch über SRAM.
close	Schließt das Interface zur Hardware.
saveStruct	Speichern einer Struktur.
clearStruct	Löschen aller gespeicherten Strukturen.

Tabelle 5.1: Übersicht über alle Befehle für das RTS-DSP-Board und das Sundance Modul.

Kapitel 6

Tests und Beispiel

6.1 Simulationsablauf

Dieser Abschnitt soll zeigen wie aus den in Abschnitt 4.2.2 beschriebenen Dateien jene für die Kommunikation mit der Hardware erzeugt werden. Der gesamte Ablauf kann von Matlab aus mittels eines Matlab-Scripts gesteuert werden. Im Anhang B ist ein examplarisches Script sowohl für Sundance als auch für ARCS angeführt.

Das Script muß nicht in allen Punkten eingehalten werden und kann, je nach Bedarf, abgeändert werden. Zu Beginn sollte der Workspace mittels *clear mex* aufgeräumt werden um Fehler in der Simulation zu vermeiden.

Danach müssen die Dateien für *GenCaddOn* erzeugt werden. Dies erfolgt über *GenC* mit den Parametern `SL` und `3L`. Danach erfolgt die Umwandlung auf die jeweilige Zielplattform. Anschließend müssen noch die für den PC erzeugeten Interface Dateien in mex-Dateien umgesetzt werden.

Die Compilierung der DSP Projektdateien kann auch aus Matlab erfolgen. Wurde der Code für die Sundance Hardware erstellt, kann hier die Simulation durchgeführt werden. Wurde für das RTS-Board entwickelt, muss noch das Programm mittels LANFlasher programmiert werden.

Das Erstellen und das Kompilieren von weiteren Dateien, die für die Simulation benötigt werden, kann zu jedem beliebigen Zeitpunkt, aber natürlich vor dem Simulationsstart, erfolgen. Im Script für das RTS-Board sind weiters Resets der Hardware vorgesehen. Diese sollten unbedingt ausgeführt werden um eine fehlerfreie Simulation durchführen zu können.

6.2 Test der Funktionsweise

Ein wesentlicher Teil jeder Arbeit stellen die Funktionstests dar. Im folgenden Abschnitt soll geklärt werden wie und was geprüft wurde.

Da bei der Konzeptionierung der einzelnen Software-Elemente darauf geachtet wurde, dass sie universell einsetzbar sind, konnte dadurch der

Prüfungsaufwand verkleinert werden. Dies bezieht sich vorallem auf die beiden Matlab-Schnittstellen zu den Hardwareplattformen. Mit diesen beiden soll nun auch begonnen werden.

6.2.1 Matlab-Hardwareinterface

Das Interface kann in drei Hauptteile gegliedert werden und zwar:

- Auswerten der übergebenen Befehle und deren Parameter
- Umwandlung der Nutzdaten in C- oder Matlab Daten und
- die Kommunikation mit der Hardware.

Die ersten beiden Punkte konnten für beide Plattformen verwendet werden, wobei bei Ersterem gewisse Unterschiede bestehen, aber die Grundfunktionen dieselben sind.

Um die Auswertung zu überprüfen wurde ein Matlab Script erstellt, welches jeden gültigen Befehl sowie alle möglichen Parameterkonstelationen aufruft. Es wurden aber auch ungültige Kommandos und Argumente übergeben um die Fehlererkennung innerhalb der mex-Datei zu überprüfen.

In der mex-Datei wurde anstelle von den Funktionsaufrufen einfache *Command Window* Ausgaben durchgeführt. Diese gaben Auskunft über die Gültigkeit des übergeben Befehls oder aber auch um welchen Fehler es sich handelt. Die Fehlersuche wurde dahingehend erleichtert, dass es, wie schon in Abschnitt 3.1.1 erwähnt, möglich ist, mex-Dateien unter Visual Studio zu entwickeln. Vergleicht man die Befehle von Abschnitt D.1 und Abschnitt D.2 stellt man fest, dass sich hier in vielen Fällen nur das Kommando unterscheidet, nicht aber die Parameter und die Alternative-Aufrufe. Da sich auch die Umsetzung in der mex-Datei dadurch nicht unterscheidet, außer anderer Kommandoname, mussten nicht zweimal die gesamten Variationen überprüft werden.

Die Umwandlung der Nutzdaten in C- oder Matlab-Daten ist bei beiden Hardwaresystemen annähernd gleich. Beim Einlesen der Daten gibt es zwischen den Plattformen keinen Unterschied und bei der Ausgabe ist die Abweichung, nur der Variablen Identifier (*var_id*), nur minimal. Überprüft werden musste,

- bei den Eingangsdaten
  - die korrekte Analyse der Daten,
  - deren Umwandlung in C- Speicherformat sowie
  - das Speichern der Strukturinformationen und

- bei den Ausgangsdaten
  - die Umwandlung in ein Matlab Array, abhängig vom Datentyp,
  - das Konvertieren in eine Datenstruktur und
  - jeweils für beide Hardwareplattformen.

Der Ablauf der Tests wurde in der Reihenfolge der vorherigen Aufzählung durchgeführt. Die korrekte Analyse der einzelnen Datentypen, sowie deren Umwandlung in das Speicherformat von C kann noch vollständig überprüft werden. Schwieriger gestaltet sich dies bei Arrays und vor allem bei Datenstrukturen. Hier kann wegen der großen Anzahl von Möglichkeiten nur ein kleiner Anteil kontrolliert werden. Bei der Auswahl der Testfälle wurde darauf geachtet, dass diese so viele Variationen wie möglich beinhalten, um eine richtige Funktion nachzuweisen. Dasselbe gilt auch für die Umwandlung in die gewünschten Datenformate der empfangenen Daten. Die Umwandlungsroutinen wurden ohne Hardwarezugriffe getestet um möglichen Fehlern der Kommunikationschicht zu entgehen.

Die korrekte Funktion der Zugriffe auf die Hardware wurde jeweils durch ein C-Programm kontrolliert. Der Grund dafür lag daran, dass beide Hardwarehersteller einfache Programme und den zugehörigen Sourcecode bereitstellten. Diese Programme wurden als Basis für die Kommunikationsschicht verwendet, die um dieses herum aufgebaut wurde.

Die Überprüfung der Hauptteile erfolgte teilweise schon während der Erstellung, spätestens aber nach Fertigstellung. Dadurch wurde der Aufwand, der für den Test des Gesamtsystems notwendig ist, minimiert.

6.2.2 GenCaddOn

Während der Entwicklung des Programmes wurden die einzelnen Routinen schon auf ihre Funktion hin überprüft. Als Input kam hier die im Anhang C.1 angeführte *Generic C* Datei `dsp.gc` zum Einsatz. Der Grund warum diese Datei verwendet wurde ist, dass mehrere Ein- und Ausgänge mit unterschiedlich einstellbaren Raten vorhanden sind.

Die Überprüfung des vom Programm produzierten Outputs wurde in zwei Bereiche aufgeteilt. Einen Teil stellt das DSP-Projekt, den zweiten Teil die S-Funktion, jeweils mit Kommunikationsschnittstelle, dar. Diese beiden Teile werden auch bei den zwei Hardwaresystemen unterschieden. Nachfolgend wird die Vorgehensweise für das RTS-Board beschrieben.

Der DSP-Bereich wurde mittels C-Programm und mittels Matlab-Interface getestet. In der Entwicklunsphase wurde die JTAG-Funktion des DSP's genützt. Dies weil der LANFlasher noch nicht zur Verfügung Stand. Der eigentlich Hauptgrund ist die Debug-Möglichkeit, die das CCS bietet.
Die Testfälle umfassten verschiedenste Raten der Ein- und Ausgänge und variieren der Anzahl. Weiters wurden auch deren Datentypen geändert.

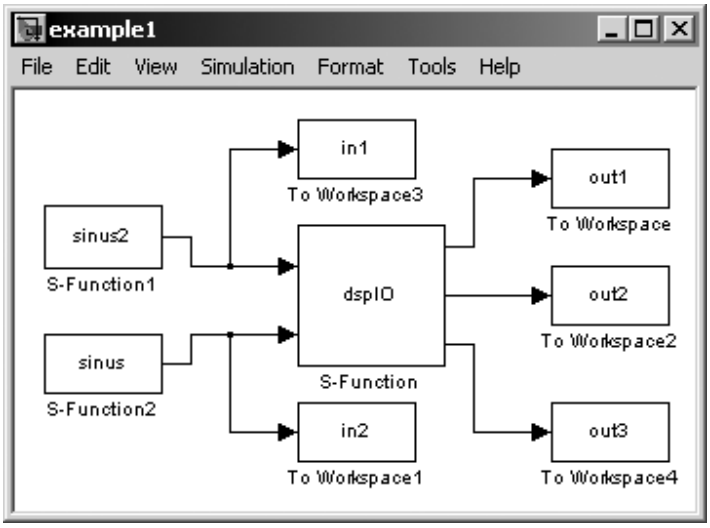


Abbildung 6.1: Simulink Modell.

Für die Generierung und Auswertung der Daten für und vom DSP wurde ein eigenes Programm erstellt. Als Basis wurde ein Beispiel-Programm zur Hardware verwendet, da hier die Funktionstüchtigkeit sichergestellt ist.

Anschließend konnte die Funktionsweise der S-Funktion überprüft werden. Es wurden hier die selben Testfälle wie am DSP angewendet. Das Simulationsmodell, das hier zur Anwendung kam, ist in Abbildung 6.1 für den Testfall mit zwei Eingängen und drei Ausgängen dargestellt. Je nach Testbedingungen musste diese entsprechend konfiguriert werden.

Der Berechnugsalgorithmus wurde während der gesamten Tests nicht wesentlich verändert. Lediglich wenn Ein- und/oder Ausgänge hinzugefügt oder entfernt wurden, musste die Berechnung verändert werden. Es wurde immer versucht die Grundzüge beizubehalten. Der Gund dafür lag darin, dass bei diesem Algorithmus das Resultat der empfangenen Daten Null ergab und dadurch schnell die Richtigkeit festgestellt werden konnte.

6.3 Analyse des Ergebnisses

Einer der Anwendungsbereiche soll neben der schnelleren und einfacheren Umsetzung die Auslagerung von Simulationsteilen auf Hardware sein. Vorweg kann schon gesagt werden, dass das Auslagern auf Hardware nicht immer einen schnelleren Simulationsablauf bewirkt.

Den größten Einfluss auf die Simulationsgeschwindigkeit haben das verwendete System und die Programme. Nachfolgend sind diese beiden Komponenten in groben Zügen angeführt.

Testsystem: Dieses setzte sich aus einem AMD Athlon XP 2400+ Prozessor und 512 MB RAM zusammen. Als Betriebssystem kam Microsoft Windows 2000 mit Service Pack 4 zum Einsatz. Als Compiler für die mex-Dateien, S-Funktionen und C-Programme wurde Microsoft Visual Studio Version 6.0 verwendet. Die Simulation wurde unter Matlab Version 6.5.1.199709 Release 13 (Service Pack 1) durchgeführt. Als DSP Algorithmus kam das schon öfters erwähnte Programm `dsp.gc` zu Einsatz. Neben den schon bekannten Optionen kann die Berechnungszeit auch durch die zweite Schleife mittels des Parameters *KMAX* beeinflußt werden.

Der größte Zeitverlust ergibt sich durch die Initialisierung der Hardware. Weitere Verluste müssen durch den Datentransfer hingenommen werden. Dies kann dann eingeschränkt werden, wenn größere Datenpakete auf einmal transferiert werden. Mit den Tabelle 6.1 und 6.2 und Abbildung 6.2 soll dies verdeutlicht werden. Als Testalgorithmus wurde das Programm `dsp.c` (C.1) verwendet. Mit dem Parameter M wurde die Größe der Datenpakete und mit *KMAX* die Berechnungszeit des DSP variiert.

In Abbildung 6.2 sind die Datenraten in Abhängigkeit von M für die beiden Hardwareplattformen dargestellt. Es wurden für jedes System zwei verschiedene Kurven ermittelt. Einmal wird die Initialisierung in die Messung einbezogen und einmal zählt diese nicht zur Messung. Dies wird durch die Zusätze „mit“ und „ohne“ gekennzeichnet. Dies kann als Zeitverlust durch die Initialisierung gesehen werden. Wie schon erwähnt verzögert auch der Datentransfer die Simulation. Dies wird deutlich, wenn man die beiden Kurven ohne die Initialisierungsroutine miteinander vergleicht. Der große Unterschied kann durch den unterschiedlichen Datenweg erklärt werden. Bei Sundance erfolgt der Austausch der Daten nur innerhalb des PCs über den PCI Bus. Der Datenaustausch bei der Hardware von ARCS erfolgt über eine Ethernetverbindung. Weiters müssen die Daten noch vom Prozessor an den FPGA weitergeleitet werden bevor sie zum DSP gelangen. In Abschnitt 4.4.1 wurde erwähnt, dass vor jeder Sende- und Empfangsroutine eine 20 μs lange Wartezeit eingefügt wurde. Es steckt also noch Potential für eine Erhöhung der Datenrate in der Übertragung.

In nachfolgenden Tabellen sind unter anderem die Anzahl der Durchläufe in Abhängigkeit von M aufgelistet. Außerdem sind die gesamt transferierten Daten in MByte angeführt. In der letzten Spalte befinden sich die Anzahl der Durchläufe, wenn die Simulation nur unter Simulink durchgeführt wird. Vergleicht man diese Zahlen miteinander, stellt man vorallem bei kleinen Datenmengen fest, dass die Unterschiede beträchtlich ausfallen. Je größer diese jedoch werden desto geringere Unterschiede sind feststellbar.

Die Werte *F1*, *F2* und *F3* kennzeichnen Fehler während der Simulation. Nachfolgend ist der Matlab Fehlercode angeführt.

```
??? Error using ==> sim
Memory allocation error.
```

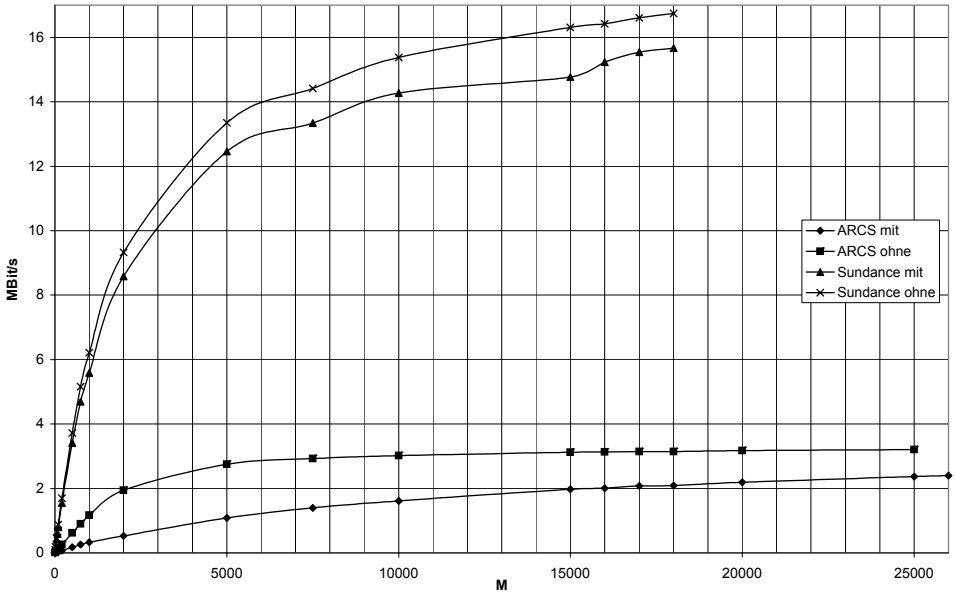


Abbildung 6.2: Datenraten in MBit/s in Abhängigkeit des Parameters M.

Ab diesen Werten konnten keine weiteren Tests durchgeführt werden.

Erhöht man den Berechnungsaufwand mittels $KMAX$, fallen diese Unterschiede immer geringer aus.

Abschließend kann gesagt werden, dass eine Simulation auf der Hardware nur bei großen Datenmengen und vielen Berechnungen schneller durchgeführt werden kann.

M	RTS mit AdD	RTS ohne AdD	SMT mit AdD	SMT ohne AdD	Simulink AdD
10	35	137	895	938	1779
20	36	138	875	948	1779
50	36	137	856	940	1779
75	36	137	826	922	1531
100	37	137	831	913	1699
200	37	136	810	882	1647
500	36	129	711	775	1529
750	36	125	653	718	1391
1000	35	122	582	648	1293
2000	28	102	447	486	1012
5000	23	58	260	279	618
7500	20	41	186	201	465
10000	17	32	149	161	374
15000	14	22	103	114	265
16000	14	21	100	107	247
17000	13	20	96	102	218
18000	13	19	91	97	207
20000	12	17	F2	F2	194
25000	10	14	-	-	F3
26000	10	F1	-	-	-
27000	F1	-	-	-	-

Tabelle 6.1: Anzahl der Durchläufe(AdD) mit und ohne Initialisierung, jeweils für die Hardware von ARCS (RTS) und Sundance (SMT), sowie unter Simulink über einen Zeitraum von ca. 30 Sekunden.

M	SMT mit MByte	SMT ohne MByte	RTS mit MByte	RTS ohne MByte
10	0,013	0,052	0,322	0,338
20	0,027	0,102	0,626	0,683
50	0,066	0,249	1,541	1,692
75	0,098	0,373	2,230	2,489
100	0,134	0,496	2,992	3,287
200	0,267	0,982	5,832	6,350
500	0,649	2,325	12,798	13,950
750	0,973	3,377	17,631	19,386
1000	1,261	4,394	20,952	23,328
2000	2,017	7,346	32,184	34,992
5000	4,140	10,441	46,800	50,220
7500	5,400	11,071	50,220	54,270
10000	6,120	11,521	53,640	57,960
15000	7,560	11,880	55,620	61,560
16000	8,064	12,096	57,600	61,632
17000	7,956	12,240	58,752	62,424
18000	8,424	12,312	58,986	62,856
20000	8,640	12,240	F2	F2
25000	9,000	12,600	-	-
26000	9,360	F1	-	-
27000	F1	-	-	-

Tabelle 6.2: Gesamttransfervolumen in MByte mit und ohne Initialisierung, jeweils für die Hardware von ARCS (RTS) und Sundance (SMT), sowie unter Simulink über einen Zeitraum von ca. 30 Sekunden.

Kapitel 7

Schlussbemerkungen

Die Wahl des Themas bzw. Aufgabenstellung kann auch nach Beendigung der Arbeit als absolut richtig gewählt betrachtet werden. Die am Beginn der Arbeit gesteckten Ziele konnten vollständig realisiert werden. Die Aufgabenstellung wurde auch um das Interface zwischen Matlab und den Hardwareplattformen erweitert. Dies wurde während des Entstehungsprozesses nach Gesprächen mit Mitarbeitern von ARCS beschlossen.

Der Entstehungsprozess spiegelt sich im Aufbau dieser Arbeit wieder. Am Beginn stand das Kennenlernen des ARCS RTS-DSP-Boards. Mit dem Modul von Sundance wurde schon während des Berufspraktikums gearbeitet und somit war hier kein Einarbeiten für die Diplomarbeit mehr von Nöten. Die nächsten Schritte stellten das Analysieren von *GenC*, mex-Dateien und S-Funktionen dar. Nach einer ersten groben Analyse wurde mit der Entwicklung von *GenCaddOn* begonnen. Bei dessen Realisierung traten einige Schwierigkeiten auf, die nachfolgend ein wenig betrachtet werden sollen.

Ein wesentliches Problem stellten die häufigen Änderungen in der Low-Level-Kommunikation des RTS-Boards dar. Dies resultierte daraus, dass die Entwicklung an dem Board noch nicht zur Gänze abgeschlossen war. Deshalb mussten während der Realisierung öfters Änderungen in *GenCaddOn* vorgenommen werden.

Am Schluss dieses Kapitels bzw. der gesamten Arbeit soll noch auf Verbesserungsmöglichkeiten eingegangen werden. So könnte die Möglichkeit geschaffen werden *GenCaddOn* parametrisierbarer zu gestalten. Der Benutzer könnte so bestimmen, welche Dateien er erstellen lassen möchte. Weiters könnte die Verzeichnisstruktur, die zurzeit fix vorgegeben ist, auch flexibler gestaltet werden. Dadurch würde die Möglichkeit bestehen, den Erstellungsprozess noch individueller gestalten zu können. Weiters könnte das Interface zwischen Matlab und dem Sundance-Modul noch um Befehle erweitert werden, wie z. B. um einen nicht default HSC Handler. Bei den Interfaces könnte weiters die Möglichkeit geschaffen werden, einzelne gespeicherte Strukturen zu löschen.

Anhang A

Verzeichnisstruktur für die Simulation

Anmerkung: Das Hauptverzeichnis ist für beide Hardwaresysteme ident und wird deshalb nur einmal angeführt. Dateien die während der Kompilierung des DSP-Projektes entstehen wurden nicht berücksichtigt.

A.1 Hauptverzeichnis

Pfad: /

compile.m	Matlab Script.
example.mdl	Simulink Modell.
GenC.exe	Programm zum umwandeln von *Generic C* Dateien.
GenCaddOn.exe	Erweiterungsprogramm zu *GenC*.
progname.c	Durch *GenC* erstellte DSP-Datei.
progname.cpp	Durch *GenC* erstellte S-Funktion.
progname.dll	Kompilierte S-Funktion.
prognameIO.cpp	S-Funktion von *GenCaddOn* generiert.
prognameIO.dll	Kompilierte S-Funktion mit Hardwarezugriffen.
progname.gc	*Generic C* Datei.

A.2 Unterverzeichnis für ARCS

Pfad: /DspProject/

Bios.cdb	Konfigurationsdatei zum Kompilieren des DSP-Projektes.
dsp.pjt	Projektdatei für TI CCS.
dspRTS.c	Hauptdatei des DSP-Projektes.

LANFlasher.exe	Programm zur Programmierung des DSP über Ethernet.

Pfad: /includes/

comFunct(.c,.h)	Kommunikationsschnittstelle zu *net_pc*.
globalTypesPc.h	Enthält Makros und Datenstrukturen.
net_pc(.c,.h)	Low-Level Hardwarefunktionen.
PcDspIO(.c,.h)	Sende-, Empfangs- und Initialisierungsroutinen.
platform.h	Definitionen für *net_pc*.

Pfad: /Tools/

CodeUpload(.c,.h)	Modul für das Flashen des DSP.
CycleBuffer(.c,.h)	Zugriff auf FPGA Cycle Buffer.
DSP_CPU_ComAddr.h . .	Definitionen für den DSP und die CPU.
DSP_MSG.h	Message Definitionen für den DSP.
DspPcIO(.c,.h)	Routinen für Kommunikation vom DSP zur CPU.
Flash(.c,.h)	Modul für das Flashen des DSP.
FPGA_Boot(.c,.h)	Bootroutinen für den FPGA.
global.h	Definitionen für das gesamte Projekt.
globalTypesDsp.h	Enthält Makros und Datenstrukturen.
LEDs.h	Definitionen für die LEDs.
net_dsp(.c,.h)	Kommunikation zwischen DSP und CPU.
platform.h	Definitionen für *net_dsp*
SendMsg(.c,.h)	Routinen zum Senden von Debugnachrichten.
SysTypes.h	System-Typen-Definitionen für verschiedene Compiler.

A.3 Unterverzeichnis für Sundance

Pfad: /DspProject/

DspPcIO(.c,.h)	Routinen für Kommunikation vom DSP zum PC.
MAKEFILE	Compilereinstellungen.
NMAKE.exe	Programm zum Erstellen der `*.app` Datei.
prognameSMT.app	DSP-Programm.
prognameSMT.c	Hauptdatei des DSP-Projektes.
prognameSMT.CFG . . .	Konfigurationsdatei für *NMAKE*.

Pfad: /includes/

hscdflt.h	Schnittstelle zum Default-HSC-Handler[1].
PcDspIO(.cpp,.h)	Sende-, Empfangs- und Initialisierungsroutinen.
Smtdrv.h	Enthält Definitionen für die Bibliothek[1].

[1] Nicht auf der CD-ROM enthalten, da sie einen Teil des SMT6025 darstellen.

SmtDrv.lib	Dient als Schnittstelle zur Hardware[2].
smthsc.h	Beschreibt die HSC-Schnittstelle[2].

[2]Nicht auf der CD-ROM enthalten, da sie einen Teil des SMT6025 darstellen.

Anhang B

Matlab-Scripte

B.1 Simulationsablauf für das ARCS RTS-DSP-Board `compileRTS.m`

```
clc;
clear;
disp('!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!');
disp('!! RESET BOARD AND WAIT UNTIL BOARD IS INITIALIZED !!');
disp('!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!');
pause;
clc;
clear mex;
dos('GenC dsp.gc SL');
dos('mex dsp.cpp ');
dos('GenC sinus.gc SL');
dos('mex sinus.cpp');
dos('GenC sinus2.gc SL');
dos('mex sinus2.cpp');

dos('GenC dsp.gc 3L');

stat = dos('GenCaddOn RTS dsp -p');
if(stat ~= 0)
    return;
end;
dos('mex DspIO.cpp includes/PcDspIO.c includes/comFunct.c
     includes/net_pc.c');

disp('compile DSP core.');
cd DspProject
[nmakeStatus, nmakeResult] = system('timake dsp.pjt DEBUG -a');
%if you want to see compiler outputs,
%use above line without semicolon (;)
if nmakeStatus~=0
    nmakeResult
    return;
end;
```

```
34
35 disp('download DSP Core.');
36 [nmakeStatus1, nmakeResult1] = system('LANFlasher -r
37                                        10 Debug\dspRTS.out')
38
39 if nmakeStatus1~=0
40     nmakeResult1
41     return;
42 end;
43
44 cd ..
45 clc;
46 disp('!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!');
47 disp('!! RESET BOARD AND WAIT UNTIL BOARD IS INITIALIZED !!');
48 disp('!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!');
49 pause;
50 clc;
51 sim('example1')
```

B.2 Matlab-ARCS-Hardware-Interface exampleRTS.m

```
1 IN1.var_id =335544325; %0x14000005
2 IN1.data = [1 2 3 4];
3
4 saveNr = comRTS('saveStruct',IN1);
5 conNr = comRTS('con2Board');
6 if(conNr >= 0)
7   for i=1:5
8     comDspPc('send',data);
9     OUT = comRTS('receive',saveNr,1);
10  end;
11
12  IN2 = [1 2 3 4];
13  saveNr2 = comRTS('send',IN2,20,'save');
14  OUT1 = comRTS('receive','double');
15  comRTS('close');
16 end;
```

B.3 Simulationsablauf für das Sundance-Modul compileSMT.m

```
1 clc;
2 clear;
3 clear mex;
4 dos('GenC dsp.gc SL');
```

```
5  dos('mex dsp.cpp ');
6  dos('GenC sinus.gc SL');
7  dos('mex sinus.cpp');
8  dos('GenC sinus2.gc SL');
9  dos('mex sinus2.cpp');
10
11 dos('GenC dsp.gc 3L');
12
13 stat = dos('GenCaddOn SMT dsp -p');
14 if(stat ~= 0)
15     return;
16 end;
17 dos('mex DspIO.cpp includes/PcDspIO.cpp');
18
19 cd DspProject
20 [nmakeStatus, nmakeResult] = system('nmake');
21 cd ..
22 if nmakeStatus~=0
23     nmakeResult
24     return;
25 end;
26
27 sim('example1')
```

B.4 Matlab Sundance-Hardware Interface exampleSMT.m

```
1  for i=1:10
2    IN1.data(i) = int32(i);
3    IN1.data1(i) = single(i);
4  end;
5
6  comSMT('init','dsp.app');
7  saveNr = comSMT('saveStruct',IN1);
8  for i=1:5
9    comSMT('sendpci',IN1,'save');
10   OUT1 = comSMT('receivepci',saveNr,1);
11 end;
12
13 comSMT('resettim');
14 comSMT('loadProg','HscDsp.app');
15 comSMT('cpwrite',int32(128));
16 comSMT('cpread','int32',1);
17 comSMT('close');
```

Anhang C

Generierte Dateien

Anmerkung: Aus den nachfolgenden Sourcecodes wurden aus Gründen der Übersichtlichkeit die gesamten Kommentare entfernt. In den *DSP-Main* und *Simulink S-Funktion* Dateien sind die Änderungen gegenüber den von *GenC* erstellten Dateien hervorgehoben.

C.1 Generic C Datei `dsp.gc`

```
1  #DEFINE M 1000
2  #DEFINE KMAX 1
3
4  INPUT_PORT(1) int *IN1;
5  RATE(INPUT_PORT(1)) = 2*M;
6  INPUT_PORT(2) int *IN2;
7  RATE(INPUT_PORT(2)) = M;
8
9  OUTPUT_PORT(1) float *OUT1;
10 RATE(OUTPUT_PORT(1)) = 2*M;
11 OUTPUT_PORT(2) float *OUT2;
12 RATE(OUTPUT_PORT(2)) = 2*M;
13 OUTPUT_PORT(3) float *OUT3;
14 RATE(OUTPUT_PORT(3)) = 2*M;
15
16 STATE float stat[1];
17
18 init_dsp()
19 {
20 stat[0]=0.0;
21 }
22
23 dsp()
24 {
25 int i,k;
26 for (i=0; i<M; i++)
27     {
28     OUT1[i*2]  =((float)(IN1[2*i]+IN2[i]))/2.0;
```

```
29      OUT1[2*i+1]=((float)(IN1[2*i+1]+IN2[i]))/2.0;
30      OUT2[i*2]   =((float)(IN1[2*i]-IN2[i]))/2.0;
31      OUT2[2*i+1]=((float)(IN1[2*i+1]-IN2[i]))/2.0;
32      for (k=0; k<KMAX; k++)
33        {
34        stat[0]=0.9*stat[0]+0.1*((float)IN1[2*i]);
35        OUT3[i*2]   =stat[0];
36        stat[0]=0.9*stat[0]+0.1*((float)IN1[2*i+1]);
37        OUT3[2*i+1]=stat[0];
38        }
39      }
40  }
```

C.2 Dateien für das ARCS DSP-Board

C.2.1 Projektdatei

```
; Code Composer Project File, Version 2.0
    (do not modify or remove this line)

[Project Settings]
ProjectDir="D:\FH-HAGENBERG\SicherungDA\Daten_DA\Test\ARC\DspProject\"
ProjectType=Executable
CPUFamily=TMS320C64XX
Tool="Compiler"
Tool="DspBiosBuilder"
Tool="Linker"
Config="Debug"
Config="Release"

[Source Files]
Source="..\Tools\CodeUpload.c"
Source="..\Tools\CycleBuffer.c"
Source="..\Tools\DspPcIO.c"
Source="..\Tools\Flash.c"
Source="..\Tools\FPGA_Boot.c"
Source="..\Tools\SendMsg.c"
Source="Bios.cdb"
Source="dspRTS.c"
Source="Bioscfg.cmd"

[Generated Files]
Source="Bioscfg.s62"
Source="Bioscfg_c.c"

["Compiler" Settings: "Debug"]
Options=-g -q -fr"D:\FH-HAGENBERG\SicherungDA\Daten_DA\Test
  \ARC\DspProject\Debug" -d"_DEBUG" -d"CHIP_6416" -ml3 -ms3 -mv6400
```

```
["Compiler" Settings: "Release"]
Options=-q -o3 -fr"D:\FH-HAGENBERG\SicherungDA\Daten_DA\Test
                    \ARC\DspProject\Release" -mv6400

["DspBiosBuilder" Settings: "Debug"]
Options=-v6x

["DspBiosBuilder" Settings: "Release"]
Options=-v6x

["Linker" Settings: "Debug"]
Options=-q -c -m".\Debug\dspRTS.map" -o".\Debug\dspRTS.out" -x

["Linker" Settings: "Release"]
Options=-q -c -m".\Release\dspRTS.map" -o".\Release\dspRTS.out" -x
```

C.2.2 DSP-Main Datei dspRTS.c

```
1  #include "../Tools/DspPcIO.h"
2
3  void main(void) {}
4  char *IN1_temp, *IN2_temp;
5  char *OUT1_temp, *OUT2_temp, *OUT3_temp;
6  int *IN1;
7  int *IN2;
8  float *OUT1;
9  float *OUT2;
10 float *OUT3;
11 int i,k;
12 static float stat[1];
13 static char first_run=0;
14 int size = 4;
15
16 IN1_temp = (char *)(0x80000000 + size);
17 IN1 = (int *)IN1_temp;
18 size += (2*M*sizeof(int)) + 4;
19 IN2_temp = (char *)(0x80000000 + size);
20 IN2 = (int *)IN2_temp;
21 size += (M*sizeof(int)) + 4;
22 OUT1_temp = (char *)(0x80000000 + size);
23 OUT1 = (float *)OUT1_temp;
24 size += (2*M*sizeof(float)) + 4;
25 OUT2_temp = (char *)(0x80000000 + size);
26 OUT2 = (float *)OUT2_temp;
27 size += (2*M*sizeof(float)) + 4;
28 OUT3_temp = (char *)(0x80000000 + size);
29 OUT3 = (float *)OUT3_temp;
30
31 Init();
32 if (first_run==0) { first_run=1;
33 stat[0]=0.0;
34 }
```

```
35
36 for (;;){
37 LookVar();
38 call_IN1(2*M*4,IN1_temp);
39 call_IN2(M*4,IN2_temp);
40
41 { /* start program body */
42 for (i=0; i < M; i++)
43    {
44    OUT1[i*2]   =((float)(IN1[2*i]+IN2[i]))/2.0;
45    OUT1[2*i+1]=((float)(IN1[2*i+1]+IN2[i]))/2.0;
46    OUT2[i*2]   =((float)(IN1[2*i]-IN2[i]))/2.0;
47    OUT2[2*i+1]=((float)(IN1[2*i+1]-IN2[i]))/2.0;
48    for (k=0; k < KMAX; k++)
49      {
50      stat[0]=0.9*stat[0]+0.1*((float)IN1[2*i]);
51      OUT3[i*2]   =stat[0];
52      stat[0]=0.9*stat[0]+0.1*((float)IN1[2*i+1]);
53      OUT3[2*i+1]=stat[0];
54      }
55    }
56 }
57
58 call_OUT1(2*M*4,OUT1_temp);
59 call_OUT2(2*M*4,OUT2_temp);
60 call_OUT3(2*M*4,OUT3_temp);
61   } /* end of endless loop */
62 } /* end of program */
```

C.2.3 DSP Wrapper H-Datei `DspPcIO.h`

```
1 #include "../Tools/CycleBuffer.h"
2 #include "../Tools/FPGA_Boot.h"
3 #include "../Tools/LEDs.h"
4 #include "../Tools/SysTypes.h"
5 #include "../Tools/DSP_CPU_ComAddr.h"
6 #include "../Tools/globalTypesDsp.h"
7
8 #define SMRT_DATA_AVAIL_ID 0
9
10 extern CBUFFER *DPRamTX;
11 extern CBUFFER *DPRamRX;
12
13 #define SMRT_CF_STATE  (*((volatile unsigned int *)0x68020FF8))
14 #define SMRT_DSP_STATE (*((volatile unsigned int *)0x68020FFC))
15 #define SMRT_STATE_UNKNOWN_INIT     2
16 #define SMRT_STATE_NETREADERR       4
17 #define SMRT_STATE_INVALIDCMD       8
18
19 void Init();
20 void LookVar();
```

```
21 void call_IN1(int size, void *data);
22 void call_IN2(int size, void *data);
23
24 void call_OUT1(int size, void *data);
25 void call_OUT2(int size, void *data);
26 void call_OUT3(int size, void *data);
```

C.2.4 DSP Wrapper C-Datei DspPcIO.c

```
1  #include "../Tools/DspPcIO.h"
2
3  CBUFFER *DPRamTX;
4  CBUFFER *DPRamRX;
5
6  void LookVar() {
7    unsigned int VarType = 0, prog = 0;
8    SMRT_COMMAND Cmd;
9
10   while(prog == 0) {
11     for( ;!(VarType=look_dp( DPRamRX )); );
12       switch (VarType) {
13         case SMRT_MEMCOPY_ID:
14         case SMRT_MEMSET_ID:
15           RecvBinaryData(VarType);
16           break;
17         case SMRT_COMMAND_ID:
18           RecDataTFromNet(SMRT_COMMAND_ID, (int*) &Cmd);
19           switch( Cmd.commandcode ) {
20             case SMRT_COMMAND_FLASH:
21               ProgramFlash( (Byte*) Cmd.data[2],
22                             (Byte*) Cmd.data[0],
23                             Cmd.data[1], Cmd.data[3]);
24               break;
25       // insert other commands
26           }
27         default:
28           prog = 1;
29       break;
30       //insert ohter datatypes
31     }
32   }
33 }
34
35 int SendDataToNet( int *data ){
36   register int Len,*Data,WriteLen;
37
38   for( Len = *data&0x00FFFFFF, Data=data; Len;
39        Len-=WriteLen, Data+=WriteLen ){
40     if( (LITTLE_BIG_ENDIAN_INT(CPU_STATE)==MSG_NET_NEW_REG) ){
41       CPU_STATE = 0; // clear MSG
42       return -1;
```

```
43      }
44
45      WriteLen = write_cb( (CB_DAT*)Data, Len, DPRamTX );
46    }
47    return 0;
48  }
49
50  int RecDataTFromNet( int type, int *data ){
51    register int Len,*Data,ReadLen,Head=1;
52
53    for( Len=type&0x00FFFFFF, Data=data; Len;
54         Len-=ReadLen, Data+=ReadLen ){
55      if( (LITTLE_BIG_ENDIAN_INT(CPU_STATE)==MSG_NET_NEW_REG) ){
56        CPU_STATE = 0; // clear MSG
57        return -1;
58      }
59
60      ReadLen = read_cb( (CB_DAT*)Data, Len, DPRamRX );
61      if( Head ){
62        if( *Data!=type )
63          return -2;
64        Head = 0;
65      }
66    }
67    return 0;
68  }
69
70  int RecDataFromNet( int *data ){
71    register int Len,*Data,ReadLen;
72
73    for( ;!(Len=look_dp( DPRamRX )); );
74      for( Len&=0x00FFFFFF, Data=data; Len;
75           Len-=ReadLen, Data+=ReadLen ){
76        ReadLen = read_cb( (CB_DAT*)Data, Len, DPRamRX );
77        if( (LITTLE_BIG_ENDIAN_INT(CPU_STATE)==MSG_NET_NEW_REG) ){
78          CPU_STATE = 0; // clear MSG
79        return -1;
80      }
81    }
82    return 0;
83  }
84
85  void Init() {
86    setGPEN    = 0xFFFF ;
87    setGPDIR   = LED0 + LED1 + LED2;
88    setLEDs &= ~(LED0 + LED1 + LED2);
89    Wait_CPU_Ready();
90    FPGA_Reset_PAR();
91    FPGA_Boot_PAR( (short*)0x60010000 );
92    FPGA_Reset( ALL_RESET );
93    mysleep( 10 );
94    DSP_STATE = 0x00000000;
95    SET_SEND_MODE( SEND_CB );
96    SET_DEBUG_MODE( 0
```

```
97        //  +DEBUG_SND_VAR_INFO
98        //  +DEBUG_SND_NET_BLK_INFO
99        //  +DEBUG_SND_ALL_DATA
100       //  +DEBUG_REC_VAR_INFO
101       //  +DEBUG_REC_NET_BLK_INFO
102       //  +DEBUG_REC_ALL_DATA
103     );
104
105     DPRamTX = DP_RAM_DSP_PC;
106     init_cb(DPRamTX, 0x07FC);
107
108     DPRamRX = DP_RAM_PC_DSP;
109     init_cb(DPRamRX, 0x07FC);
110   }
111
112   void call_IN1(int size, void *data) {
113     MATLAB_DATA1 *tmpData;
114     tmpData = (MATLAB_DATA1*)(((int)data)-4);
115     for( ;RecDataTFromNet(MATLAB_DATA1_ID,(int*)tmpData) < 0 ; );
116   }
117
118   void call_IN2(int size, void *data) {
119     MATLAB_DATA2 *tmpData;
120     tmpData = (MATLAB_DATA2*)(((int)data)-4);
121     for( ;RecDataTFromNet(MATLAB_DATA2_ID,(int*)tmpData) < 0 ; );
122   }
123
124   void Send(void *data) {
125     if(SendDataToNet((int *)data) < 0 )
126       DPRamTX->r_ind = DPRamTX->w_ind; // delete buffer
127     else
128       DPRamRX->w_ind = DPRamRX->r_ind; // delete buffer
129   }
130
131   void call_OUT1(int size, void *data) {
132     MATLAB_DATA3 *tmpData;
133     tmpData = (MATLAB_DATA3*) (((int)data)-4);
134     tmpData->var_id = MATLAB_DATA3_ID;
135     Send(tmpData);
136   }
137
138   void call_OUT2(int size, void *data) {
139     MATLAB_DATA3 *tmpData;
140     tmpData = (MATLAB_DATA3*) (((int)data)-4);
141     tmpData->var_id = MATLAB_DATA3_ID;
142     Send(tmpData);
143   }
144
145   void call_OUT3(int size, void *data) {
146     MATLAB_DATA3 *tmpData;
147     tmpData = (MATLAB_DATA3*) (((int)data)-4);
148     tmpData->var_id = MATLAB_DATA3_ID;
149     Send(tmpData);
150   }
```

C.2.5 DSP GlobalTypes H-Datei `globalTypesDsp.h`

```
#ifndef _GLOBALTYPES
#define _GLOBALTYPES

#include "global.h"

#define ID_TO_SIZE( ID ) (((ID)&0x00FFFFFF)*4)
// size of a variable with this ID in bytes/4
#define ID_TO_SIZE_4( ID ) ((ID)&0x00FFFFFF)
// type of a variable with this ID (0 to 255)
#define ID_TO_TYPE(ID) (((unsigned int)(ID))>>24)

#define M 1000
#define KMAX 1

typedef struct {
  unsigned int  var_id;
  int  data[2*M];
} MATLAB_DATA1;

#define MATLAB_DATA1_ID MAKE_ID( 50, MATLAB_DATA1 )

typedef struct {
  unsigned int  var_id;
  int  data[M];
} MATLAB_DATA2;

#define MATLAB_DATA2_ID MAKE_ID( 51, MATLAB_DATA2 )

typedef struct {
  unsigned int  var_id;
  float  data[2*M];
} MATLAB_DATA3;

#define MATLAB_DATA3_ID MAKE_ID( 52, MATLAB_DATA3 )

#endif //_GLOBALTYPES
```

C.2.6 Simulink S-Funktion Datei `dspIO.cpp`

```
#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME dspIO

#include "simstruc.h"
#include "includes/PcDspIO.h"
#include "includes/globalTypesPc.h"
#include <windows.h>

//static bool isInit = false;

```

```
11 static void mdlInitializeSizes(SimStruct *S)
12 {
13     ssSetNumSFcnParams(S, 0);
14     if (ssGetNumSFcnParams(S) !=
15           ssGetSFcnParamsCount(S)) {
16
17         printf("error: number of parameters inconsistent!\n");
18         return;
19        }
20       else
21        {
22        }
23
24     ssSetNumContStates(S, 0);
25     ssSetNumDiscStates(S,1+1);
26
27     if (!ssSetNumInputPorts(S, 2)) return;
28     ssSetInputPortWidth(S, 0, 2*M);
29     ssSetInputPortDirectFeedThrough(S, 0, 1);
30     ssSetInputPortWidth(S, 1, M);
31     ssSetInputPortDirectFeedThrough(S, 1, 1);
32
33     if (!ssSetNumOutputPorts(S, 3)) return;
34     ssSetOutputPortWidth(S, 0, 2*M);
35     ssSetOutputPortWidth(S, 1, 2*M);
36     ssSetOutputPortWidth(S, 2, 2*M);
37
38     ssSetNumSampleTimes(S, 1);
39     ssSetNumRWork(S, 0);
40     ssSetNumIWork(S, 0);
41     ssSetNumPWork(S, 0);
42     ssSetNumModes(S, 0);
43     ssSetNumNonsampledZCs(S, 0);
44
45     ssSetOptions(S, 0);
46 }
47
48 static void mdlInitializeSampleTimes(SimStruct *S)
49 {
50     ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
51     ssSetOffsetTime(S, 0, 0.0);
52 }
53
54 #define MDL_INITIALIZE_CONDITIONS
55 #if defined(MDL_INITIALIZE_CONDITIONS)
56    static void mdlInitializeConditions(SimStruct *S)
57   {
58
59 real_T *xdisc= ssGetRealDiscStates(S);
60 int run_i;
61
62  //if(!isInit) {
63    //isInit = true;
64    if(Init() != 0)
```

```
65        mexErrMsgTxt("error init board \n");
66   //}
67
68   for (run_i=0; run_i<1+1; run_i++)
69      xdisc[run_i]=0.0;
70
71   }
72
73   #endif /* MDL_INITIALIZE_CONDITIONS */
74
75   #undef MDL_START
76   #if defined(MDL_START)
77     static void mdlStart(SimStruct *S)
78     {
79     }
80   #endif /*  MDL_START */
81
82   static void mdlOutputs(SimStruct *S, int_T tid)
83   {
84   float tmpOUT1[2*M];
85   float tmpOUT2[2*M];
86   float tmpOUT3[2*M];
87   int outCnt = 0;
88   const real_T
89     *const *IN1_temp = ssGetInputPortRealSignalPtrs(S,0);
90   int IN1[2*M];
91   const real_T
92     *const *IN2_temp = ssGetInputPortRealSignalPtrs(S,1);
93   int IN2[M];
94   real_T *OUT1 = ssGetOutputPortRealSignal(S,0);
95   real_T *OUT2 = ssGetOutputPortRealSignal(S,1);
96   real_T *OUT3 = ssGetOutputPortRealSignal(S,2);
97
98   real_T *xdisc= ssGetRealDiscStates(S);
99   float stat[1];
100
101  int i,k;
102  int run_i;
103  char first_run;
104
105  for (run_i=0; run_i<1; run_i++)
106      stat[run_i]=(float)xdisc[run_i];
107
108      first_run=(char)xdisc[run_i];
109
110
111  if (first_run==0) { first_run=1;
112  stat[0]=0.0;
113  }
114
115  for (run_i=0; run_i<2*M; run_i++)
116     IN1[run_i]=(int)*IN1_temp[run_i];
117  for (run_i=0; run_i<M; run_i++)
118     IN2[run_i]=(int)*IN2_temp[run_i];
```

```
119
120 {
121 Sleep(20);
122 if(SendData_IN1(sizeof(IN1),&IN1[0]) != 0 )
123  mexErrMsgTxt("send failure.\n");
124 Sleep(20);
125 if(SendData_IN2(sizeof(IN2),&IN2[0]) != 0 )
126  mexErrMsgTxt("send failure.\n");
127
128 Sleep(20);
129 if(ReceiveData_OUT1(sizeof(tmpOUT1),&tmpOUT1[0]) != 0)
130  mexErrMsgTxt("receive failure.\n");
131 Sleep(20);
132 if(ReceiveData_OUT2(sizeof(tmpOUT2),&tmpOUT2[0]) != 0)
133  mexErrMsgTxt("receive failure.\n");
134 Sleep(20);
135 if(ReceiveData_OUT3(sizeof(tmpOUT3),&tmpOUT3[0]) != 0)
136  mexErrMsgTxt("receive failure.\n");
137
138 for(outCnt=0; outCnt < (2*M);outCnt++)
139  OUT1[outCnt] = (double)tmpOUT1[outCnt];
140
141 for(outCnt=0; outCnt < (2*M);outCnt++)
142  OUT2[outCnt] = (double)tmpOUT2[outCnt];
143
144 for(outCnt=0; outCnt < (2*M);outCnt++)
145  OUT3[outCnt] = (double)tmpOUT3[outCnt];
146
147 }
148 for (run_i=0; run_i <1; run_i++)
149      xdisc[run_i]=(real_T)stat[run_i];
150 xdisc[run_i]=(real_T)first_run;
151 }
152
153 #undef MDL_UPDATE
154 #if defined(MDL_UPDATE)
155    static void mdlUpdate(SimStruct *S, int_T tid)
156    {
157    }
158 #endif /* MDL_UPDATE */
159
160 #undef MDL_DERIVATIVES
161 #if defined(MDL_DERIVATIVES)
162    static void mdlDerivatives(SimStruct *S)
163    {
164    }
165 #endif /* MDL_DERIVATIVES */
166
167 static void mdlTerminate(SimStruct *S)
168 {
169   if( Close() != 0 ) {
170      mexErrMsgTxt("\nconnetion could not be closed!");
171   }
172 }
```

```
173
174
175 #ifdef  MATLAB_MEX_FILE
176 #include "simulink.c"
177 #else
178 #include "cg_sfun.h"
179 #endif
```

C.2.7 PC Wrapper H-Datei `PcDspIO.h`

```
1  #ifndef _PCDSPIO
2  #define _PCDSPIO
3  #ifdef __cplusplus
4  extern "C" {
5  #endif
6
7  int Init();
8  int Close();
9
10 int SendData_IN1(int size, void *data);
11 int SendData_IN2(int size, void *data);
12
13 int ReceiveData_OUT1(int size, void *data);
14 int ReceiveData_OUT2(int size, void *data);
15 int ReceiveData_OUT3(int size, void *data);
16
17 #ifdef __cplusplus
18 }
19 #endif
20 #endif //_PCDSPIO
```

C.2.8 PC Wrapper C-Datei `PcDspIO.c`

```
1  #include <stdio.h>
2  #include "simstruc.h"
3  #include "globalTypesPc.h"
4  #include "comFunct.h"
5  static int gConNr = -1;
6
7  int Init() {
8    RTS_CONNECTION     *Con=NULL;
9
10   if(StartUp(&gConNr,RACKID,&Con,REGPORT,
11              DATAPORT,RECEIVEBUFFER) != 0)
12     return -1;
13   else
14     return 0;
15 }
16
```

```
17 int Close() {
18   if(CloseConnection() != 0)
19     return -1;
20   else
21     return 0;
22 }
23
24 int SendData_IN1(int size, void *data) {
25   MATLAB_DATA1 tmpData;
26   tmpData.var_id = MATLAB_DATA1_ID;
27   memcpy(tmpData.data,data,size);
28   return SendVariable(&tmpData,
29                       ID_TO_SIZE(MATLAB_DATA1_ID),gConNr);
30 }
31
32 int SendData_IN2(int size, void *data) {
33   MATLAB_DATA2 tmpData;
34   tmpData.var_id = MATLAB_DATA2_ID;
35   memcpy(tmpData.data,data,size);
36   return SendVariable(&tmpData,
37                       ID_TO_SIZE(MATLAB_DATA2_ID),gConNr);
38 }
39
40 int ReceiveData_OUT1(int size, void *data) {
41   MATLAB_DATA3 tmpData;
42
43   if( RTS_DataAvail( 5000000 ) == 1) {
44     if ( ReceiveVariable(&tmpData, ID_TO_SIZE(MATLAB_DATA3_ID),
45                          &gConNr) == 0 )
46       memcpy(data,&tmpData.data[0],size);
47     else
48       return -1;
49   } else
50     return -2;
51
52   return 0;
53 }
54
55 int ReceiveData_OUT2(int size, void *data) {
56   MATLAB_DATA3 tmpData;
57
58   if( RTS_DataAvail( 5000000 ) == 1) {
59     if ( ReceiveVariable(&tmpData, ID_TO_SIZE(MATLAB_DATA3_ID),
60                          &gConNr) == 0 )
61       memcpy(data,&tmpData.data[0],size);
62     else
63       return -1;
64   } else
65     return -2;
66
67   return 0;
68 }
69
70
```

```
71 int ReceiveData_OUT3(int size, void *data) {
72   MATLAB_DATA3 tmpData;
73
74   if( RTS_DataAvail( 5000000 ) == 1) {
75     if ( ReceiveVariable(&tmpData, ID_TO_SIZE(MATLAB_DATA3_ID),
76                          &gConNr) == 0 )
77       memcpy(data,&tmpData.data[0],size);
78     else
79       return -1;
80   } else
81     return -2;
82
83   return 0;
84 }
```

C.2.9 PC GlobalTypes H-Datei `globalTypesPc.h`

```
1  #ifndef _GLOBALTYPES
2  #define _GLOBALTYPES
3
4  #define MAKE_ID( type , var )
5                  ((((unsigned int)(type))<<24)+sizeof(var)/4)
6  // size of a variable with this ID in bytes
7  #define ID_TO_SIZE( ID ) (((ID)&0x00FFFFFF)*4)
8  // size of a variable with this ID in bytes/4
9  #define ID_TO_SIZE_4( ID ) ((ID)&0x00FFFFFF)
10 // type of a variable with this ID (0 to 255)
11 #define ID_TO_TYPE(ID) (((unsigned int)(ID))>>24)
12
13 #define M 1000
14 #define KMAX 1
15
16 typedef struct {
17   unsigned int  var_id;
18   int  data[2*M];
19 } MATLAB_DATA1;
20
21 #define MATLAB_DATA1_ID MAKE_ID( 50, MATLAB_DATA1 )
22
23 typedef struct {
24   unsigned int  var_id;
25   int  data[M];
26 } MATLAB_DATA2;
27
28 #define MATLAB_DATA2_ID MAKE_ID( 51, MATLAB_DATA2 )
29
30 typedef struct {
31   unsigned int  var_id;
32   float  data[2*M];
33 } MATLAB_DATA3;
34
35 #define MATLAB_DATA3_ID MAKE_ID( 52, MATLAB_DATA3 )
36
37 #endif //_GLOBALTYPES
```

C.3 Sundance Dateien

C.3.1 Projektdatei dspSMT.CFG

```
! ******************************************************
! Sundance Multiprocessor Technology Ltd.
! NOTE: You need to specify the type of TIM you are using.
! See the 3L Diamond documentation for more details.
! ******************************************************
processor root SMT365_8_2
task dspSMT data=7M
place dspSMT root
```

C.3.2 Projektdatei MAKEFILE.smt

```
# ****************************************************
# Sundance Multiprocessor Technology Ltd.
# ****************************************************
# To use this makefile, type nmake from a
# command line prompt.
# ****************************************************
dspSMT.app: dspSMT.tsk dspSMT.cfg MAKEFILE
config /a dspSMT.cfg dspSMT.app

dspSMT.tsk: dspSMT.obj DspPcIO.obj MAKEFILE
c64slink dspSMT DspPcIO

dspSMT.obj: dspSMT.c MAKEFILE
c64c dspSMT.c DspPcIO.h DspPcIO.c -g
```

C.3.3 DSP Main-Datei dspSMT.c

```
1  #define M 100
2  #define KMAX 1000
3
4  #include <alt.h>
5  #include "DspPcIO.h"
6
7  void main(int argc, char *argv[], char *envp[],CHAN *in_ports[],
8            int ins, CHAN *out_ports[], int outs)
9  {
10 char IN1_temp[2*M*4];
11 int *IN1 = (int *) IN1_temp;
12 char IN2_temp[M*4];
13 int *IN2 = (int *) IN2_temp;
14 char OUT1_temp[2*M*4];
```

```
15 float *OUT1 = (float *) OUT1_temp;
16 char OUT2_temp[2*M*4];
17 float *OUT2 = (float *) OUT2_temp;
18 char OUT3_temp[2*M*4];
19 float *OUT3 = (float *) OUT3_temp;
20
21 int i,k;
22 static float stat[1];
23 static char first_run=0;
24
25 Init();
26 if (first_run==0) { first_run=1;
27
28 stat[0]=0.0;
29 }
30
31 for (;;){
32 call_IN1(2*M*4,IN1_temp);
33 call_IN2(M*4,IN2_temp);
34
35 { /* start program body */
36 for (i=0; i<M; i++)
37    {
38    OUT1[i*2]   =((float)(IN1[2*i]+IN2[i]))/2.0;
39    OUT1[2*i+1]=((float)(IN1[2*i+1]+IN2[i]))/2.0;
40    OUT2[i*2]   =((float)(IN1[2*i]-IN2[i]))/2.0;
41    OUT2[2*i+1]=((float)(IN1[2*i+1]-IN2[i]))/2.0;
42    for (k=0; k<KMAX; k++)
43       {
44       stat[0]=0.9*stat[0]+0.1*((float)IN1[2*i]);
45       OUT3[i*2]   =stat[0];
46       stat[0]=0.9*stat[0]+0.1*((float)IN1[2*i+1]);
47       OUT3[2*i+1]=stat[0];
48       }
49    }
50 }
51
52 call_OUT1(2*M*4,OUT1_temp);
53 call_OUT2(2*M*4,OUT2_temp);
54 call_OUT3(2*M*4,OUT3_temp);
55   } /* end of endless loop */
56 } /* end of program */
```

C.3.4 DSP Wrapper H-Datei DspPcIO.h

```
1  #include <stdlib.h>
2  #include <SMT_BSI.h>
3  #include <link.h>
4
5  void Init();
6  void call_IN1(int size, void *data);
7  void call_IN2(int size, void *data);
8
9  void call_OUT1(int size, void *data);
10 void call_OUT2(int size, void *data);
11 void call_OUT3(int size, void *data);
```

C.3.5 DSP Wrapper C-Datei DspPcIO.c

```
1  #include "DspPcIO.h"
2
3  SC6xSmtI     * pBSM =0;
4  int            chan=0;
5
6  void Init() {
7    // initialize the board
8    if (OpenBoardServices(&pBSM)) {
9      link_out_word(2,3);
10     exit(1);
11   }
12
13   if (Smt_HscInit(pBSM)) {
14     link_out_word(3,3);
15     exit(1);
16   }
17
18   link_out_word(0,3);     // Send ok to Host
19 }
20
21 void OpenPci() {
22   // wait until the host locks PCI memory
23   Smt_HscControl(pBSM, chan, HSC_OpenPci, 0);
24 }
25
26 void ClosePci() {
27   Smt_HscControl(pBSM, chan, HSC_ClosePci, 0);
28 }
29
30 void call_IN1(int size, void *data) {
31   int readLen = 0;
32
33   OpenPci();
34   readLen = Smt_HscPciRead( pBSM, chan, size, data );
35   ClosePci();
```

```
36    link_out_word(readLen , 3);
37  }
38
39  void call_IN2(int size, void *data) {
40    int readLen = 0;
41
42    OpenPci();
43    readLen = Smt_HscPciRead( pBSM, chan, size, data );
44    ClosePci();
45    link_out_word(readLen , 3);
46  }
47
48  void call_OUT1(int size, void *data) {
49    int writeLen = 0;
50
51    OpenPci();
52    writeLen = Smt_HscPciWrite( pBSM, chan, size, data );
53    ClosePci();
54    link_out_word(writeLen , 3);
55  }
56
57  void call_OUT2(int size, void *data) {
58    int writeLen = 0;
59
60    OpenPci();
61    writeLen = Smt_HscPciWrite( pBSM, chan, size, data );
62    ClosePci();
63    link_out_word(writeLen , 3);
64  }
65
66  void call_OUT3(int size, void *data) {
67    int writeLen = 0;
68
69    OpenPci();
70    writeLen = Smt_HscPciWrite( pBSM, chan, size, data );
71    ClosePci();
72    link_out_word(writeLen , 3);
73  }
```

C.3.6 Simulink S-Funktion Datei dspIO.cpp

```
1  #define S_FUNCTION_LEVEL 2
2  #define S_FUNCTION_NAME dspIO
3
4  #include "simstruc.h"
5  #include "includes/PcDspIO.h"
6
7  static bool isInit = false;
8
9  #define M 100
10 #define KMAX 1000
11
12 static void mdlInitializeSizes(SimStruct *S)
13 {
14     ssSetNumSFcnParams(S, 0);
15     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
16         printf("error: number of parameters inconsistent!\n");
17         return;
18       }
19      else
20       {
21       }
22
23     ssSetNumContStates(S, 0);
24     ssSetNumDiscStates(S,1+1);
25
26     if (!ssSetNumInputPorts(S, 2)) return;
27     ssSetInputPortWidth(S, 0, 2*M);
28     ssSetInputPortDirectFeedThrough(S, 0, 1);
29     ssSetInputPortWidth(S, 1, M);
30     ssSetInputPortDirectFeedThrough(S, 1, 1);
31
32     if (!ssSetNumOutputPorts(S, 3)) return;
33     ssSetOutputPortWidth(S, 0, 2*M);
34     ssSetOutputPortWidth(S, 1, 2*M);
35     ssSetOutputPortWidth(S, 2, 2*M);
36
37     ssSetNumSampleTimes(S, 1);
38     ssSetNumRWork(S, 0);
39     ssSetNumIWork(S, 0);
40     ssSetNumPWork(S, 0);
41     ssSetNumModes(S, 0);
42     ssSetNumNonsampledZCs(S, 0);
43
44     ssSetOptions(S, 0);
45 }
46
47 static void mdlInitializeSampleTimes(SimStruct *S)
48 {
49     ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
50     ssSetOffsetTime(S, 0, 0.0);
51 }
52
```

```
53 #define MDL_INITIALIZE_CONDITIONS
54 #if defined(MDL_INITIALIZE_CONDITIONS)
55   static void mdlInitializeConditions(SimStruct *S)
56   {
57
58 real_T *xdisc= ssGetRealDiscStates(S);
59 int run_i;
60
61   if(!isInit) {
62     isInit = true;
63     int error = Init();
64     switch(error) {
65       case 0:
66         break;
67       case -1:
68         mexErrMsgTxt("Could not open SmtDrv library.\n");
69         break;
70       case -2:
71         mexErrMsgTxt("No Sundance carrier pboards found.\n");
72         break;
73       case -3:
74         mexErrMsgTxt("Could not open the carrier pboard.\n");
75         break;
76       case -4:
77         mexErrMsgTxt("Could not get interface
78                              to default handler.\n");
79         break;
80       case -5:
81         mexErrMsgTxt("error Reset TIM or download app.\n");
82         break;
83       case -6:
84         mexErrMsgTxt("DSP not started up correctly.\n");
85         break;
86       default:
87         mexErrMsgTxt("unknown Init() error.\n");
88         break;
89     }
90   }
91
92 for (run_i=0; run_i<1+1; run_i++)
93    xdisc[run_i]=0.0;
94
95 }
96
97 #endif /* MDL_INITIALIZE_CONDITIONS */
98
99 #undef MDL_START
100 #if defined(MDL_START)
101   static void mdlStart(SimStruct *S)
102   {
103   }
104 #endif /*  MDL_START */
105
106 static void mdlOutputs(SimStruct *S, int_T tid)
```

```
107 {
108 float tmpOUT1[2*M];
109 float tmpOUT2[2*M];
110 float tmpOUT3[2*M];
111 int outCnt = 0;
112 const real_T
113   *const *IN1_temp = ssGetInputPortRealSignalPtrs(S,0);
114 int IN1[2*M];
115 const real_T
116   *const *IN2_temp = ssGetInputPortRealSignalPtrs(S,1);
117 int IN2[M];
118 real_T *OUT1 = ssGetOutputPortRealSignal(S,0);
119 real_T *OUT2 = ssGetOutputPortRealSignal(S,1);
120 real_T *OUT3 = ssGetOutputPortRealSignal(S,2);
121
122 real_T *xdisc= ssGetRealDiscStates(S);
123 float stat[1];
124
125 int i,k;
126 int run_i;
127 char first_run;
128
129 for (run_i=0; run_i<1; run_i++)
130     stat[run_i]=(float)xdisc[run_i];
131
132     first_run=(char)xdisc[run_i];
133
134
135 if (first_run==0) { first_run=1;
136 stat[0]=0.0;
137 }
138
139 for (run_i=0; run_i<2*M; run_i++)
140    IN1[run_i]=(int)*IN1_temp[run_i];
141 for (run_i=0; run_i<M; run_i++)
142    IN2[run_i]=(int)*IN2_temp[run_i];
143
144 {
145 if(SendData_IN1(sizeof(IN1),&IN1[0]) != 0 )
146   mexErrMsgTxt("send failure.\n");
147 if(SendData_IN2(sizeof(IN2),&IN2[0]) != 0 )
148   mexErrMsgTxt("send failure.\n");
149
150 if(ReceiveData_OUT1(sizeof(tmpOUT1),&tmpOUT1[0]) != 0)
151   mexErrMsgTxt("receive failure.\n");
152 if(ReceiveData_OUT2(sizeof(tmpOUT2),&tmpOUT2[0]) != 0)
153   mexErrMsgTxt("receive failure.\n");
154 if(ReceiveData_OUT3(sizeof(tmpOUT3),&tmpOUT3[0]) != 0)
155   mexErrMsgTxt("receive failure.\n");
156
157 for(outCnt=0; outCnt ¡ (2*M);outCnt++)
158   OUT1[outCnt] = (double)tmpOUT1[outCnt];
159
160 for(outCnt=0; outCnt ¡ (2*M);outCnt++)
```

```
161     OUT2[outCnt] = (double)tmpOUT2[outCnt];
162
163   for(outCnt=0; outCnt ¡ (2*M);outCnt++)
164     OUT3[outCnt] = (double)tmpOUT3[outCnt];
165
166   }
167   for (run_i=0; run_i<1; run_i++)
168       xdisc[run_i]=(real_T)stat[run_i];
169   xdisc[run_i]=(real_T)first_run;
170   }
171
172   #undef MDL_UPDATE
173   #if defined(MDL_UPDATE)
174     static void mdlUpdate(SimStruct *S, int_T tid)
175     {
176     }
177   #endif /* MDL_UPDATE */
178
179   #undef MDL_DERIVATIVES
180   #if defined(MDL_DERIVATIVES)
181     static void mdlDerivatives(SimStruct *S)
182     {
183     }
184   #endif /* MDL_DERIVATIVES */
185
186   static void mdlTerminate(SimStruct *S)
187   {
188     //Close();
189   }
190
191   #ifdef MATLAB_MEX_FILE
192   #include "simulink.c"
193   #else
194   #include "cg_sfun.h"
195   #endif
```

C.3.7 PC Wrapper H-Datei `PcDspIO.h`

```
1   #ifndef _PCDSPIO
2   #define _PCDSPIO
3
4   int Init();
5   void Close();
6
7   int SendData_IN1(int size, void *data);
8   int SendData_IN2(int size, void *data);
9
10  int ReceiveData_OUT1(int size, void *data);
11  int ReceiveData_OUT2(int size, void *data);
12  int ReceiveData_OUT3(int size, void *data);
13  #endif //_PCDSPIO
```

C.3.8 PC Wrapper C-Datei PcDspIO.cpp

```
1  #include "PcDspIO.h"
2  #pragma comment(lib, "includes\\SmtDrv.lib")
3  #include "HscDflt.h"
4
5  int SendVariable(void *inData, int size);
6  int ReceiveVariable(void *outData, int size);
7
8  static IFHw          * pBoard  = 0;
9  static HscDflt       * pHSC    = 0;
10
11 int Init() {
12   int status = 0;
13
14   SMTRet ret = SmtOpen();
15   if (ret!=SMT_OK)
16     return -1;
17
18   if (SmtGetBoardCount()==0)
19     return -2;
20
21   pBoard = SmtOpenBoard(0);
22   if (!pBoard)
23     return -3;
24
25   pHSC = (HscDflt *) pBoard->GetHandler(0);
26   if (!pHSC)
27     return -4;
28
29   try {
30     pBoard->ResetTIMs();
31     pBoard->BinaryLoad("DspProject/dspSMT.app");
32     /* Make sure the DSP started up correctly */
33     pBoard->CpRead(&status, sizeof(int));
34     if (status)
35       return -6;
36   }catch(SMTExc &e) {
37     e.GetError();
38     return -5;
39   }
40
41   return 0;
42 }
43
44 void Close() {
45   SmtCloseBoard(0);
46 }
47
48 int SendVariable(void *inData, int size) {
49   unsigned int  * lock    = 0;
50   char *data;
51   unsigned int req = 0;
52   int status = -1;
```

```
53
54    data = new char[size];
55    memcpy(data,inData,size);
56    lock = (unsigned int*) pHSC->PciOpenSync(size, data);
57    if (!lock)  {
58      delete [] data;
59      return -1;
60    }
61    req = pHSC->PciPutSync();
62    if(req != size) {
63      delete [] data;
64    return -2;
65    }
66    pHSC->PciPutDone(req);
67    pHSC->PciCloseSync();
68    pBoard->CpRead(&status, sizeof(int));
69    if(status != size) {
70      delete [] data;
71      return -3;
72    }
73    delete [] data;
74    return 0;
75  }
76
77  int ReceiveVariable(void *outData, int size) {
78    unsigned int  * lock    = 0;
79    unsigned int req = 0;
80    char *data;
81    int status = -1;
82    data = new char[size];
83    lock = (unsigned int*) pHSC->PciOpenSync(size, data);
84    if (!lock) {
85      delete [] data;
86      return -1;
87    }
88    req = pHSC->PciGetSync();
89    if(req != size) {
90      delete [] data;
91      return -2;
92    }
93    memcpy(outData,data,size);
94    pHSC->PciGetDone(req);
95    pHSC->PciCloseSync();
96    pBoard->CpRead(&status, sizeof(int));
97    if(status != size) {
98      delete [] data;
99      return -3;
100   }
101   delete [] data;
102   return 0;
103 }
104
105 int SendData_IN1(int size, void *data) {
106   return SendVariable(data,size);
```

```
107 }
108
109 int SendData_IN2(int size, void *data) {
110   return SendVariable(data,size);
111 }
112
113 int ReceiveData_OUT1(int size, void *data) {
114   return ReceiveVariable(data,size);
115 }
116
117 int ReceiveData_OUT2(int size, void *data) {
118   return ReceiveVariable(data,size);
119 }
120
121 int ReceiveData_OUT3(int size, void *data) {
122   return ReceiveVariable(data,size);
123 }
```

Anhang D

Befehle für das Hardware-Interface

Anmerkungen: Ist bei den Parametern ein *Default* Wert angeben, müssen diese nicht mehr übergeben werden. Die Möglichkeiten, die sich dadurch ergeben, werden als alternative Aufrufe angeführt.

Bei den nachfolgenden Befehlen sind Fehlermeldungen nur angeführt, wenn es sich um keine Standardfehlermeldungen handelt. Bei diesen Fehlermeldungen handelt es sich um eine ungültige Anzahl von übergebenen Argumenten oder um Argumente vom falschen Datentyp. Unter falschen Datentypen kann verstanden werden, dass anstatt eines Skalar ein String und umgekehrt übergeben wird. Ein String-Datentyp wird nachfolgend mittels `'string'` gekennzeichnet.

D.1 Befehle und deren Parameter für comRTS

D.1.1 con2Board

Stellt eine Verbindung zur Hardware her. Dieser Befehl erfüllt die zusammengefasste Funktionalität der Befehle *open*, *sendregcmd* und *waitforregcon*. Er kann Anstelle der vorhin genannten Befehle verwendet werden.

Aufruf

```
conNr = comRTS('con2Board', rackid, dataport,
               regport, buffsize);
```

Alternative Aufrufe

```
conNr = comRTS('con2Board');
conNr = comRTS('con2Board', rackID);
```

Parameter

`rackid`	Rack Identifier. Default 10
`dataport`	Sende- und Empfangsport für den Datenaustausch. Default 5000
`regport`	Sende- und Empfangsport für die Registrierung. Default 5001
`buffsize`	Größe des Empfangsbuffers für UDP-Pakete. Default 1 000 000 Byte

Fehlermeldungen

`error open connection.`	Verbindung konnte nicht geöffnet werden.
`error set receivebuffer.`	Empfangsbuffer konnte nicht initialisiert werden.
`error SendRegCmd.`	Fehler beim Senden des Registrierungskommandos.
`error WaitForRegCon.`	Verbindung konnte nicht registriert werden.

Rückgabewert

Liefert die aktuelle Nummer der Verbindung zurück.

D.1.2 send

Um Daten richtig senden zu können muss, wie schon öfter erwähnt, vor den Daten ein *Variablen Identifier* vorhanden sein (siehe Abschnit 4.3). Dieser kann vom Benutzer in Matlab oder von der Sendefunktion generiert werden. Bei Verwendung der ersten Variante muss darauf geachtet werden, dass die Größe der Struktur in Matlab und C unterschiedlich ist. In Matlab werden mehr Bytes für das Speichern eines Datentyp benötigt als in C. Für ein fehlerfreies Senden wird die Größe im C-Format benötigt. Das nachfolgende Beispiel soll dies verdeutlichen. Wie aus Tabelle D.1 hervorgeht, benötigt

	Matlab	*C*

Daten	`MATLAB_DATA.var_id = uint32(0);`
	`MATLAB_DATA.data = [1 2 3 4 5];`
		`typedef struct {`
		`unsigned int var_id;`
		`double data[5]; }`

Speicher	292 Bytes	44 Bytes

Tabelle D.1: Beispiel zum unterschiedlichen Speicherbedarf von Matlab und C.

Matlab um 248 Bytes mehr, um die selben Daten zu speichern. Um nun die

benötigte Größe zu erhalten, müssen pro Element in der Struktur 124 Bytes abgezogen werden.

Um diese nicht unter Matlab berechnen zu müssen, kann die *var_id* auch durch die *send-Funktion* berechnet werden. Dafür ist es notwendig den Typ zu übergeben. Die *var_id* wird dann automatisch, vor den zu sendenden Daten, eingefügt.

Aufruf

```
saveNr = comRTS('send', data, type, 'save');
saveNr = comRTS('send', data, 'save');
```

Alternative Aufrufe

```
comRTS('send', data);
comRTS('send', data, type);
```

Parameter

`data`	Zu sendende Daten.
`type`	Typ zur Berechnung der *var_id*.
`'save'`	Struktur von Daten wird gespeichert.

Rückgabewert

Wird die `'save'` Option genutzt, liefert die Funktion die Speicherstelle zurück.

D.1.3 receive

Dieser Befehl dient zum Empfangen von Daten.

Aufruf

```
data = comRTS('receive', structNr, noOfElements, timeout);
data = comRTS('receive', 'datatyp', timeout);
```

Alternative Aufrufe

```
data = comRTS('receive', structNr, noOfElements);
data = comRTS('receive', 'datatyp');
```

Parameter

`structNr`	An welcher Speicherstelle befindet sich die Struktur.
`noOfElements`	Anzahl wieviele Elemente einer bestimmten Struktur empfangen werden.
`'datatyp'`	In welchen Datentypen sollen die Daten umgewandelt werden. Unterstützte Datentypen: double, single, int8, uint8 int16, uint16, int32, uint32
`timeout`	Wie lange soll auf die Daten gewartet werden. Default 100000 μs

Fehlermeldungen

`no data available.`	Innerhalb des *timeout* wurden keine Daten empfangen.
`receive error.`	Fehler beim Empfangen der Daten.

Rückgabewert

Empfangene Daten.

D.1.4 saveStruct

Mittels dieser Funktion ist es möglich eine Struktur vor dem Senden zu speichern.

Aufruf

```
saveNr = comRTS('saveStruct', data);
```

Parameter

`data`	Datenstruktur die gespeichert werden soll.

Fehlermeldung

`to many structs saved.`	Zu viele Strukturen gespeichert.

Rückgabewert

Liefert die Nummer der Speicherstelle zurück.

D.1.5 close

Beendet eine bestehende Verbindung zur Hardware.

Aufruf

```
comRTS('close');
```

Fehlermeldung

`unable to close connection.`	Verbindung konnte nicht beendet werden.

D.1.6 clearStruct

Löschen aller gespeicherten Strukturen.

Aufruf

```
comRTS('clearStruct');
```

D.1.7 resetBoard

Schließen und erneutes Öffnen einer Verbindung. Ersetzt die Befehle *close* und ein darauffolgendes *con2Board*.

Aufruf

```
conNr = comRTS('resetBoard', rackid, dataport,
               regport, buffsize);
```

Alternative Aufrufe

```
conNr = comRTS('resetBoard');
conNr = comRTS('resetBoard', rackID);
```

Parameter

`rackid`	Rack Identifier. Default 10
`dataport`	Sende und Empfangsport für die Registrierung. Default 5000
`regport`	Sende und Empfangsport für Datenaustausch. Default 5001
`buffsize`	Größe des Empfangsbuffers für UDP- Pakete. Default 1 000 000 Byte

Fehlermeldungen

`error open connection.`	Verbindung konnte nicht geöffnet werden.
`error set receivebuffer.`	Empfangsbuffer konnte nicht initialisiert werden.
`error SendRegCmd.`	Fehler beim Senden des Registrierungskommandos.
`error close connection.`	Verbindung konnte nicht beendet werden.
`error WaitForRegCon.`	Verbindung konnte nicht registriert werden.

Rückgabewert

Liefert die aktuelle Nummer der Verbindung zurück.

D.1.8 open

Öffnet eine Verbindung zur Hardware.

Aufruf

```
err = comRTS('open', dataport, regport);
```

Alternativer Aufruf

```
err = comRTS('open');
```

Parameter

`dataport`	Sende und Empfangsport für die Registrierung. Default 5000
`regport`	Sende und Empfangsport für Datenaustausch. Default 5001

Rückgabewert

Errorcode bei einem Fehler.

D.1.9 sendregcmd

Sendet Registrierungskommando an die Hardware.

Aufruf

```
err = comRTS('sendregcmd', rackID);
```

Alternative Aufrufe

```
err = comRTS('sendregcmd');
```

Parameter

`rackID`	Rack Identifier. Default 10

Fehlermeldungen

`first open connection.`	Es wurde noch keine Verbindung geöffnet.
`error sending registration command.`	Fehler beim Senden des Registrierungskommandos.

D.1.10 waitforregcon

Wartet auf die Registrierung der Hardware.

Aufruf

```
error = comRTS('waitforregcon');
```

Fehlermeldungen

`Send registration command first.`	Es wurde noch kein Registrierungskommando gesendet.
`error waiting for registration.`	Fehler beim Warten auf die Registrierung der Hardware.

D.2 Befehle und deren Parameter für comSMT

D.2.1 init

Dieser Befehl dient zum Initialisieren der Hardware. Es wird eine Verbindung hergestellt und ein Programm auf das Modul geladen.

Aufruf

```
comSMT('init','appName');
```

Parameter

`'appName'` Dateiname und Pfad der `.app` Datei.

Fehlermeldungen

`error during initialize.` Fehler beim Initialisieren der Hardware.

D.2.2 loadProg

Programmiert eine `.app` Datei bei bestehender Verbindung.

Aufruf

```
comSMT('loadProg','appName');
```

Parameter

`appName` Dateiname und Pfad der `.app` Datei.

Fehlermeldung

`error downloading file.` Angegebene Datei konnte nicht programmiert werden.

D.2.3 resettim

Resetiert die vorhandenen TIMs.

Aufruf

```
comSMT('resettim');
```

Fehlermeldung

`error reset TIMs.` TIMs konnten nicht resetiert werden.

D.2.4 resetboard

Resetiert die gesamte Hardware.

Aufruf

```
comSMT('resetboard');
```

Fehlermeldung

`error reset Board.`	Fehler beim Resetieren der gesamten Hardware.

D.2.5 cpwrite

Sendet Daten über das *comport*.

Aufruf

```
comSMT('cpwrite', data);
saveNr = comSMT('cpwrite', data, 'save');
```

Parameter

`data`	Daten die über comport übertragen werden.
`'save'`	Datenstruktur wird gespeichert.

Fehlermeldungen

`error comport write.`	Fehler beim Schreiben auf das *comport*.
`to many structs saved.`	Zu viele Datenstrukturen gespeichert.

Rückgabewert

Wird die `'save'` Option genutzt, liefert die Funktion die Speicherstelle zurück.

D.2.6 sendpci

Datenaustausch über den PCI-Bus.

Aufruf

```
comSMT('sendpci', data);
saveNr = comSMT('sendpci', data, 'save');
```

Parameter

`data`	Daten die über PCI-Bus ausgetauscht werden.
`'save'`	Datenstruktur wird gespeichert.

Fehlermeldungen

`error pci write.`	Fehler beim Schreiben auf den PCI-Bus.
`to many structs saved.`	Zu viele Datenstrukturen gespeichert.

Rückgabewert

Wird die `'save'` Option genutzt, liefert die Funktion die Speicherstelle zurück.

D.2.7 sendsram

Datenaustausch über SRAM.

Aufruf

```
comSMT('sendsram', data);
saveNr = comSMT('sendsram', data, 'save');
```

Parameter

`data`	Daten die über SRAM getauscht werden.
`'save'`	Datenstruktur wird gespeichert.

Fehlermeldungen

`error sram write.`	Fehler beim Schreiben auf das SRAM.
`to many structs saved.`	Zu viele Datenstrukturen gespeichert.

Rückgabewert

Wird die `'save'` Option genutzt, liefert die Funktion die Speicherstelle zurück.

D.2.8 cpread

Iniziert einen Lesevorgang vom *comport*.

Aufruf

```
data = comSMT('cpread', strcutNr, noOfElem);
data = comSMT('cpread', 'datatype', noOfElem);
```

Parameter

`structNr`	An welcher Speicherstelle befindet sich die Struktur.
`noOfElements`	Anzahl wieviele Elemente einer bestimmten Struktur empfangen werden.
`'datatyp'`	In welchen Datentypen sollen die Daten umgewandelt werden. Unterstützte Datentypen: double, single, int8, uint8 int16, uint16, int32, uint32

Fehlermeldung

`error comport read.` Fehler beim Lesen vom *comport*.

Rückgabewert

Umgewandelte Daten.

D.2.9 receivepci

Datenaustausch über PCI-Bus.

Aufruf

```
data = comSMT('receivepci', 'datatyp', noOfElem);
data = comSMT('receivepci', structNr, noOfElem);
```

Parameter

`structNr`	An welcher Speicherstelle befindet sich die Struktur.
`noOfElements`	Anzahl wieviele Elemente einer bestimmten Struktur empfangen werden.
`'datatyp'`	In welchen Datentypen sollen die Daten umgewandelt werden. Unterstützte Datentypen: double, single, int8, uint8 int16, uint16, int32, uint32

Fehlermeldung

`error pci read.` Fehler beim Lesen vom PCI-Bus.

Rückgabewert

Umgewandelte Daten.

D.2.10 receivesram

Datenaustausch über SRAM.

Aufruf

```
data = comSMT('receivesram', 'datatyp', noOfElem);
data = comSMT('receivesram', structNr, noOfElem);
```

Parameter

`structNr`	An welcher Speicherstelle befindet sich die Struktur.
`noOfElements`	Anzahl wieviele Elemente einer bestimmten Struktur empfangen werden.
`'datatyp'`	In welchen Datentypen sollen die Daten umgewandelt werden. Unterstützte Datentypen: double, single, int8, uint8 int16, uint16, int32, uint32

Fehlermeldung

`error sram read.`	Fehler beim Lesen vom SRAM.

Rückgabewert

Umgewandelte Daten.

D.2.11 close

Schließt das Interface zur Hardware.

Aufruf

```
comSMT('close');
```

D.2.12 saveStruct

Mittels dieser Funktion ist es möglich eine Struktur vor dem Senden zu speichern.

Aufruf

```
saveNr = comSMT('saveStruct', data);
```

Parameter

`data`	Datenstruktur die gespeichert werden soll.

Fehlermeldung

`to many structs saved.`	Zu viele Strukturen gespeichert.

Rückgabewert

Liefert die Nummer der Speicherstelle zurück.

D.2.13 clearStruct

Löschen aller gespeicherten Strukturen.

Aufruf

```
comSMT('clearStruct');
```

Anhang E

Inhalt der CD-ROM

File System: Joliet

Mode: Single-Session (CD-ROM)

E.1 Diplomarbeit und Dokumente

Pfad: /docs/

/coldfire/	Dokumentation MCF5272 Coldfire
6025UserManual.pdf	Sundance SMT6025 Manual
da.dvi	Diplomarbeit (DVI-Datei)
da.pdf	Diplomarbeit (PDF-Datei)
da.ps	Diplomarbeit (PostScript-Datei)
ds031.pdf	Xilinx Virtex2 Datenblatt
GenC_readme3.doc	Dokumentation *GenC*
paper_dsp04.pdf	Design of a flexible and scalable 4x4 MIMO testbed
SMT310QUserManual.pdf	Sundance SMT310Q Manual
smt365usermanual.pdf	Sundance SMT365 Manual
tms320c6416.pdf	TI TMS320C6416 Datenblatt

E.2 *LaTeX*-Dateien

Pfad: /docs/da_src/

0_abstract.tex	Abstract
0_kurzfassung.tex	Kurzfassung
0_vorwort.tex	Vorwort, Danksagung
1_einleitung.tex	Kapitel 1

2_hwbeschreibung.tex .	Kapitel 2
3_swbeschreibung.tex .	Kapitel 3
4_GencAddOn.tex . . .	Kapitel 4
5_hwinterface.tex	Kapitel 5
6_bspTest.tex	Kapitel 6
7_zusammenfassung.tex	Kapitel 7
anhang_a.tex	Anhang A (Verzeichnisstruktur für die Simulation)
anhang_b.tex	Anhang B (Matlab-Scripte)
anhang_c.tex	Anhang C (Generierte Dateien)
anhang_d.tex	Anhang D (Befehle für das Matlab-Hardwareinterface)
anhang_e.tex	Anhang E (Inhalt CD-ROM)
da.tex	Hauptdokument
hagenberg.sty	Hagenberg Style Vorlage für *LaTeX*
literatur.bib	Literatur Datenbank für BibTeX
messbox.tex	Messbox zur Druckkontrolle
mod_siam.bst	BibTeX Style (SIAM bibliography style)
subfigure.sty	Benötigtes StyleFile für das Einbinden von Subfigures

E.3 Abbildungen

Pfad: /docs/da_src/images/

ablaufGenCaddOn(.eps,.pdf,.vsd)	Umwandlungsablauf von GenCaddOn
blockRTS(.eps,.pdf,.vsd)	Blockschaltbild ARCS-RTS-DSP-Board
datenraten(.eps,.pdf) .	Datenraten.
datentransferRTS(.eps,.pdf,.vsd)	Schematische Dartstellung des Datentransfers zwischen PC und ARCS-RTS-DSP-Board
dirARC(.eps,.pdf,.tif) .	Verzeichnisstruktur ARCS-RTS-DSP-Board
dirSundance(.eps,.pdf,.tif)	Verzeichnisstruktur Sundance-Modul
hwinterface(.eps,.pdf,.vsd)	Konzeptioneller Aufbau des Matlab Hardware Interfaces
mexFunct(.eps,.pdf,.vsd)	Erstellen einer Matlab mex-Datei unter Microsoft Windows
SDK6025(.eps,.pdf,.vsd)	SMT6025 Schnittstelle zur Sundance-Hardware

sFunct(.eps,.pdf,.vsd) . Einfacher Simulationsablauf einer Simulation unter Simulink mit einer S-Funktion
simulinkBlock(.eps,.pdf,.tif) Simulink-Modell
smt365(.eps,.pdf) . . . Blockschaltbild Sundance-SMT365
speicherRTS(.eps,.pdf,.vsd) Speicherbelegung des externen Speichers bei der ARCS-Hardware

E.4 Sonstiges

Pfad: /docs/da_src/

src/ . . . . . Code Beispiele in der Diplomarbeit
src/RTS/ . . . . . Code Beispiele in der Diplomarbeit für das ARCS-RTS-DSP-Board
src/SMT/ . . . . . Code Beispiele in der Diplomarbeit für das Sundance-SMT-Modul
fonts-bakoma/ . . . . . BaKoMa True Type Fonts
german-bibtex-styles/ . Deutsche BibTeX Dateien

E.5 Kurzfassung

Pfad: /docs/kurzfassung/

0_kurzfassung.tex . . . . Kurzfassung *LaTeX*-Source
0_kurzfassung.pdf . . . Kurzfassung (PDF-Datei)

E.6 Abstract

Pfad: /docs/abstract/

0_abstract.pdf . . . . . Abstract (PDF-Datei)
0_abstract.tex . . . . . Abstract *LaTeX*-Source
ExtendedAbstract.doc . Extended Abstract (DOC-Datei)
ExtendedAbstract.pdf . Extended Abstract (PDF-Datei)
ExtendedAbstract.tex . Extended Abstract *LaTeX*-Source

E.7 Beispiele

Pfad: /examples/

ARCS-RTS-Board/ . . Beispiel für das ARCS-RTS-DSP-Board
Sundance-SMT/ . . . . Beispiel für das Sundance-SMT-Modul

E.8 Programme

Pfad: /programme/

GenC.exe	Programm zum Kompilieren von *Generic C* Code
GenCaddOn.exe	Erstellen der hardwarespezifischen Dateien
LANFlasher.exe	Programm zum Programmieren des RTS-Board

E.9 Sourcecodes

Pfad: /src/

comRTS/	Interface Dateien zwischen Matlab und ARCS-RTS-DSP-Board
comSMT/	Interface Dateien zwischen Matlab und Sundance-SMT365-Modul
GenCaddOn/	Sourcecode von *GenCaddOn*.

Literaturverzeichnis

[1] *MCF5272: Integrated ColdFire Version 2 Microprocessor*. URL, http://e-www.motorola.com, Dezember 2003. Kopie auf CD-ROM.

[2] E. Aschbacher, S. Caban, W. Keim, R. Langwieser, G. Maier, C. Mehlführer, M. Rupp, and A. L. Scholtz, *Design and subsystem verification of a flexible and scalable 4 x 4 MIMO testbed*, tech. report, Institute of Communications and Radio-Frequency Engineering University of Technology Vienna, Sep. 2004. to be presented at the *IEEE Radio and Wireless Conference (RAWCON'04), Workshop on "MIMO Implementation Aspects"*.

[3] E. Aschbacher, S. Caban, G. Maier, C. Mehlführer, and M. Rupp, *Design of a flexible and scalable 4x4 MIMO testbed*, tech. report, Institute of Communications and Radio-Frequency Engineering University of Technology Vienna, 2004. to be published in Proc. of the *11th IEEE Signal Processing Workshop (DSP 2004), Kopie auf CD-ROM.*

[4] R. Kloibhofer and H. Eberl, *Kanalsimulator SmartSim*. Inoffizielle Dokumentation, Dezember 2003.

[5] R. Lieger, *Allgemeine Informationen zum Programm LANFlasher*. Persönliches Gespräch und Email-Verkehr, Mai 2004.

[6] M. Meyenburg, *Kommunikationsgrundlagen ARCS RTS-DSP-Board*. Persönliches Gespräch und Email-Verkehr, Jänner 2004.

[7] M. Rupp, *GenC-Conversion from Generic C to . . .* . Institute of Communications and Radio-Frequency Engineering University of Technology Vienna, November 2001. Kopie auf CD-ROM.

[8] Sundance Multiprocessor Technology Ltd., *SMT 365 User Manual Version 3.0*, Februar 2001. Kopie auf CD-ROM.

[9] Sundance Multiprocessor Technology Ltd., *SMT 310Q User Manual Version 3.1*, März 2003. Kopie auf CD-ROM.

[10] Texas Instruments, *TMS320C6414, TMS320C6415, TMS320C6416 fixed-point digital signal processors*, Februar 2001. Kopie auf CD-ROM.

[11] Xilinx, *Virtex-II Platform FPGAs: Complete Data Sheet*, März 2004. Kopie auf CD-ROM.