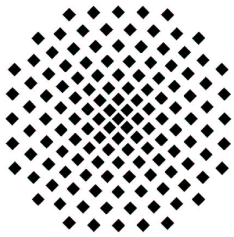

Universität Stuttgart

Fakultät für Konstruktions- und Fertigungstechnik

ISW Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungs- einrichtungen

FISW Steuerungstechnik GmbH

Studienarbeit

Realisierung einer Robotersteuerung mit CAN-Bus Kommunikation unter RT-Linux

Rainer Sigle

Themensteller: Prof. Dr.-Ing. G. Pritschow

Betreuer: Dipl.-Ing. Ulrich Laible,
Dipl.-Ing. Till Franitza

Abgabetermin: 01.04.2002

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Behinderte Menschen die im Rollstuhl sitzen haben eine eingeschränkte Bewegungsmöglichkeit. Aus diesem Grund wurde am FISW Stuttgart im Projekt GRIP (Generic Robot Interacting with People) eine Greifhilfe für körperlich Behinderte entwickelt. Diese Greifhilfe besteht aus einem Roboterarm der von der Firma Wittenstein konzeptioniert wurde. Der Arm kann in die Gruppe der Vertikal-Knickarm-Roboter eingeordnet werden. Er besitzt sechs serielle Achsen, mit einer Gesamtlänge von 1,5 m. Am Handfußpunkt des Arms ist ein Greifer montiert, der dazu dient Gegenstände zu handhaben oder sonstige Arbeiten im Haushalt zu erledigen. Aufgrund seines geringen Gewichts kann der Arm an einen Rollstuhl montiert werden.
Die Antriebe der einzelnen Armachsen besitzen eine hohe Leistungsdichte, geringen Energieverbrauch, sowie eine intelligente Steuerung über CAN-Bus. Die Steuerung des Arms erfolgt über einen Joystick, mit dem es auf einfache Art möglich ist alle Achsen zu bewegen. Die dafür notwendige Bahnsteuerung läuft auf einem kleinen, kompakten PC in PC/104 Bauweise, der ebenfalls am Rollstuhl montiert wird. Somit ist es möglich das komplette System (bestehend aus Arm, Joystick und Steuerungsrechner) autonom, zu betreiben.
Die Bahnsteuerung basiert auf einem relativ umfangreichen NC-Kern der Firma ISG, der auch in Werkzeugmaschinen eingesetzt wird, und dient zur Generierung von Lagesollwerten für die einzelnen Achsen aus den Joystickdaten. Die Steuerungssoftware läuft unter dem Betriebssystem Windows CE 3.0. Da diese Bahnsteuerung inklusive der Transformation sehr rechenintensiv, und an die Grenze der Leistungsfähigkeit des PC/104 Systems ging, suchte man nach Alternativen.
Ziel dieser Studienarbeit war es, ein vergleichbares Steuerungssystem für den Greifarm unter RT-Linux aufzubauen.

1.2 Ziele dieser Arbeit

Es galt also ein System aufzubauen das erst mal grundsätzlich auf dem PC/104-System unter RT-Linux bootet und läuft, und es sollte möglich sein mit einem Entwicklungssystem zu kommunizieren, auf dem Software geschrieben und kompiliert werden kann. Dazu ist über Ethernet eine Netzwerkverbindung zu installieren, mit der es möglich ist Programme auf das PC/104-System zu

laden. Anschließend muß für die CAN-Karte ein Gerätetreiber als Echtzeitmodul entwickelt werden, der in der Lage ist die CAN-Kommunikation zwischen Arm und Steuerung zu übernehmen. Um die Funktion dieses Echtzeitmoduls unter Beweis zu stellen, sollte schlußendlich ein einfaches Steuerungssystem entwickelt werden, mit dem es möglich ist die Achsen des Roboters zu steuern. Als Grundlage für weitere Entwicklungen (Hinzufügen der Transformation) sollte am Ende der Studienarbeit ein Steuerungssystem bestehen, das in der Lage ist harte Echtzeitanforderungen bzgl. CAN-Bus Kommunikation unter RT-Linux zu erfüllen.

1.3 Aufbau der Ausarbeitung

Die vorliegende Ausarbeitung beschreibt die einzelnen Komponenten dieses Steuerungssystems. Im zweiten Kapitel werden die Grundlagen der CAN-Bus-Kommunikation sowie der Aufbau des verwendeten CAN-Controllers SJA1000 beschrieben. Anschließend wird auf den Aufbau des Steuerungssystems näher eingegangen, das heißt die Hardware und ihre einzelnen Komponenten werden vorgestellt. Im vierten Kapitel erkläre ich dann die Vorgehensweise wie man ein funktionsfähiges RT-Linux auf dem PC/104-System installiert. Im fünften Kapitel wird das Echtzeitmodul das die Steuerung realisiert vorgestellt.

Kapitel 2

Grundlagen

2.1 CAN-Bus

2.1.1 Grundlegende Merkmale

Feldbussysteme dienen als Kommunikationsmedium für verteilte Aktoren, Sensoren und Steuerungen in der Automatisierungstechnik und überall dort, wo intelligente elektronische Bausteine mit geringem Verkabelungsaufwand verbunden werden müssen. An einen Feldbus und die angeschlossenen Elemente werden hohe Anforderungen gestellt: sie müssen kostengünstig, stör- und ausfallsicher sowie technisch sehr leistungsfähig sein.Im Automobilbereich treffen die im letzten Abschnitt beschriebenen Anforderungen auf die günstige Tatsache der weiten Verbreitung und des Masseneinsatzes, die wiederum Voraussetzung sind für geringe Preise und Standardisierung des Feldbussystems und dessen Anschlußelemente. Der Controller Area Network- (CAN)-Bus wurde 1981 von Bosch und Intel entwickelt und findet international weite Verbreitung im Automobil- (Mercedes, BMW) und Haushaltsgerätesektor (Bosch). Aber auch im Bereich der Industrie-Automation steigt der Anteil der CAN-basierten Anwendungen. Durch die hohen Stückzahlen sind preiswerte Anschaltkomponenten erhältlich.

2.1.1.1 Anwendungsgebiete

Als Feldbus wird CAN für die Vernetzung komplexer Controller und Steuergeräte verwendet. Wichtigster Anwendungsbereich ist hier die Automobilindustrie, die CAN einsetzt, um die wegen des steigenden Elektronikanteils im KFZ, immer aufwendigere Verkabelung im Auto in den Griff zu bekommen. CAN kann auch als Aktor/Sensor-Bus im Bereich der Antriebstechnik eingesetzt werden, und überall dort wo hohe Datenraten und eine sichere Übertragung verlangt werden.

2.1.1.2 Eigenschaften

Die wesentlichen Merkmale von CAN sind eine hohe Datenübertragungsgeschwindigkeit von 10 kBit/s bis 1 Mbit/s bei einer Buslänge von 40 m bis 1 km. In den geringen Datenblocklängen von maximal 8 Byte können nur kurze Nachrichten übermittelt werden, die jedoch geringe Datenübertragungszeiten benötigen. Außerdem ist die kurze Datenlänge für Anwendungen im KFZ-Bereich bei den meisten Anwendungen ausreichend. Die Reaktionszeiten werden darüber hinaus durch die

mögliche Prioritätsvergabe über eine spezielle Message-ID, den sogenannten CAN-Identifeir beeinflußt. Hier ist auch besondere Vorsicht geboten: Hochpriore Nachrichten können bei genügender Übertragungshäufigkeit niederprioren Datenobjekten den Buszugang über längere Zeit oder gar für immer versperren. CAN fungiert als Multi-Master-Bus. Jeder sendewillige Teilnehmer kann Master werden und versucht, sein Nachrichtenobjekt auf den Bus zu legen. Das bitweise und prioritätsgesteuerte Buszugriffsverfahren (Busarbitrierung) unterscheidet zwischen ”dominanten” und ”rezessiven” Bits oder Spannungspegeln, so daß sich schließlich die Nachricht mit der höchsten Priorität am Bus durchsetzt. Insgesamt können maximal 2^{11} = 2048 Identifier (11Bit = Lände des Identifiers) für verschiedene Nachrichtenobjekte vergeben werden. Bei Verwendung des sogenannten *extended* Formats sogar bis zu 2^{29} (29 Bit ID). CAN ist also nicht teilnehmer-, sondern nachrichten- bzw. objektorientiert. Jeder Knoten empfängt alle Nachrichten und entscheidet ob er die Nachricht annimmt oder nicht. Da CAN als Feldbus im Automobil starken Störeinflüssen (starke elektromagnetische Einstrahlungen, physikalische Belastungen) unterliegt, wurde auf die Übertragungssicherheit und Datenkonsistenz besonderer Wert gelegt. Verschiedene Maßnahmen führen zu einer Hammingdistanz von 6 und einer Restfehlerwahrscheinlichkeit von ca. 10^{-13}.

2.1.2 CAN auf der Bitübertragungsschicht

2.1.2.1 Busmedium und Bustopologie

OSI-Schicht 1: Grundsätzlich ist CAN für die serielle Datenübertragung in einer Bus-Topologie ausgelegt. Als physikalisches Medium wird die Zweidrahtleitung (RS 485) verwendet, im Störfall kann man auch über eine Eindrahtleitung kommunizieren. Hierfür müssen jedoch entsprechende Schaltvorrichtungen vorgesehenen sein und im Fehlerfall weitere Maßnahmen durchgeführt werden. Die Zweidrahtleitung besitzt den weiteren Vorteil, die Spannungspegel differentiell übertragen zu können. Auf diese Weise werden Störungen, die aus zu großem Massepotentialversatz oder von elektromagnetischen Strahlungen herrühren, vermieden oder zumindest verringert. Die Netzausdehnung ist durch sie Signallaufzeit auf etwa 40 m beit 1 MBit/s oder 1 km bei bei 80kBit/s eingeschränkt. Es gibt keine im Protokoll festgelegte Höchstgrenze für die Anzahl der Busteilnehmer. In der Praxis wird dies allerdings durch die Leistungsfähigkeit der Bustreiber beschränkt.

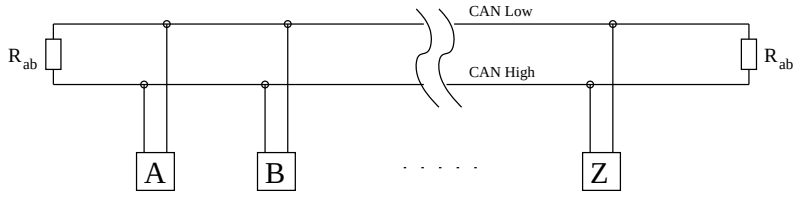

Abbildung 2.1: CAN Bustopologie

2.1.3 CAN auf der Sicherungsschicht

2.1.3.1 Prinzip der Busarbitrierung

OSI-Schicht 2a (MAC):
Als Zugriffsverfahren verwendet der CAN-Bus CSMA/CA (Carrier Sense Multiple Access with Collision Avoidence). Das klassische CSMA-Verfahren funktioniert wie beim Ethernet: Jeder sendewillige Teilnehmer prüft, ob gerade eine andere Station sendet. Entdeckt er den Carrier nicht (11 Bitzeiten Ruhepotenzial), greift er selbst auf den Bus zu. Bei fast gleichzeitigem Sendewunsch mehrerer Stationen erkennen die Teilnehmer aufgrund der Signalausbreitungszeit nur den freien Bus und senden. Die entstehenden Kollisionen werden entweder durch verfälschte Prüfbits (CRC-Fehler) erkannt (z.B. beim Ethernet mit CSMA / Collision Detect) oder, wie bei CAN, von vornherein durch entsprechend kurze Leitungen und bitweiser Busarbitrierung vermieden (Collision Avoidence) (so zumindest die Theorie). Hierbei beginnt eine Station mit der Aussendung eines Nachrichtenobjekts, indem zunächst die (eigene) Quelladresse bitweise auf den Bus gelegt wird. Die sogenannte ”dominante” Quelladresse mit der kleineren Identifizierungszahl (ID) setzt sich durch, indem jede Sendestation mit der Aussendung des höchstwertigen Adressbits beginnt und zwischen ”dominanten” (Bit = 0) und ”rezessiven” (Bit = 1) Spannungspegeln unterschieden wird. Der dominante Zustand setzt sich am Bus durch, die anderen Teilnehmer brechen ihren Sendeversuch ab. Auf diese Weise werden Kollisionen vermieden. Dies setzt natürlich voraus, daß gleichzeitig (bitsynchron) an jeder Station der gleiche Spannungspegel anliegt.

2.1.3.2 CAN-Botschaften

OSI-Schicht 2b (LLC):
CAN überträgt alle Botschaften in einem definierten Datenrahmen. Es gibt mehrere dieser Datenrahmen. Folgende Formate sind im CAN-Bus definiert:

- Datentelegramm (Data Frame): für die Datenübertragung
- Datenanforderungstelegramm (Remote Frame): dient zur Anforderung von Daten
- Überlasttelegramm (Overload Frame): Flußregelung
- Fehlertelegramm (Error Frame): Fehlererkennung und -benachrichtigung

Das bitorientierte Rahmenformat einer CAN-Nachricht ähnelt dem Format des HDLC-Protokolls (High-Level Data Link Control: Datenstruktur der zweiten Schicht von packetvermittelten Datennetzen).
Das Datentelegramm besitzt folgenden Aufbau:

Startfeld	1 bit
Arbitrierungsfeld	11 + 1 bit
Steuerfeld	6 bit
Datenfeld	0 .. 64 bit
CRC-Feld	15 + 1 bit
ACK-Feld	1 + 1 bit

Endefeld	7 bit

Tabelle 2.1: Aufbau eines CAN Datentelegramms

Startfeld: 1 dominantes Bit, mit welchem der Sendeversuch gestartet wird.
Arbitrierungsfeld: 11 Bit Identifier + 1 Bit RTR (Remote Transmit Request). In diesem Feld sendet die Station die Nachrichten-ID.

- RTR = 0 Data Frame
- RTR = 1 Remote Frame

Das RTR-Bit dient für die Unterscheidung eines Datentelegramms und eines Datenanforderungstelgeramms. Ist dieses Bit gesetzt, dann wird eine Nachricht angefordert.
Steuerfeld: Höherwertige 2 Bits für Extended CAN reserviert, 4 Bit Data Length Code (Länge der Botschaft: 0 - 8 Bytes)
Datenfeld: Datensegment mit 0 - 8 Bytes Daten
CRC-Feld: 15 Bit Generator-Polynom + 1 Bit rezessives CRC Begrenzungsbit
Bestätigungsfeld: 1 Bit ACK-Slot + 1 Bit ACK-Begrenzung (rezessiv): sendender Knoten überträgt rezessives Bit im Slot; Knoten, die fehlerfrei empfangen haben, senden dominantes Bit im Slot.
Endefeld: EOF (End of Frame): 7 rezessive Bits
Es ergeben sich somit je Rahmen 44 Bit Overhead + 0 ... 64 Bit Daten.

2.1.3.3 Fehlererkennung und Fehlerbehandlung

CAN verfügt über eine Reihe von Kontrolleinrichtungen zur Störungserkennung und -korrektur:

- Cyclic Redundancy Check (CRC)
  Der CRC-Check erkennt mit einem Rahmensicherungswort im Datenrahmen Übertragungsfehler. Nach Festellung eines Übertragungsfehlers fordert der Empfänger erneut eine Übertragung an. Der leistungsfähige CRC des CAN ist speziell an die kurzen Botschaften angepasst.
- Während des Sendens überprüft jede Station den Buspegel. Findet sie auf der Busleitung - außer im Arbitrierungsfeld - einen anderen Bitwert als den gesendeten, liegt ein Fehler vor.
- Ein Datenrahmen darf bis zum Ende des CRC maximal fünf aufeinander folgende Bits gleicher Polarität haben. Daher fügt der Sender, wenn er fünf gleiche Bits übertragen hat, automatisch ein zusätzliches Bit entgegengesetzter Polarität ein (Bit-Stuffing). Die Empfänger eliminieren dieses zusätzliche Bit wieder (Destuffing). Dadurch wird Codetransparenz (Bitfolgeunabhängigkeit) erzielt. Das Verfahren verhindert, daß eine zufällige auftretende Bitkombination mit einer anderen Nachricht verwechselt wird; z. Bsp mit einem Error-Frame, welcher aus sechs Bits gleicher Polarität besteht.
- Bei einer Rahmensicherung machen die CAN-Bausteine alle Botschaften, die von den Formatvorgaben des CAN-Protokolls abweichen, automatisch ungültig.

Stellt ein CAN-Controller eine Störung fest, bricht er die laufende Übertragung durch das Senden eines ”Error Frames” an alle anderen Stationen ab. Dieses besteht aus sechs Bits gleicher Polarität und setzt sich - im Falle sechs dominanter Bits - gegen jede andere Botschaft auf dem Bus durch. So ist sicher, daß alle Stationen den Fehler erkennen.

2.2 Der CAN-Controller Phillips SJA1000

Das verwendete CAN-Interface PC-I 04/104 der Firma IXXAT, verwendet den passiven CAN-Controller SJA1000 von Phillips. Da zur Implementierung des Gerätetreibers Kenntnisse über die Funktion des Controllers und den Aufbau seiner Register nötig waren, soll dieser hier beschrieben werden.

2.2.1 Eigenschaften

Der SJA1000 besitzt folgende Eigenschaften:

- zwei Modi: BasicCAN- und PeliCAN-Mode
- pinkompatibel zum Vorgänger-Modell PCA82C200, im BasicCAN-Mode sogar vollständig softwarekompatibel
- 64 kByte großer Empfangspuffer
- CAN 2.0B kompatibel
- unterstützt normal- als auch extended-Format Identifier
- Bitrate bis zu 1 MByte/s
- interne Taktfrequenz: 24 MHz
- zusätzliche Eigenschaften im PeliCAN-Mode:
  - Fehlerzähler kann gelesen/geschrieben werden
  - Error-Warning-Limit kann programmiert werden
  - Bestimmung des letzten aufgetretenen Fehlers
  - Arbitration-Lost-Capture-Register (zeigt Position im ID, ab welcher Arbitrierung verlohren ging)
  - Single-Shot Transmission (im Falle eines Arbitrierungsverlusts, erfolgt keine erneute Übertragung der Nachricht)
  - Listen Only Mode (Nachrichten werden entgegen genommen, es erfolgt kein ACK, Übertragungsfehler wird über passives Error-Flag signalisiert)
  - Self Reception Request (CAN-Loopback, auf diese Weise kann ein Selbsttest durchgeführt werden)
  - Leistungsfähigere Akzeptanzfilterung (4-byte Acceptance-Code, 4-byte Acceptance-Mask)
  - Automatische Bitraten-Erkennung

2.2.2 Blockschaltbild

Die Abbildung 2.2 zeigt das Blockschaltbild des Controllers.

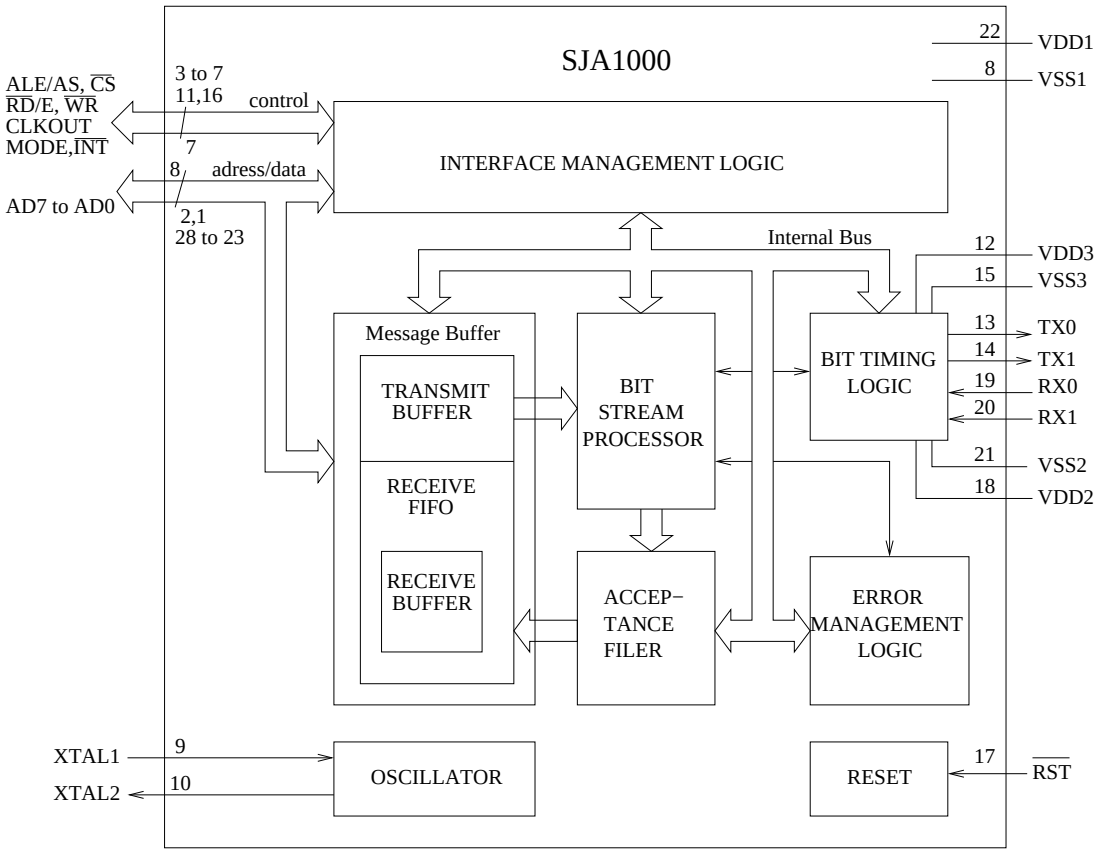

Abbildung 2.2: Blockschaltbild SJA1000

Die einzelnen Funktionsblöcke haben dabei folgende Aufgaben:

Interface Managment Logic (IML)

- interpretiert die Befehle von der CPU
- kontrolliert die Adressierung der CAN-Register
- stellt Interrupts und Statusinformationen zur Verfügung

Transmit Buffer (TXB)

- Schnittstelle zwischen CPU und Bit Stream Processor
- Kann eine komplette Nachricht speichern
- 13 Byte groß

- beschrieben von der CPU, ausgelesen vom Bit Stream Processor

Receive Buffer (RXB,RXFIFO)

- Schnittstelle zwischen dem Akkzeptanzfilter und der CPU
- Speichert die empfangenen und akzeptierten Nachrichten vom CAN-Bus
- Receive Buffer (RXB): 13 Byte großes Fenster des RXFIFO, auf das die CPU zugreift
- RXFIFO: 64 Bytes groß. Wenn im RXFIFO kein Platz mehr für neue Nachricht, dann wird diese verworfen und ein Flag im Statusregister gesetzt (und wenn erlaubt, wird ein Interrupt ausgelöst) $\Rightarrow$ CPU kann eine Nachricht abarbeiten und während dessen weitere empfangen

Acceptance Filter (ACF)

- Vergleicht die empfangene ID mit dem Inhalt des Akzeptanzfilter-Register
- Entscheidet, ob eine Nachricht akzeptiert wird oder nicht
- Wenn eine Nachricht akzeptiert wurde, wird die gesamt Nachricht im RXFIFO gespeichert

Bit Stream Processor (BSP)

- Kontrolliert den Datenstrom zwischen dem RXFIFO und dem CAN-BUS
- Fehlererkennung, Arbitrierung, Stuffing und Fehlerbehandlung

Bit Timing Logic (BTL)

- Beobachtet den CAN-Bus und kümmert sich um das Bit-Timing
- Synchronisiert den Bit-Strom auf dem CAN-Bus

Error Managment Logic (EML)

- Verantwortlich für Fehlerbegrenzung
- Empfängt Fehlerbenachrichtigungen vom Bit Stream Processor
- Informiert den Bit Stream Processor und die Interface Management Logik über Fehlerstatistiken

2.2.3 Registerlayout im BasicCAN-Mode

Auf der CAN-Karte wird die Basis-Adresse das CAN-Controllers über eine Jumperleiste eingestellt. Diese Adresse dient als Interface um die in den Registern des Controllers liegenden Daten zu lesen bzw. in die Register schreiben zu können. Der relvante Bereich umfasst einen 32 Byte breiten Speicherbereich, welcher mittels I/O mapped-memory in den Adressraum des Hauptspeichers eingeblendet wird. So kann auf ihn mittels einfachen und sehr schnellen Schreib- und Leseoperationen zugegriffen werden kann.
Im folgenden wird das Registerlayout des Controllers im sog. BasicCAN-Mode beschrieben. Der erweitere PeliCAN-Mode besitzt einen anderen Aufbau der Register, soll aber hier, da dieser nicht verwendet wurde, übergangen werden (für weitere Informationen: siehe Datenblatt zum SJA1000).
Im BasicCAN-Mode sind die einzelnen Bereiche der Register folgendermassen aufgeteilt:

Address	**Segment**	**Operating Mode**		**Reset Mode**	
		Read	Write	Read	Write
0	Control	control	control	control	control
1		0xFF	command	0xFF	command
2		status	-	status	-
3		interrupt	-	interrupt	-
4		0xFF	-	acceptance code	acceptance code
5		0xFF	-	acceptance mask	acceptance mask
6		0xFF	-	bus timing 0	bus timing 0
7		0xFF	-	bus timing 1	bus timing 1
8		0xFF	-	output control	output control
9		res.	res.	res.	res.
10	Transmit	id (10-3)	id (10-3)	0xFF	-
11		id (0-2), RTR and DLC	id (0-2), RTR and DLC	0xFF	-
12		data byte 1	data byte 1	0xFF	-
13		data byte 2	data byte 2	0xFF	-
14		data byte 3	data byte 3	0xFF	-
15		data byte 4	data byte 4	0xFF	-
16		data byte 5	data byte 5	0xFF	-
17		data byte 6	data byte 6	0xFF	-
18		data byte 7	data byte 7	0xFF	-
19		data byte 8	data byte 8	0xFF	-
20	Receive	id (10-3)	id (10-3)	id (10-3)	id (10-3)
21		id (0-2), RTR and DLC	id (0-2), RTR and DLC	id (0-2), RTR and DLC	id (0-2), RTR and DLC
22		data byte 1	data byte 1	data byte 1	data byte 1
23		data byte 2	data byte 2	data byte 2	data byte 2
24		data byte 3	data byte 3	data byte 3	data byte 3
25		data byte 4	data byte 4	data byte 4	data byte 4
26		data byte 5	data byte 5	data byte 5	data byte 5
27		data byte 6	data byte 6	data byte 6	data byte 6
28		data byte 7	data byte 7	data byte 7	data byte 7
29		data byte 8	data byte 8	data byte 8	data byte 8
30		0xFF	-	0xFF	-
31	Clock	clock divider	clock divider	clock divider	clock divider

Tabelle 2.2: Belegung der Bytes im CAN-Controller

Anhand der Tabelle ist zu erkennen, daß zwei grundsätzliche Betriebsarten existieren:

- Operating Mode: In diesem Mode kann der CAN-Controller Nachrichten senden und empfangen. Es ist der eigentliche Betriebsmodus des CAN-Controllers. Man hat keinen Zugriff auf Parameter im Control-Segment.

- Reset Mode: Der Reset-Mode dient zur Parametrierung des Controllers. Nach einem Hardware-Reset und bei der Initialisierung des Controllers befindet man sich im Reset-Mode. Die Parametrierung erfolgt durch Schreiben von Daten in das Command- und Clock-Divider-Segement auf welche nun lesend und schreibend zugegriffen werden kann. Sind alle erforderlichen Parameter (Bus Timing, Acceptance Code, Acceptance Mask, Output Control sowie die Interrupt-Enable Bits) geschrieben erfolgt schlußendlich das Starten des Controllers durch ein Wechseln in den Operating-Mode.

Der Aufbau des Controllers läßt sich in vier Segmente gliedern:

1. Control-Segment
2. Transmit Buffer
3. Receive Buffer
4. Clock Divider

Diese einzelnen Segmente weden nun noch näher beschrieben:

2.2.3.1 Control-Segment

Das Control-Segment liegt zwischen Byte 0 und Byte 9 des CAN-Controllers. Dieser Bereich ist für die Konfiguration sowie Kontrolle des Controllers verantwortlich. Im Folgenden werden die Register des Control-Segments detailliert beschrieben:

1. Control-Register

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
Res.	Res.	Res.	Overrun Interrupt Enable	Error Interrupt Enable	Transmit Interrupt Enable	Receive Interrupt Enable	Reset Request

Tabelle 2.3: Layout des Control-Registers

Das Control-Register wird verwendet um das Verhalten des Controllers zu verändern. Es beinhaltet die Interrupt-Enable-Bits. Mit ihnen wird angegeben oder bei Eintreffen eines bestimmten Ereignisses ein Interrupt ausgelöst werden soll oder nicht.

- Overrun Interrupt Enable: Sobald das Data Overrun Bit gesetzt wird, wird ein Interrupt ausgelöst
- Error Interrupt Enable: Ist das Fehlerlimit erreicht, oder ändert sich der Zustand des Busses, so wird ein Interrupt ausgelöst
- Transmit Interrupt Enable: Interrupt wird ausgelöst bei erfolgreicher Übertragung einer Nachricht

- Receive Interrupt Enable: Wird eine Nachricht ohne Fehler empfangen, erfolgt eine Interrupt Auslösung

Bit 0 (Reset Request) dient dem Umschalten zwischen Operating-Mode und Reset-Mode. Bei Setzen dieses Bits wird ein Reset-Request ausgelöst. Die aktuelle Übertragung wird abgebrochen und in den Reset-Mode gewechselt.

2. Command-Register

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
Res.	Res.	Res.	Go To Sleep	Clear Data Overrun	Release Receive Buffer	Abort Transm.	Transm. Request

Tabelle 2.4: Layout des Command-Registers

Das Setzen eines Bits im Command Register startet eine Aktion auf der Transfer-Ebene des SJA1000. In das Register kann nur geschrieben werden; ein Lesen liefert immer 0xFF.

- Go To Sleep: Wechseln in den Sleep Mode. Bus Aktivitäten werden ignoriert, es werden keine Interrupts ausgelöst
- Clear Data Overrun: Ist ein Data Overrun aufgetreten, so dient dieses Bit zum Rücksetzen des Fehlerzustandes.
- Release Receive Buffer: Nachdem vom RXFIFO eine Nachricht gelesen wurde, muß mit Release Receive Buffer, der zugehörige Speicherbereich freigegeben werden. Dies hat zur Folge, daß die als nächstes anstehende Nachricht eine Positoin weiter nach vorne rückt, und ausgelesen werden kann.
- Abort Transmission: Bricht eine gerade in Arbeit befindliche Übertragung ab, um z.Bsp. einer höher prioren Nachricht Vorrang zu geben
- Transmission Request: Nachricht im Transmit Buffer soll versendet werden

3. Status-Register

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
Bus S.	Error S.	Transmit S.	Receive S.	Transm. Compl. S.	Transmit Buffer S.	Data Overrun S.	Receive Buffer S.

Tabelle 2.5: Layout des Status-Registers

Im Status-Register lässt sich der aktuelle Zustand des Controllers auslesen. Es ist ein Read-Only Register.

- Bus Status: 1 = Bus off; 0 = Bus on
- Error Status: 1 = error, mindestens einer der Fehlerzähler hat das Limit erreicht; 0 = ok
- Transmit Status: 1 = Nachricht wird übertragen; 0 = Frei
- Receive Status: 1 = Nachricht ist angekommen; 0 = nichts da
- Transmission Complete Status: 1 = die letzte Übertragung war erfolgreich, 0 = Übertragung ist noch nicht beendet
- Transmit Buffer Status: 1 = in den Transmit Buffer kann geschrieben werden; 0 = Transmit Buffer ist belegt
- Data Overrun Status: 1 = Overrun, eine Nachricht ging verloren; 0 = ok
- Receive Buffer Status: 1 = Voll, eine oder mehrere Nachrichten stehen im RXFIFO; 0 = leer, keine Nachrichten vorhanden

4. Interrupt-Register

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
Res.	Res.	Res.	Wake-Up I.	Data Overrun I.	Error I.	Transmit I.	Receive I.

Tabelle 2.6: Layout des Interrupt-Registers

Im Interrupt-Register wird je nach eingestellten Enable-Bits im Control-Register und eintreffendem Event das entsprechende Bit gesetzt.

- Wake-Up Interrupt: Befindet sich der Controller im Sleep Mode, und es kommt eine Nachricht an, dann wird Wake-Up-Interrupt gesetzt
- Data-Overrun Interrupt: Data Overrun ist aufgetreten
- Error Interrupt: Fehler, mindestens einer der Fehlerzähler hat das Fehlerlimit erreicht
- Transmit Interrupt: Wird ausgelöst, sobal der Zustand des Transmit-Buffer-Status-Bits von logisch 0 nach logisch 1 wechselt (Transmit Buffer ist wieder frei)
- Receive Interrupt: Gesetzt solange der Receive FIFO nicht leer ist

5. Acceptance-Code

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
AC.7	AC.6	AC.5	AC.4	AC.3	AC.2	AC.1	AC.0

Tabelle 2.7: Layout des ACC-Registers

6. Acceptance-Mask

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0

Tabelle 2.8: Layout des ACM-Registers

Das Akkzeptanz-Code-Register und das Akkzeptanz-Mask-Register bilden zusammen das Akkzeptanzfilter. Dieses ist verantwortlich für die Nachrichtenfilterung.
Eine Nachricht wird angenommen wenn folgende Gleichung erfüllt ist.

$$(\text{ID.10 to ID.3}) \equiv (\text{AC.7 to AC.0}) \vee (\text{AM.7 to AM.0}) \equiv 11111111$$

Sollen alle Nachrichten angenommen werden so hat im ACR = 00000000, und im AMR = 11111111 zu stehen.

7. Bus Timing 0

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
SJW.1	SJW.0	BRP.5	BRP.4.	BRP.3	BRP.2	BRP.1	BRP.0

Tabelle 2.9: Layout des BT0-Registers

Der Inhalt des Bus Timing 0 Registers definiert die Werte des sog. Baudraten-Vorteilers (BRP), und der Synchronisationssprungweite (SJW). Beide werden zur Einstellung der Baudrate benötigt.
Befindet sich der Controller Reset Mode, so können diese Werte geschrieben werden.

8. Bus Timing 1

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

Tabelle 2.10: Layout des BT1-Registers

Im Bus Timing 1 Register stehen die Werte für die Länge der Bitperiode, die Lage des Abtastzeitpunkts, sowie die ein Bit (SAM) welches die Anzahl der Samples angibt, die pro Abtastung durchgeführt werden sollen.
Aus Bustiming Register 1 und Bustiming Register 0 wird die Baudrate errechnet. Eine ausführliche Erklärung wie diese Berechnung funktioniert steht in [16].

9. Output-Control-Register

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0

Transistor P1	Transistor N1	Polarity 1	Transistor P0	Transistor N0	Polarity 0	Mode 1	Mode 0

Tabelle 2.11: Layout des Output Control-Registers

Über die Mode-Bits M0 und M1 wird der Ausgabemodus eingestellt. Es stehen drei Modi zur Verfügung:

- Im Normal-Output-Mode wird der Bitstrom auf beiden Ausgängen gesendet. Hierfür kann die Arbeitsweise der Treibertransistoren (Pull Up, Pull Down, Push mittels P1,P0,N1,N0) sowie die Polarität (Pol1,Pol0) der Ausgänge eingestellt werden.
- Im Clocked-Output-Mode wird über den einen Ausgang der Bitstrom und über den anderen ein Clock-Signal gesendet, dessen Frequenz sich aus den Einstellungen in den Bus Timing Registern ergibt.
- Im Bi-Phase-Output-Mode werden die dominanten Bits eines Objekts abwechselnd auf einem der beiden Ausgänge ausgegeben, so daß der Bitstrom insgesamt keinen Gleichspannungsanteil besitzt. [4]

2.2.3.2 Transmit Buffer

Adr.	Feld	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
10	Deskriptor	Id byte 1	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
11		Id byte 2	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
12	Daten	TX Daten 1	Transmit Daten byte 1							
13		TX Daten 2	Transmit Daten byte 2							
14		TX Daten 3	Transmit Daten byte 3							
15		TX Daten 4	Transmit Daten byte 4							
16		TX Daten 5	Transmit Daten byte 5							
17		TX Daten 6	Transmit Daten byte 6							
18		TX Daten 7	Transmit Daten byte 7							
19		TX Daten 8	Transmit Daten byte 8							

Tabelle 2.12: Transmit Buffer Layout

Der Transmit-Buffer dient als Aufnahme einer zu sendenden Nachricht. Er ist unterteilt in einen Deskriptorfeld und ein Datenfeld. Die ersten zwei Bytes stellen das Deskriptorfeld dar. Dort sind enthalten:

- 11 Bit Identifier der CAN-Botschaft (je kleiner desto höhere Priorität)

- RTR (Remote Transmission Request)

- DLC (Data Length Code)

2.2.3.3 Receive Buffer

Der Receive Buffer ist vom Aufbau her identisch wie der Transmit Buffer. Es ist derjenige Teil des RXFIFOs auf den lesend zugegriffen werden kann. Ankommende Nachrichten werden in den RXFIFO gestellt und können über den Receive Buffer ausgelesen werden. Innerhalb des CAN-Controllers liegt er an den Adressen 20 bis 29.

2.2.3.4 Clock Divider

Bit 7	**Bit** 6	**Bit** 5	**Bit** 4	**Bit** 3	**Bit** 2	**Bit** 1	**Bit** 0
CAN mode	CBP	RXINTEN	-	clock off	CD.2	CD.1	CD.0

Tabelle 2.13: Clock-Divider Register

Mittels des Clock Divider Registers können Einstellungen getroffen werden, die hautpsächlich die an den CAN-Controller angeschlossene Hardware betreffen. So wird hier die CLKOUT-Frequenz (CD.0..CD.2) eingestellt, sowie der zugehörende CLKKOUT-Pin aktiviert oder deaktiviert (clock off). Weiterhin wird über das Bit ”CAN mode” zwischen BasicCAN- und PeliCAN-Mode umgeschaltet. Das Clock Divider Register liegt an Offset 31 des CAN-Controllers.

2.2.4 Initialisierung

Bei der Initialisierung des Controllers müssen folgende Werte innerhalb der Register gesetzt werden:

- Control
- Bus-Timing
- Acceptance-Code
- Acceptance-Mask
- Output-Control

Für unseren Controller verwendeten wir die Einstellungen:

- CTR = 0x1E (sämtliche Interrupts sind enabled)
- BTR1 = 0x01
- BTR0 = 0x1C (entspricht 250 KBit/s)
- ACR = 0x00

- AMR = 0xFF (alle Nachrichten akkzeptieren)
- OCR = 0x5E (nach Ixxat Handbuch muß dieser Wert eingetragen sein)

2.3 Real-Time Linux

Nachdem die Hardware vorgestellt wurde, geht es nun um die Softwarekomponenten mit denen die Steuerung realisiert werden sollte. Als nächstes wird die Betriebssystem-Erweiterung RT-Linux vorgestellt.

Der Standard Linux Kernel ist nicht echtzeitfähig. Echtzeitfähig wird hier verwendet im Sinne von ”harter Echtzeit”. Zeitkritische Anwendungen müssen auf jeden Fall realisiert werden können. Ein Prozess muss die Bedingung der Vorhersagbarkeit erfüllen. Wird die Zeitanforderung nicht eingehalten, so sind Ergebnisse nicht mehr verwendbar. Eine Verletzung der vorgegebenen Reaktionszeit führt sofort zum maximalen Schaden. Die Einhaltung der Zeitvorgaben ist also die Hauptaufgabe eines harten Echtzeitbetriebssystems.
Echtzeit-Anforderungen treten in vielen Bereichen auf, in denen es gilt Motoren zu steuern, oder sicherheitskritische Zeitbeschränkungen einzuhalten. Ein großer Anwendungsbereich sind die Steuerungs- und Regelungstechnik. Um z.Bsp. einen Roboter eine Bahn abfahren zu lassen ist ein kontinuierlicher Datenstrom nötig. Es muss auf Fehlersituationen in einer vorhersagbaren Zeit reagiert werden können. D. h. jegliche Anforderungen an Dead-Lines müssen erfüllt werden. Ein Allround-Betriebssystem wie Linux optimiert die durchschnittliche Leistungsfähigkeit des System, ist aber nicht für harte Echtzeit geeignet.

2.3.1 Design und Architektur

Victor Yodaiken und Michael Barabanov von FSMLab’s entwickelten 1999 eine Erweiterung des Betriebssystems Linux um harte Echtzeitanforderungen erfüllen zu können: RT-LINUX.
Die Kernidee die in RT-Linux steckt ist folgende: Der Standard Linux-Kernel wird so modifiziert, dass unterhalb des Linux-Betriebssystems ein zusätzliche Schicht eingefügt wird, die für die Echtzeitfähigkeit verantwortlich ist. Dieser neue Kernel arbeitet unabhängig vom Linux Kernel. Linux selbst steht unter der Kontrolle dieses Echtzeit-Kernels. Es ist der niedrigst priore Task der ausgeführt wird. Parallel können weitere RT-Tasks laufen, die von einem eigenen Scheduler verwaltet werden. Linux läuft nur noch im Hintergrund und bearbeitet dieApplikationen, die den weichen Echtzeitbedingungen genügen, wohingegen RT-Linux die Anteile übernimmt, die auf harte Echtzeit setzen. Erreicht werden die harten Echtzeitbedingungen über ein Umlenken von Hardware Interrupts, unter der Kontrolle des Echtzeitkerns.

Abbildung 2.3 verdeutlicht diese Archtiktur.

Der Großteil der RT-Linux Funktionalität ist in einer Sammlung ladbarer Kernelmodule vereinbart. Sie erlauben zusätzliche Funktionalitäten und Abstraktionslevel zur Laufzeit hinzuzufügen. Folgende Module sind vorhanden:

- RT-Linux Kern Modul (Kernel Patch),
- `rtl_sched` prioritätsgesteuertes Scheduling Modul,
- `rtl_time` kontrolliert die Prozessor-Zeit, abstraktes Timer-Interface

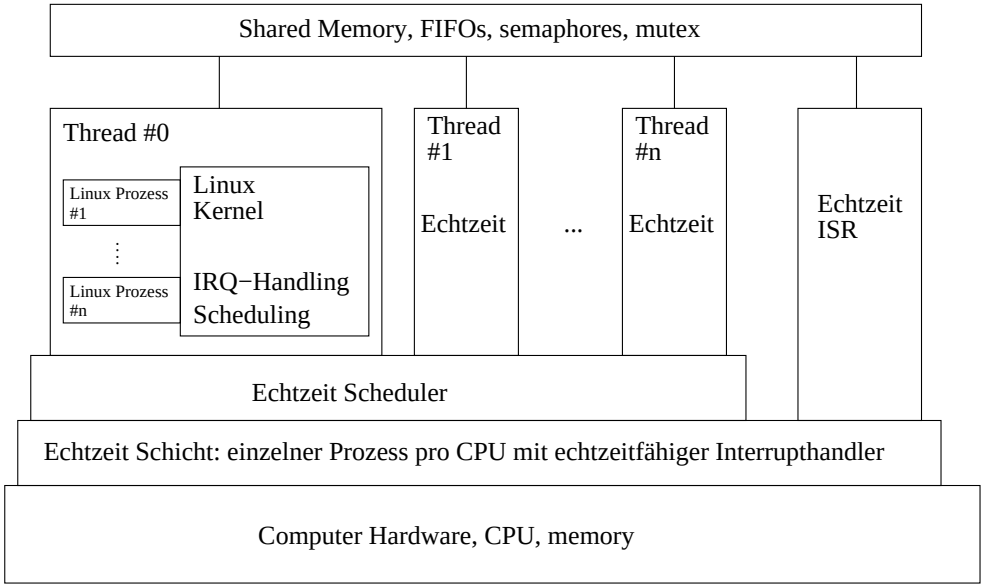

Abbildung 2.3: RT-Linux Architektur

- `rtl_posixio` liefert POSIX konformes Read/Write-Interface für Gerätetreiber
- `rtl_fifo` RT-FIFOs zwischen Kernel- und User-Space
- `mbuff` Kommunikation über shared memory zwischen Kernel- und User-Space
- `semaphore` Unterstützung für Task-blockierende Semaphoren

Eine Ausnahme ist das RT-Linux Kernel Modul, da es nicht zur Laufzeit wie ein Modul, sondern als Kernel Patch in das System eingebracht wird. Das verlangt ein anschließendes kompilieren des Kernels. Ansonsten kann RT-Linux zur Laufzeit durch das Modulkonzept in beliebiger Weise skaliert werden.

2.3.2 Interrupts

Das größte Problem von Linux 2.0/2.2 ist, daß der Kernel zeitweise aus Synchronisationszwecken Interrupts sperrt, was mitunter in der Entwicklungsgeschichte von Linux begründet ist. Daher werden im Linux Kernel häufig Funktionen wie cli() und sti() verwendet, was hohe Interrupt-Latenzzeiten zur Folge haben kann. Durch den Einschub der RT-Linux Schicht wird die bisherige Verbindung zwischen dem Interrupt Controller und dem Linux Kernel unterbrochen, indem alle cli(), sti() und iret() Befehle im Linux Kernel durch folgende emulierte Makros ersetzt werden: r_cli, r_sti, r_iret Auf diese Weise werden alle Interrupts von der RT-Linux Schicht abgefangen und können nach Belieben zurückgehalten oder an Linux weitergeleitet werden. Wenn in Linux nun Interrupts gesperrt werden, wird lediglich eine Variable, auf 0 gesetzt. Bei Auftreten eines Interrupts, überprüft der Emulator diese Variable; wenn sie ungleich Null ist, d.h. wenn Linux Interrupts erlaubt, wird der Interrupt sofort an den RT-Task “Linux“ weitergeleitet, so daß dort wie gewöhnlich die ISR gestartet werden kann. Hat der Linux Kernel Interrupts gesperrt, wird jedes

Auftreten eines Interrupts in einer Variablen im Emulator gespeichert und später an Linux übergeben. Interessant ist dieses eingeschränkte Handlungsvermögen von Linux jetzt für die parallel laufenden RT-Tasks. Diese können, von Linux unberührt, Interrupts zu jeder Zeit abfangen und sie abarbeiten, ohne Gefahr zu laufen, daß im Emulator Interrupts verzögert werden oder verloren gehen.

2.3.3 RT-Interrupt-Handler

RT-Tasks und normale Linux Kernelfunktionen können einen Interrupthandler anmelden und eine eigene Echtzeit-ISR zur Verfügung stellen. Wenn nun eine RT-ISR für einen bestimmten Interrupt angemeldet worden ist und dieser eintrifft, so wird diese sofort gestartet, unabhängig davon, was der niedrig priore RT-Task “Linux” zur Zeit macht. Das führt zu einem optimierten Verhalten der Reaktion auf externe, asynchrone Ereignisse. Selbst ein Aufruf von disable_irq() kann keine Interrupts maskieren, so dass keine Interrupts verloren gehen können.

2.3.4 Scheduler

Die Hauptaufgabe eines Echtzeitschedulers liegt darin, die Zeitanforderungen der Echtzeitprozesse zu befriedigen. In RT-Linux wird auch der Scheduler als Modul in das System eingebracht. So kann jederzeit eine neue Implementierung eines Schedulers schnell und unkompliziert ausprobiert werden. Der Standard Scheduler arbeitet nach einer eindeutigen Prioritätenvergabe, d.h. jede Priorität ist einzigartig. Immer, wenn ein höher priorer Prozeß rechenbereit ist, bekommt er die CPU und die anderen Prozesse werden sofort unterbrochen. Von jedem Prozeß wird dabei erwartet, daß er die Kontrolle freiwillig in einem solchen Fall abgibt. Der Scheduler unterstützt periodische Prozesse, so daß der Startzeitpunkt und die Dauer der Ausführung für jeden Prozeß angegeben werden muß.

2.3.5 RT-Linux Interprozeßkommunikation

Um zwischen RT-Linux und Linux Prozessen kommunizieren zu können, muß ein Mechanismus bereit gestellt werden, der diese Aufgabe übernimmt. Dafür wurden sogenannte Realtime FIFOs implementiert, die im Kernel Adreßspeicher liegen. Von Linux aus kann auf sie über die Geräteschnittstelle /dev/rtf* zugegriffen werden. Das Interface von RT-Linux stellt Erzeugung, Zerstörung der FIFOs, Lesen und Schreiben aus bzw. in die FIFOs zur Verfügung. Um von einem Benutzerprozeß auf einen FIFO zuzugreifen, stehen die Funktionen “open()”, “close()”, “read()”, “write()” für das jeweilige FIFO Gerät (/dev/rtf*) zur Verfügung. Neben den FIFOs können als zweite Möglichkeit zur Kommunikation zwischen Kernel- und User-Space Shared Memory Bereiche angelegt werden. Dabei greifen RT-Prozesse über einen Zeiger auf den Speicher zu, wohingegen Linux Prozesse über das Device /dev/mbuff oder mittels der “mmap()” Funktion den Speicherbereich in ihren privaten Adreßbereich einblenden.

2.3.6 RT-Linux-API

Die RT-Linux Version 2 ist ein kompletter Umbau der ursprünglichen Version 1. Einige der Änderungen betreffen Unterstützung für SMP, Möglichkeit auf größeren Systemen zu laufen, und eine einfachere Benutzbarkeit. Die folgende Liste zeigt die wichtigsten Funktionen der RT-Linux API seit Version 2.0:

Prozessinitialisierung, -zerstörung, -scheduling, -kontrolle

`rt_task_init()`:
: Erzeugung eines Echtzeitprozesses und dessen Stacks.

`rt_task_delete()`:
: Löscht die Struktur eines Echtzeitprozesses und dessen Stacks.

`rt_task_make_periodic()`:
: Setzen eines Echtzeitprozesses auf periodisch “ausführbar”

`rt_task_wait()`:
: Beendet Ausführung eines periodisch wiederkehrenden Prozesses bis zu Beginn seiner nächsten Periode.

`rt_task_suspend()`:
: Suspendiert Echtzeitprozeß. rt_task_make_periodic() oder rt_task_wakeup() müssen aufgerufen werden, um ihn wieder zu aktivieren.

`rt_task_wakeup()`:
: Setzt einen nicht periodischen Prozeß auf “ausführbar”.

Interruptverwaltung:

`rtl_request_irq()`:
: Verknüpft einen Interrupt-Handler mit einem spezifizierten Interrupt.

`rtl_free_irq()`:
: Gibt zuvor angelegten Handler wieder frei.

`rtl_hard_enable_global_irq()`:
: erlaubt Interrupt-Kontrolle

`rtl_hard_disable_global_irq()`:
: sperrt Interrupt-Kontrolle

`rtl_allow_interrupts()`:
: erlaubt Hardware-Interrupts auf dem aktuellen Prozessor

`rtl_stop_interrupts()`:
: sperrt Hardware-Interrupts auf dem aktuellem Prozessor

`rtl_no_interrupts()`:
: sperrt Hardware-Interrupts auf dem aktuellem Prozessor und speichert aktuellen Interrupt-Zustand

`rtl_save_interrupts()`:
: speichert aktuellen Interrupt-Zustand

`rtl_restore_interrupts()`:
: stellt den zuvor gespeicherten Interrupt-Zustand wieder her

`rtl_get_soft_irq()`:
: installiert einen Soft-Interrupt-Handler

`rtl_free_soft_irq()`:
: entfernt einen Soft-Interrupt-Handler

`rtl_global_pend_irq()`:
: schickt einen globalen Soft-Interrupt an Linux

`rtl_local_pend_irq()`:
: schickt einen lokalen Soft-Interrupt an Linux

Erzeugung und Benutzung von FIFOS:

`rtf_create()`:
: Erzeugt einen FIFO; (Dieser muß zuvor unter /dev/rtf(x) angelegt worden sein).

`rtf_create_handler()`:
: Verknüpft eine Funktion mit FIFO.

`rtf_destroy()`:
: Löscht zuvoror angelegten FIFO.

`rtf_get()`:
: Ließt aus einem FIFO.

`rtf_put()`:
: Schreibt in einen FIFO.

Posix - Threads

`pthread_create()`:
: erzeugt einen Thread

`pthread_exit()`:
: exit

`pthread_delete_np()`:
: stopt einen Thread, und gibt seine Resourcen wieder frei

`pthread_setfp_np()`:
ermöglicht oder sperrt Floating-Point Operationen im Thread

`pthread_wakeup_np()`:
reanimiert einen zuvor aufgehängten Thread

`pthread_suspend_np()`:
setzt ein Thread vorübergehend aus

`pthread_wait_np()`:
thread wird ausgestzt bis zu Beginn seiner der nächsten Periode

`pthread_make_periodic_np()`:
Thread wird zyklisch mit einstellbarer Periodendauer ausgeführt

`pthread_attr_init()`:
Attribute für Thread-Erzeugung

`pthread_att_getcpu_np()`:
Unterstützung für SMP, holt Prozessor-Nr.

`pthread_att_setcpu_np()`:
Unterstützung für SMP, setzt Prozessor-Nr.

`pthread_t pthread_self()`:
liefert Thread-ID

`pthread_attr_setschedparam()`:
schreibt Scheduling-Parameter

`pthread_attr_getschedparam()`:
liest Scheduling-Parameter

`sched_get_priority_max()`:
liefert das Prioritäten-Limit des Schedulers

`sched_get_priority_min()`:
liefert das Prioritäten-Limit des Schedulers

`pthread_kill()`:
sendet Kill-Signal an einen Thread

`pthread_mutex_*()`:
RTLinux mutex Implementierung, diverse Befehle

Diverses

`rtl_printf()`:
Ausgabe an Bildschirm

Kapitel 3

Beschreibung der Hardware-Komponenten

3.1 Der GRIP-Roboter

3.1.1 Der mechanische Aufbau

Der insgesamt nur 12 kg schwere Roboterarm besteht aus einer Hybridstruktur aus tragenden Elementen, Antrieben, Getriebe und Verkleidung. Der ”künstliche Arm” soll mit Steuerung und Energieversorgung an einem Rollstuhl montiert werden und den Behinderten bei alltäglichen Aufgaben, wie zum Beispiel Heben von Gegenständen, Öffnen von Türen oder beim Essen unterstützen. Dazu ist am vorderen Ende des Arms ein Greifer angebracht.

Abbildung 3.1: Der GRIP-Roboter

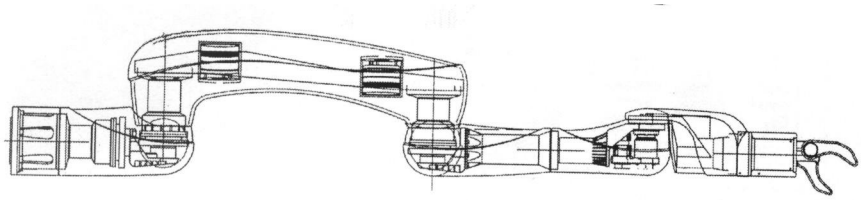

Abbildung 3.2: Schematische Darstellung des Greifarms

3.1.2 Die Antriebe

Die CAN-Antriebseinheiten der Firma Wittenstein Motion Control bestehen aus einem Servomotor mit Planetengetriebe und elektromagnetischer Bremse sowie einem Servocontroller. Um der Anforderung nach einer modularen und kompakten Bauweise nachzukommen, wurden die Servocontroller für die digitalen Antriebe direkt in den Roboterarm integriert. Auf runden Platinen befindet sich die komplette Leistungselektronik, Betriebsüberwachung und Drehmomentenregelung. In Sandwichbauweise aufgesteckt ist jeweils ein weiterer μC der für die Drehzahl- und Positionsregelung zuständig ist.

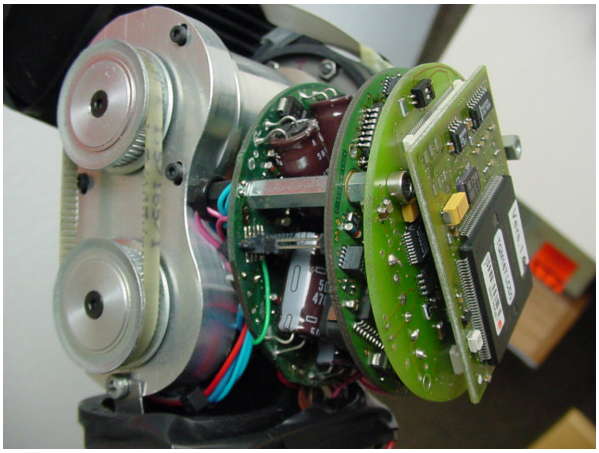

Abbildung 3.3: Photo eines Antriebs-Controllers

Durch den Einsatz des CAN-Buses ist es möglich den Roboterarm komplett modular aufzubauen. Weitere Antriebe und Gelenke können einfach integriert werden. Der Anschluss erfolgt über den CAN-Bus störungssicher und mit minimalem Verkabelungsaufwand. Durch die verteilte Intelligenz, kann der Roboter im Fehlerfall autonom reagieren, indem er zum Beispiel die Bremse setzt.

3.1.3 Das CAN-Interface des GRIP-Roboters

Die Antriebe stellen ein Interface zur Verfügung, über das die komplette Kommunikation mit dem CAN-Bus geregelt wird. Dieses Interface besteht im wesentlichen aus CAN-Botschaften, die von den Antrieben gesendet werden, oder an sie geschickt werden müssen. Die Bezeichnug der Kommunikations-Botschaften lauten:

- Transmit - PDO
- Receive - PDO
- SDO-Message
- Sync - Message
- Emergency - Objekt

Jeder Antrieb besitzt für jede dieser Nachrichten einen eigenen, fest vereinbarten Identifier. So sind alle Achsen voneinander unterscheidbar, und einzelne Achse können gezielt angesprochen werden. Der Aufbau dieser Botschaften wird im foldenden erläutert.

Transmit PDO

Das Transmit-PDO dient zur Übertragung von Ist-Werten kommend von den Antrieben an die Steuerung. Diese Botschaft wird gesendet, sobald ein Antrieb das Sync-Telegramm empfangen hat. Der Antriebsregler speichert seine Istwerte, und führt sie der Übertragung zu. Wollen mehrere Antriebe das Transmit PDO senden, entscheidet die Prioriät der Identifier, wer auf den Bus zugreifen darf. Alle Zustandsgrößen, sowohl Ist- als auch Sollwerte werden als 16 Bit Wert gesendet. Das Transmit-PDO ist eine Nachricht mit einer Datenlänge von 8 Byte und hat folgenden Aufbau:

Byte 0		Positionsistwert Bit 0 - 7
Byte 1		Positionsistwert Bit 8 -15
Byte 2		Drehzahlistwert Bit 0 - 7
Byte 3		Drehzahlistwert Bit 8 - 15
Byte 4		Drehzahlregler Stellgröße Bit 0 - 7
Byte 5		Drehzahlregler Stellgröße Bit 8 -15
Byte 6		Antriebsstatuswort
Byte 6	Bit 0	Betriebsbereit
	Bit 1	Enable-Out
	Bit 2-6	Reserve
	Bit 7	Fehler
Byte 7		Reserve

Tabelle 3.1: Aufbau des Transmit PDO

Das Antriebsstatuswort in Byte 6 meldet die Betriebsbereitschaft der jeweiligen Achse. Das Betriebsbereit-Signal wird nach einer fehlerfreien Initialisierung des Antriebs gesetzt und nur durch einen kritischen Fehler zurückgesetzt. Nach dem Wegfall des Fehlerzustands und der Fehlerquittierung durch die Steuerung ist die Achse wieder betriebsbereit. Das Signal Enable-Out zeigt an, daß die Achse in Regelung ist. Im Fall eines kritischen Fehlers wird das Fehler Signal gesetzt. Die Signale Betriebsbereit und Enable-Out werden dann zurückgesetzt. Ist die Ursache für den Fehler behoben, kann das Fehlersigal über Enable-In wieder zurückgesetzt werden.

Receive PDO

Möchte man Soll-Werte an die Antriebe übertragen schicket, so muß ein Receive-PDO gesendet werden. Entsprechend der jeweiligen Betriebsart werden hier Sollwerte in die entsprechenden Bytes der Nachricht eingetragen und gesendet. Im Falle einer Drehzahl- oder Drehmomentenregelung werden die Sollwerte direckt in die Antriebe übernommen, während bei Positionsregelung erst das nächst Sync-Objekt abgewartet wird. Das ebenfalls aus 8 Bytes bestehende Receive-PDO hat den folgenden Aufbau:

Byte 0		Positionssollwert Bit 0 - 7
Byte 1		Positionssollwert Bit 8 -15
Byte 2		Drehzahlsollwert Bit 0 - 7
Byte 3		Drehzahlsollwert Bit 8 - 15
Byte 4		Stromsollwert Bit 0 - 7
Byte 5		Stromsollwert Bit 8 -15
Byte 6		Antriebssteuerwort
Byte 6	Bit 0-2	Betriebsartenumschaltung
	Bit 3	Enable-In
	Bit 4	Bremse extern lüften
	Bit 5	Fehler quittieren
	Bit 6	Position kalibrieren
	Bit 7	Bootstrap (nur für internen Gebrauch)
Byte 7		Reserve

Tabelle 3.2: Aufbau des Receive-PDO

Das Antriebssteuerwort in emöglicht die Betriebsart der Achse zu wählen. Weiterhin wird über Enable-In die Reglerfreigabe gegeben. Dieses Bit muß daher im laufenden Betrieb gesetzt sein. Mit dem Signal Position kalibrieren wird die vom Antrieb zurückgemeldete Position auf einen vorgegeben Wert gesetzt.
Die Antriebe unterstützen folgende Betriebsarten:

- Stillsetzen:
  Antrieb wird auf Drehzahl 0 geregelt

- Position:
  Antrieb wird auf den Positionssollwert aus Receive-PDO Byte 0-1 geregelt

- Drehzahl:
  Antrieb wird auf den Drehzahlsollwert aus Receive-PDO Byte 2-3 geregelt

- Position mit Vorsteuerung:
  Antrieb wird auf den Positionssollwert aus Receive- PDO Byte 0-1 geregelt, und mit dem Drehzahlsollwert aus Byte 2-3 und dem Stromsollwert aus Byte 4-5 vorgesteuert

- Drehzahl mit Vorsteuerung:
  Antrieb wird auf den Drehzahlssollwert aus Receive- PDO Byte 2-3 geregelt, und mit dem Stromsollwert aus Byte 4-5 vorgesteuert

- Drehmoment:
  Antrieb wird auf den Stromsollwert aus Receive-PDO Byte 4-5 geregelt

Die Aufschlüsselung der Betriebsartenumschaltung innerhalb von Byte 6 ergibt sich folgendermaßen:

Bit 2	Bit 1	Bit 0	Betriebsart
0	0	0	Stillsetzen
0	0	1	Drehzahl
0	1	0	Position
0	1	1	Position + Vorsteuerung
1	0	0	Drehmoment
1	0	1	Drehzahl + Drehmoment-Vorsteuerung
1	1	0	Reserve
1	1	1	Reserve

Tabelle 3.3: Zuordnung der Bits 0-2 zu den Betriebsarten

SDO

Mit der SDO-Nachricht erfolgt die Parametrierung der Antriebe. Man hat Zugriff auf ein 16 Byte großes Parameterfeld im Antrieb auf das Schreibend oder Lesend zugegriffen werden kann. Die Zuordnung der Parameter zeigt die folgende Tabelle:

Subindex	Inhalt
1	Drehzahlgrenzwert N_{Limit}
2	Stromgrenzwert I_{Limit}
3	Positionsregler Proportionalverstärkung K_v
4	Reserve
5	Drehzahlregler Proportionalverstärkung K_p
6	Drehzahlregler Integralverstärkung K_i
7	Drehzahlregler Filter Zeitkonstante T_F
8	Drehzahlregler Struktur M_N
9	Reserve
10	Reserve
11	Reserve
12	Reserve
13	Reserve
14	Reserve
15	Kalibrier-Position Bit 0 - 7
16	Kalibrier-Position Bit 8 -15

Tabelle 3.4: SDO-Parameterfeld

Sync-Telegramm

Das Sync-Telegramm dient dazu, daß die Antriebe ihre übermittelt bekommenen Sollwerte in die Antriebe übernehmen. Zeitgleich werden von den Meßsystemen der Antrieb die internen Zustandgrößen (Istwerte) ermittelt, und die Übertragung eines Transmit-PDOs angestoßen. Das Sync-Objekt wird von allen Antrieben angenommen. Der Identifier lautet 0x80, die Datenlänge ist 0.

Emergency-Objekt

Tritt in der Achse ein kritischer Fehler auf, der zu einer Momentenfreischaltung des Antriebes führt, sendet der Antrieb das hochpriore Emergency-Objekt. Die Fehlermeldungen werden aus der Antriebs-Hardware abgeleitet oder durch die Antriebssoftware generiert. Für die Übermittlung der Hardware- und Softwarefehler des Antriebs wird im Emergency Error Code (Byte 0 -1) ”Device specific FF00” gesendet. Die Hardwarefehler werden in Byte 3, die Softwarefehler in Byte 4 übermittelt. Der Aufbau des kompletten Emergency-Objekts zeigt die folgende Tablelle:

Byte 0		Emergency Error Code
Byte 1		
Byte 2		Error Register
Byte 3		Hardware Fehler
	Bit 0	Resolverfehler
	Bit 1	Überspannung Endstufe
	Bit 2	Übertemperatur Motor
	Bit 3	Übertemperatur Endstufe
	Bit 4	Überstrom
	Bit 5	Erdschluß
	Bit 6	CPU-Fehler
	Bit 7	Reserve
Byte 4		Software Fehler
	Bit 0	BUS-Fehler
	Bit 1	Speicherfehler
	Bit 2	Maximaldrehzahl überschritten
	Bit 3	Schleppabstand zu groß
	Bit 4	Reserve
	Bit 5	Reserve
	Bit 6	Reserve
	Bit 7	Reserve
Byte 5		Reserve
Byte 6		Reserve

Byte 7		Leer

Tabelle 3.5: Beduetung der Bits im Emergency-Objekt

Zuordnung von Identifiern zu den CAN-Antrieben
Die Identifier der Achsen sind fest eingestellt. Folgende IDs müssen verwendet werden:

Achs-Nr	Receive-PDO	Transmit-PDO	SDO	Emergency
1	0x201	0x181	0x581	0x81
2	0x202	0x182	0x582	0x82
3	0x203	0x183	0x583	0x83
4	0x204	0x184	0x584	0x84
5	0x205	0x185	0x585	0x85
6	0x206	0x186	0x586	0x86
7	0x207	0x187	0x587	0x87

Tabelle 3.6: CAN-Identifier der Antriebe

Achsparameter
Für die Parametrierung der Antriebe werden folgende Werte an die Achsen gesendet:

		Achse 1	Achse 2	Achse 3	Achse 4	Achse 5	Achse 6	Achse 7
Drehzahlgrenzwert	N_{Lim}	255	255	255	255	255	255	255
Stromgrenzwert	I_{Lim}	100	255	255	50	255	30	100
Positionsregler Proportionalverstärkung	K_v	40	50	40	40	40	40	40
Drehzahlregler Proportionalverstärkung	K_p	255	200	200	100	100	100	100
Drehzahlregler Integralverstärkung	K_i	100	100	100	50	50	50	50
Drehzahlregler Filter Zeitkonstante	T_F	200	200	200	200	200	200	200
Drehzahlregler Struktur	M_N	0	0	0	0	0	0	0

Tabelle 3.7: Achsparameter

3.2 Der CAN-Joystick

Um eine einfache Bedienung des Greifarms zu ermöglichen, wird ein Joystick verwendet. Der CAN-Joystick PGJC6LK der Firma Penny & Giles ist ein Joystick mit 3 analogen (Auslenkungen) und 3 digitalen (Knöpfe) Eingängen. Alle Informationen werden von der Elektronik des Joysticks ausgewertet und über eine CAN 2.0B-Schnittstelle als 8-Byte Datenframe zur Verfügung gestellt. Über umfangreiche Diagnose- und Konfigurationsmöglichkeiten ist es möglich den Joystick zu parametrieren. Der Datenframe wird mit einer einstellbaren Zeitspanne wiederholt gesendet, wobei die Übertragungsrate zwischen 10 Kbit/sec und 1Mbit/sec eingestellt werden kann. Wir verwendeten die Einstellung kontinuierliches Senden alle 10 ms. Die drei analogen Achsen werden über eine Vor/Zurück, bzw. Rechts/Links-Auslenkung, sowie über eine am oberen Ende des Joysticks angebrachte Wippe gesteuert.
Nachfolgend ist der Aufbau des Datenframes dargestellt, der vom Joystick im Betrieb regelmäßig gesendet wird:

ID	11bit oder 29 bit
Byte	Bedeutung
1	Local ID
2	Diagnose
3	Tastenzustand
4	Richtung
5	Analog_1 (X)
6	Analog_2 (Y)
7	Analog_3 (Z)
8	Analog_4

Tabelle 3.8: Aufteilung der Joystick-Daten im PDO

Die ID ist der CAN-Identifier mit welcher der Joystick seine Botschaften sendet. Sie ist frei einstellbar. Die Länge ist im Normalfall 11 Bit und bei Verwendung des *extended*-Formats 29 Bit. Der Defaultwert ist 0x400 mit 11 Bit.
Die Local-ID ermöglicht die Identifizierung eines einzelnen Joysticks, wenn in einem CAN-Netzwerk mehrere Joysticks vorhanden sind, und diese mit dem selben Identifier senden. Dabei bedeutet eine niedrigere Local-ID, eine höhere Priorität. Default-Wert der Local-Id ist 0x0.
Das Diagnose-Byte gibt Auskunft über den Zustand des Joysticks. Sobald ein Fehler auftritt wird das entsprechende Bit gesetzt. Fehlerfälle können sein: Power-fail, Hardware-Fehler, Analoggeber defekt, Node-Guarding Fehler, EPROM-Fehler.
Im 3. Byte wird der aktuelle Tastenzutand übertragen. Ein gesetztes Bit entspricht einer betätigten Taste. Die Tasten werden softwaremäßig entprellt.

Bit	Bedeutung
7	Taster/Schalter 8
6	Taster/Schalter 7
5	Taster/Schalter 6

4	Taster/Schalter 5
3	Taster/Schalter 4
2	Taster/Schalter 3
1	Taster/Schalter 2
0	Taster/Schalter 1

Tabelle 3.9: Bedeutung der Bits im Tastenbyte

Die CAN-Elektronik ermöglicht eine Auswertung von bis zu 8 Tasten. Das vorliegende Modell verwendet aber nur deren 3. So das die Taster 4 bis Taster 8 keine Bedeutung besitzen.
Byte 4 enthält die Richtungsinformation der Analoggeber. Diese Information ermöglicht eine Unterscheidung der Bewegungsrichtung einer Achse (Bsp: links oder rechts). Die Richtungsbits sind nach folgendem Schema zugeordnet.

Bit	Bedeutung
7	Reserviert
6	Reserviert
5	Reserviert
4	Reserviert
3	Analog 4
2	Analog 3 (Z)
1	Analog 2 (Y)
0	Analog 1 (X)

Tabelle 3.10: Bedeutung der Bits im Richtungsbyte

Auch hier ist es möglich mit der Elektronik mehr Analogsignale auszuwerten als tatsächlich vorhanden sind. Deshalb wird Bit 3 nicht berücksichtigt (siehe ebenfalls Tabelle 3.8).
Nimmt man eine Entschlüsselung der Richtungsbits in reale Bewegungsrichtungen vor, dann erhält man:

Analog 1 (X)	1 = rechts	0 = links
Analog 2 (Y)	1 = senken	0 = heben
Analog 3 (Z)(Wippe)	1 = rechts	0 = links

Tabelle 3.11: Richtungsbits und entsprechende Bewegungsrichtungen

Die Bytes 5..8 enthalten das jweils 8 Bit breite Aussteuerungssignal der Joystickachsen. Zusammen mit dem Richtungsbit ergibt sich für jede Achse eine Auflösung von 510 ($2 \cdot 2^8$) möglichen Werten.

3.3 Das PC/104-System

Die eigentliche Bahnsteuerung erfolgt auf einem PC/104 System. Dieses System ist ein kompletter PC mit CPU, Speicher, seriellen und parallelen Schnittstellen, Floppy- und IDE-Interface, sowie Tastatur. Der Vorteil dieses Systems liegt in seiner Bauweise. Es gibt kein Motherboard, sondern alle Komponenten befinden sich auf Karten des Formats 90 x 96 mm. Zum Ausbau der Funktionalität können mehrere Karten übereinandergesteckt werden, so dass sich sehr kompakte, platzsparende ”Würfel” gestalten lassen. Die Verbindung der einzelnen Karten geschieht durch ein einfaches Übereinanderstecken mit einen 104 poligen Stecker. Der PC/104 -Bus erfüllt die gleichen elektrischen Spezifikationen wie der ISA-Bus erfüllt.

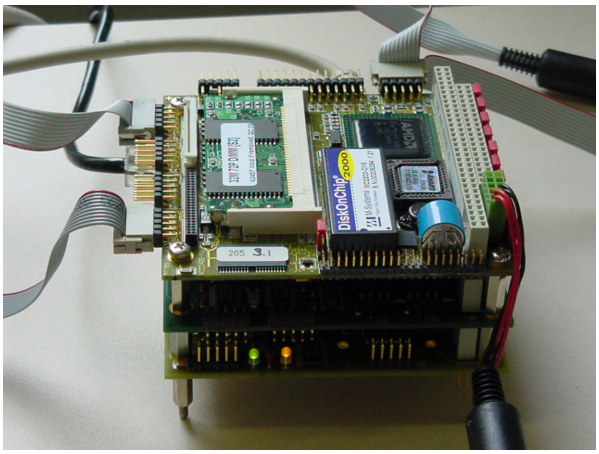

Abbildung 3.4: Photo des PC/104-Systems

Auf der vorliegenden Baugruppe HiCO486 der Firma Hitex befindet sich ein AMD Elan Prozessor. Dieser basiert auf der Architektur eines 486 SX2 und arbeitet mit einer Taktfrequenz von 66 MHz. Er besitzt allerdings keinen mathematischen Co-Prozessor, was in Bezug auf Floating-Point Operationen im Linux-Kernel ein großer Nachteil ist. Das System zeichnet sich durch eine geringe Leistungsaufnahme von unter 3 Watt aus, so dass ein Betrieb bis 70 °C ohne Kühlung möglich ist. Bei einem eingeschränkteren Temperaturbereich ist auch eine Umschaltung auf 100 MHz möglich. Die Größe des Hauptspeichers beträgt 32 MByte. Als Speichermedium kommt eine Flash-Disk der Firma M-Sys zum Einsatz. Diese kann leicht ausgetauscht werden, und ist bis zu einer Größe von 72 MByte erhältlich. Weiterhin vorhanden ist ein VGA-Controller, sowie als Anschlüsse und Schnittstellen:

- VGA-Monitor
- Flachbildschirm
- Tastatur/Maus
- 3 x serielle Schnittstelle
- Parallelschnittstelle

- Floppy-Disk
- IDE-Schnittstelle
- IrDA
- TTL-Ausgabe

Das Schaubild 3.5 zeigt eine schematische Darstellung PC/104-Systems mit einer Beschriftung der vorhandenen Schnittstellen und Anschlüsse.

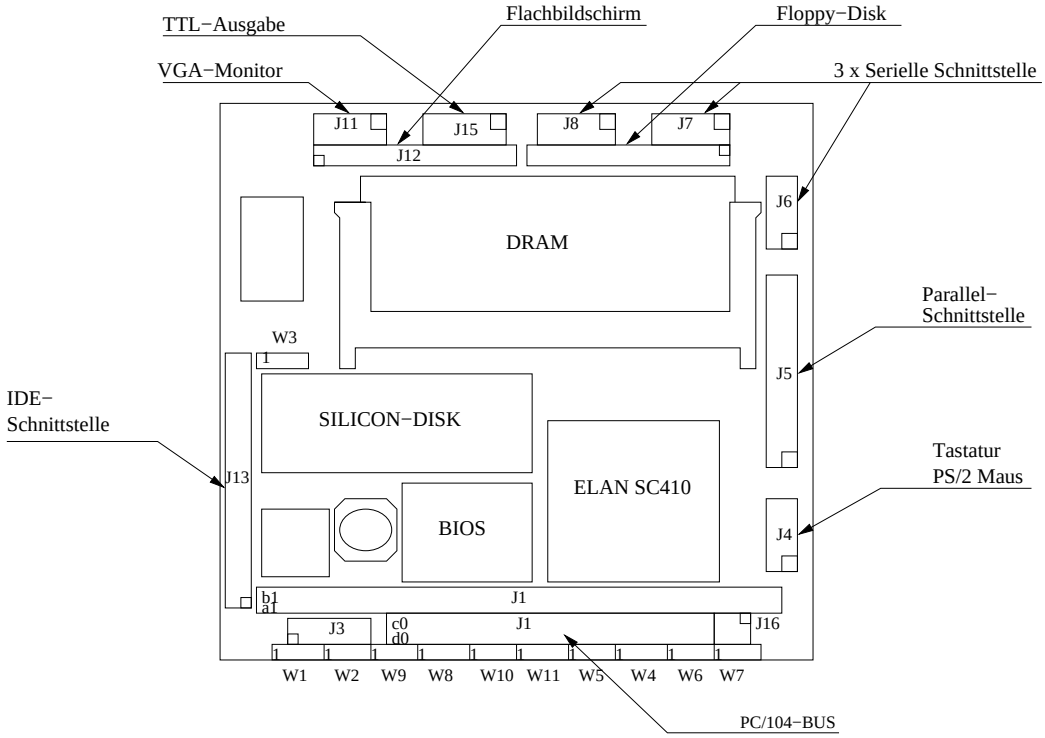

Abbildung 3.5: Darstellung des PC/104-Systems mit seinen Anschlüssen

Die zweite Platine, die über den PC/104 Bus aufgesteckt wurde, ist eine NE2000 kompatible Netzwerkkarte. Sie wird zur Kommunikation mit dem Entwicklungssystem verwendet.
Sowie als drittes Modul die CAN-Karte IXXAT PC-I 04/104. Dies ist eine passive CAN-Karte für den PC/104-Bus mit einem CAN-Kanal. Sie verwendet den Philips SJA1000 CAN-Controller, welcher im vorhergehenden Kapitel beschrieben wurde.

3.4 Das Gesamtsystem

Den Aufbau des gesamten System veranschaulicht Bild 3.6:
Der Greifarm sowie der Joystick sind über CAN-Bus mit dem PC/104 - System verbunden. Weiterhin kommt ein CANalyzer zum Einsatz, der zur Analyse dient. Mit ihm ist es möglich gesendete Nachrichten anzuzeigen, sowie das Zeitverhalten der CAN-Nachrichten zu analysieren. Er

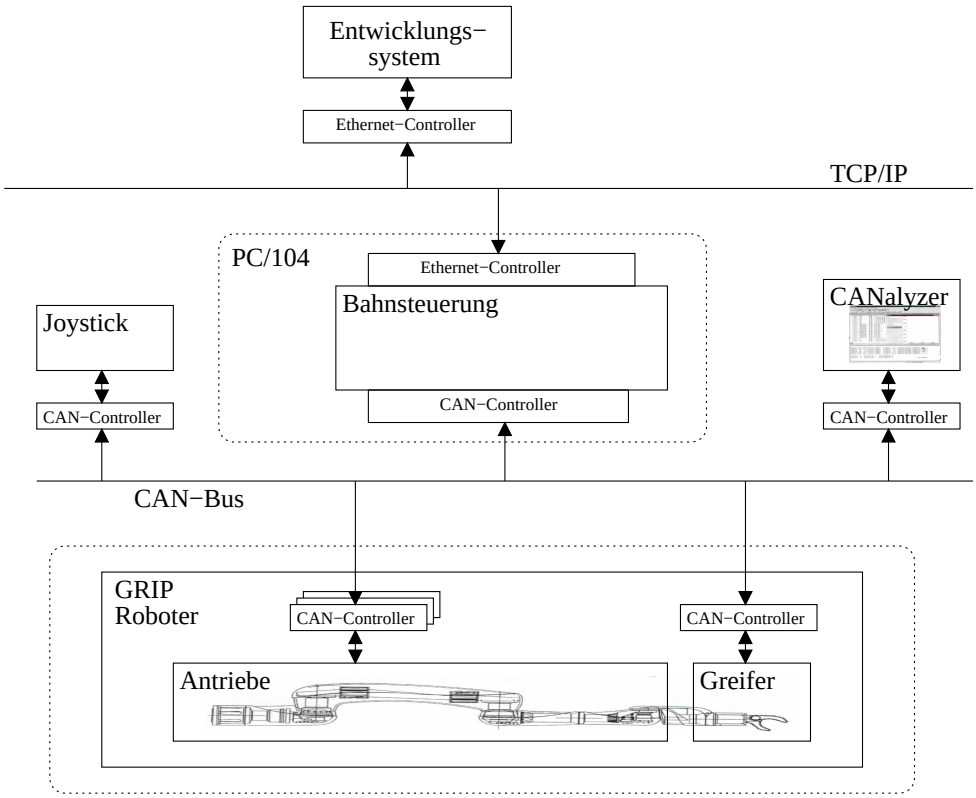

Abbildung 3.6: Schematische Darstellung des Gesamtaufbaus

kann aber auch selber Nachrichten senden. Die Steuerung läuft auf dem PC/104-System, das die Nachrichten des Roboters sowie die des Joysticks in regelmäßigen Abständen auswertet, und anschließend Sollwerte an die Achsen des Roboters schickt. Zur Entwicklung der Software diente das Entwicklungssystem, welches über eine Ethernet-Verbindung an das PC/104 System angeschlossen ist. Die entwickelte Software wurde auf dem Entwicklungssystem geschrieben und kompiliert, und anschließend auf das Target-System übertragen. Am Ende der Entwicklungsphase wird das Entwicklungssystem und der CANalyzer abgekoppelt, und es findet ein alleiniger Betrieb von Greifarm, Joystick, und PC/104 statt.

Kapitel 4

Vorgehensweise bei der Installation von RT-Linux

4.1 Einleitung

Im Rahmen des Projekts GRIP hat man als Steuerungsrechner ein PC/104-System gewählt. Dieser Abschnitt beschreibt die Installation und Inbetriebnahme der Hard- und Software, zuerst auf dem Entwicklungsrechner, und anschließend auf dem PC/104-System. Das Ergebnis ist eine auf beiden Rechnern laufende RT-Linux Plattform sowie die Möglichkeit mittels Ethernet miteinander zu kommunizieren.
RT-Linux auf dem PC/104-System dient dann als Betriebssystem für das zu entwickelnde Steuerungssystem für den Greifarm GRIP. Das Design dieser Steuerung benötigt ein echtzeitfähiges Betriebsystem.

4.2 Prinzipielle Vorgehensweise

Die durchgeführte Installation und die Dokumentation in diesem Abschnitt (in der angegebenen Reihenfolge) umfassten folgenden Schritte:

1. Hardware-Konfiguration
2. Installation von RT-Linux auf dem Entwicklungssystem
    - Installation und Konfiguration Linux auf Entwicklungssystem
    - Installation von RT-Linux auf Basis des vorher installierten Kernels auf dem Entwicklungssystem
3. Installation von RT-Linux auf dem PC/104-System
    - Booten des PC/104-Systems über die IDE-Schnittstelle mit Hilfe des zuvor installierten Entwicklungssystems
    - Mounten der Flash-Disk und installieren eines File-Systems

- Kopieren des RT-Kernels und der notwendigen Betriebssystemkomponenten
- Booten des PC/104-Systems von Flash-Disk

4. Konfiguration des Netzwerkes

Die so verfolgte Vorgehensweise führt zum Erfolg. Bei den ersten Versuchen RT-Linux zu installieren, wurde versucht über eine Boot-Diskette, das PC/104-System zu starten. Dann eine Netzwerkverbindung zum Entwicklungssystem aufzubauen, um von dort aus die Flash-Disk zu mounten und alle erforderlichen Files zu kopieren. Diese eigentlich vom Ablauf her einfachere Variante ist gescheitert, da alle verwendeten Boot-Disketten keine Floating-Point-Emulation im Kernel emulierten. Dies ist aber bei dem uns vorliegenden Prozessor AMD Elan, ohne mathematischen Co-Prozessor notwendig. Wurde versucht von der Boot-Diskette zu starten, so blieb das System beim Booten stehen. Etliche Versuche einen eigenen Kernel mit Floating-Point-Unterstützung zu kompilieren schlugen fehl, da es nicht möglich war ein komplettes Linux-System auf einer Diskette unterzubringen, um damit zu booten, und eine Netzwerkverbindung herzustellen.
Nachdem diese Versuche fehlgeschlagen sind, suchte man nach Alternativen. Es bat sich die im PC/104 integrierte IDE-Schnittstelle an. Allerdings hatte diese Schnittstelle eine andere Größe als das von Festplatten bekannte herkömmliche Format, so daß zuerst ein Adapter besorgt werden mußte, der eine Verbindung ermöglichte.

4.3 Hardware-Konfiguration

Die CAN-, sowie die Netzwerkkarte wurden folgendermaßen konfiguriert:

Device	IRQ	I/O-Port
NE2000	5	0x240
CAN	6	0xD4000

Tabelle 4.1: Hardware-Einstellungen beim PC/104-System

4.4 Die Installation von RT-Linux auf dem Entwicklungssystem

Zuerst erfolgt die Installation des Betriebssystems auf dem Entwicklungsrechner. Dazu verwendete ich die RedHat-Linux Distribution in der Version 6.2 mit dem Kernel 2.2.14. Diese Installation selbst soll hierzu nicht näher erläutert werden. Nachdem auf dem Entwicklungsrechner ein lauffähiges Betriebssystem installiert wurde, ging es nun darum dieses System mit RT-Linux zu erweitern.
Es ist zu beachten, daß mit dem jetzt konfigurierten System, im nächsten Schritt das PC/104-System gebootet werden soll. Deshalb muß die durchgeführte Konfiguration stets schon im voraus das PC/104-System bedenken. Dies trifft insbesondere auf die nun folgende Erweiterung des Kernels zu. Dabei hat man in Bezug auf das PC/104-System folgende Punkte zu berücksichtigen:

- `Math Emulation`
- `NE2000/NE1000 support`
- `NFS server support`
- `M-Systems DOC device support`

Der erste Punkt erlaubt die Emulation von Floating-Point Operationen im Linux Kernel. Dies ist wichtig da das PC104-System über keinen mathematischen Co-Prozessor verfügt. Der zweite Punkt betrifft die im PC104 integrierte Netzwerkkarte. Möchte man mittels des Network File System (NFS) kommunizieren, was die Übertragung von Daten auf das PC104-System erleichtert, so ist NFS-Server-Unterstützung ebenfalls zu integrieren.
Der letzte Punkt ist sehr entscheidend. Um auf die Flash-Disk zugreifen zu können, muß ein Kernel-Modul mit DiskOnChip (Flash-Disk)-Unterstützung vorhanden sein. Dieses ist nicht standardmäßig im Kernel enthalten, und muß zuerst über einen Patch noch hinzugefügt werden.
Aus diesem Grund ist man bei der Auswahl der möglichen Kernel sehr eingeschränkt. Es sind zwei Patches einzubringen, die beide für die selbe Kernel-Version vorliegen müssen. Zum einen die RT-Linux Erweiterungen und zum zweiten der DiskOnChip-Patch. Dies war ein großes Problem, da sobald der erste Patch ordnugsgemäß eingebrachte, der zweite Patch nicht mehr funktionierte und umgekehrt. Die DOC-Chip Unterstützung ist bei neuen Kernel-Versionen ab Kernel 2.4 mittlerweile sogar standardmäßig integriert, es war aber auch bei dieser Kernel-Version, dann nicht möglich den RT-Linux-Patch einzubringen. Das Patchen schlug fehl, und deshalb war auch ein Kompilieren des Kernels nicht möglich.
Die Lösung brachten die mit RT-Linux pre-gepatchten Kernel-Quellen für den Kernel 2.2.14, die auf der RT-Linux Homepage unter `http://www.fsmlabs.com` zu bekommen sind. In Kombination mit dem DiskOnChip TrueFFS Treiber (Version 5.0.0) der Firma M-Systems (`http://www.m-sys.com`) war es möglich den Kernel zu patchen sowie zu kompilieren.
Nachdem man den neu erstellten Kernel ins `/boot`-Verzeichnis kopiert hat, sowie den Boot-Loader entsprechend konfiguriert hat, kann man Kernel booten.
Ist alles in Ordnung so sollte RT-Linux jetzt booten. Nun muss RT-Linux noch konfiguriert werden.

- `cd /usr/src/rtlinux`
- `make config`

Anschließend werden die benötigten Module und Treiber kompiliert.

- `make`
- `make devices`
- `make install`

`make devices` erzeugt die ”special files” `/dev/rtf*`, die zur Kommunikation mit dem RT-Linux Kernel benötigt werden. `make install` kopiert die erzeugten Dateien in die dafür vorgesehenen Verzeichnisse.
Für die Flash Disk müssen ebenfalls ”special files” angelegt werden. Mittels

- `mknod /dev/fla b 62 0`
- `mknod /dev/fla1 b 62 1`
- `mknod /dev/fla2 b 62 2`
- `mknod /dev/fla3 b 62 3`
- `mknod /dev/fla4 b 62 4`

62 steht für die `major device number`. Sie ist fest eingestellt, und muss verwendet werden.

Als Ergebnis erhält man ein Linux-Betriebssystem mit RT-Linux Erweiterung sowie allen nötigen Tools zu Entwicklung von Software (X-Window, Editoren, Compiler, ..). Zusätzlich ist im Kernel ein DOC-Treiber vorhanden, mit dem es möglich ist die Flash-Disk auf dem PC/104-System zu benützen. Sowie alle sonstigen Gerätetreiber die fürs PC/104-System notwendig sind.
Für eine genauere Installationsanleitung: Siehe [11] und [12], sowie [5]

4.5 Die Installation von RT-Linux auf dem PC/104-System

Es erfolgt nun die Installation des Betriebssytems auf dem PC/104-System. Hierbei handelt es sich im wesentlichen auch wieder um mehrere aufeinanderfolgende Anweisungen, dessen Vorgehensweise im folgenden erläutern werden soll.

1. Update der Firmware
   Da das PC/104-System standardmäßig direkt von der Flash-Disk bootet, muß wenn von Festplatte gebootet werden soll zuerst die Firmware der Flash-Disk, mit einer sog. Dummy-Firmware ersetzt werden. Dazu sind die DiskOnChip Utilities notwendig, die auf eine Boot-Diskette gespielt werden. Bootet man von der Diskette so lautet der Befehl um die Firmware zu ersetzen:

   - `dformat /win:d000 /s:doc50.exb /y`

2. Booten des PC/104-System mit kompiliertem Kernel von Festplatte
   Nachdem der Kernel mit DOC-Unterstützung kompiliert wurde, sowie die Firmware ersetzt wurde, kann von der Festplatte aus gebootet werden. Der DOC-Treiber erkennt die Flash-Disk.

3. Erstellen einer Partition auf der Flash-Disk
   Es kann nun mit `fdisk` eine Partition erzeugt werden.

   - `fdisk /dev/fla`
   - Alle existierenden Partitionen löschen
   - Neue Linux Partition erstellen
   - Disk-Typ auf `Linux native` umstellen

- Neue Konfiguration speichern
- Reboot

4. Erstellung eines Linux-Dateisystems auf der Flash-Disk
   Um die erzeugte Partition nutzen zu können ist es notwendig ein Linux ext2-Dateisystem zu erstellen. Dies geschieht mittels:

   - `mke2fs /dev/fla1`

   Das Datei-System kann nun gemountet und benutzt werden

   - `mkdir /mnt/flash`
   - `mount /dev/fla1 /mnt/flash`

   Von nun an ist es möglich die Flash-Disk als zusätzlichen Speicher zu benutzen.

5. Überspielen eines Root-Dateisystems
   Um das PC/104-System eingenständig zu betreiben muß ein ein komplettes Linux-System, mit allen erforderlichen Komponenten auf die Flash-Disk überspielt werden. Dies ist von Hand sehr mühselig. Für diesen Zweck stellt der DOC-Treiber ein Skript zur Verfügung, das diese Aufgabe übernimmt.

6. Update der Firmware
   Reboot nach Dos. Die zu Beginn eingespielte Dummy-Firmware muß nun wieder mit der Original-Firmware ersetzt werden.

   - `dformat /win:d000 /s:doc50.exb /first /update`

   Die Festplatte kann nun entfernt werden werden.

7. RT-Linux bootet von DiskOnChip

4.6 Netzwerkkonfiguration

Die Konfiguration des Netzwerkes bereitete keine Probleme. Sie erfolgt auf beiden Systemen nach der üblichen Vorgehensweise.
Folgenede Parameter wurden den beiden Hosts zugewiesen:

- Enwicklungssystem
  Hostname: FISW74
  IP-Adresse: 141.58.126.74

- PC/104-System
  Hostname: FISW34
  IP-Adresse: 141.58.126.34

Alle anderen Einstellungen (Domain, Nameserver, Gateway) wurden vom vorhanden Netzwerk übernommen. Das komplette Dateisystem des PC104-Systems kann mittels des Kommandos:
`mount FISW34://mnt/fisw34` unter `/mntfisw34`
eingebunden werden.

Kapitel 5

Das Echtzeitmodul

Es sollte nun ein Software-Modul realisiert werden, das in der Lage ist mit dem GRIP-Arm zu kommunizieren. Da dieses Modul bestimmte Anforderungen bezüglich Echtzeit zu erfüllen hat, wird dieses Modul als RT-Linux Modul entwickelt. Die folgende Abbildung zeigt die hierarchische Gliederung der Komponenten die für ein solches System notwendig sind:

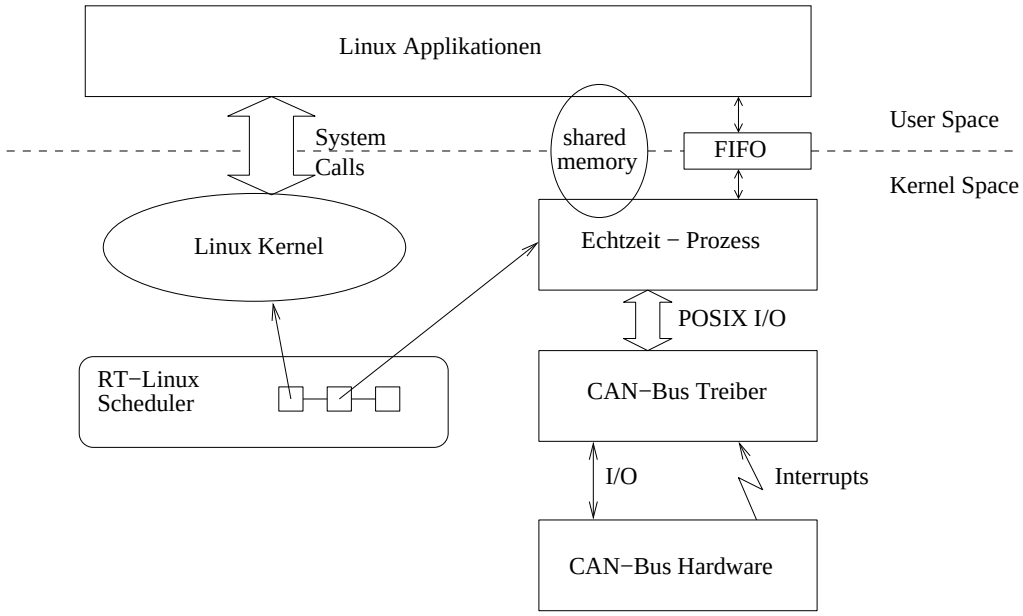

Abbildung 5.1: Hierarchischer Aufbau des Steuerungssystems

Der Aufbau einer RT-Linux Applikation ist in den meisten Fällen zwei-geteilt. Zum einen das eigentliche Echtzeitmodul, das aus einem oder mehreren Echtzeit-Tasks besteht, und im Kernel-Space läuft. Zum anderen eine nicht echtzeitfähigen Linux-Anwendung, die mit dem Echtzeit-Modul über FIFOs oder Shared-Memory kommuniziert (bei FIFOs: Systemaufrufe `write` und `read` über das Device `/dev/rtf*`)

5.1 Die einzelnen Komponenten des Steuerungssystems

Die System besteht aus mehreren Komponenten, welche auf verschiedene Dateien verteilt sind.

`sja1000.c:`
: enthält die Implementierung zur Anbindung des CAN-Controllers. Enthalten sind Funktionen zum Senden und Empfangen von CAN-Botschaften, sowie Funktionen zur Konfigurierung der Bus-Parameter.

`rt_candrv.c:`
: ist das eigentliche Echtzeit-Modul. Übernimmt die Initialisierung der Threads, sowie Erstellung von FIFOs, Handlern,... Auch der Code für die Echtzeit-Threads ist hier enthalten.

`grip_module.c:`
: Funktionen die direkt den Greifarm GRIP und den CAN-Joystick betreffen. Die Nachrichten die ankommen, werden hier entsprechend ihrem Identifier ausgewertet, und aktuelle Daten wie Drehzahl, Position werden gespeichert. Funktion zum Bereitstellung von Nachrichten für den Greifarm.

`grip.c:`
: Code für Anwendung im User-Space. Ist für die Kontrolle der Echtzeitanwendung verantwortlich. Es können Signale wie ”Start” und ”Stop” gesendet werden, die direkten Einfluß auf das Verhalten der Threads besitzen.

`commands.c:`
: Gehört zu `grip.c`. Enthält die Anweisungen die auf ein eingegebenes Kommando in GRIP ausgeführt werden.

`Makefile:`
: Das `Makefile` ist ein Steuerfile, das bei der Erstellung des Object-Code hilft. Diese Datei gibt an, wie das zu erzeugende Programm aus Einzelteilen zusammengebaut wird, und welche Abhängigkeiten dafür bestehen. Hier sind auch Punkte enthalten die dazu dienen das erstellte Modul auf das PC/104-System zu kopieren, ode eine Sicherungskopie der gesamten Anwendung zu erstellen.

Die Zuordnung dieser Komponenten in den strukturellen Aufbau Schaubild 5.2:

5.2 Architektur

Für das Steuerungssystem sind folgende Komponenten notwendig:

- RT-Threads
  Insgesamt werden 3 Threads benötigt. Jeweils einen Thread, der das Senden, bzw. das Empfangen übernimmt, und einen Thread der für die Sollwertgenerierung verantwortlich ist.

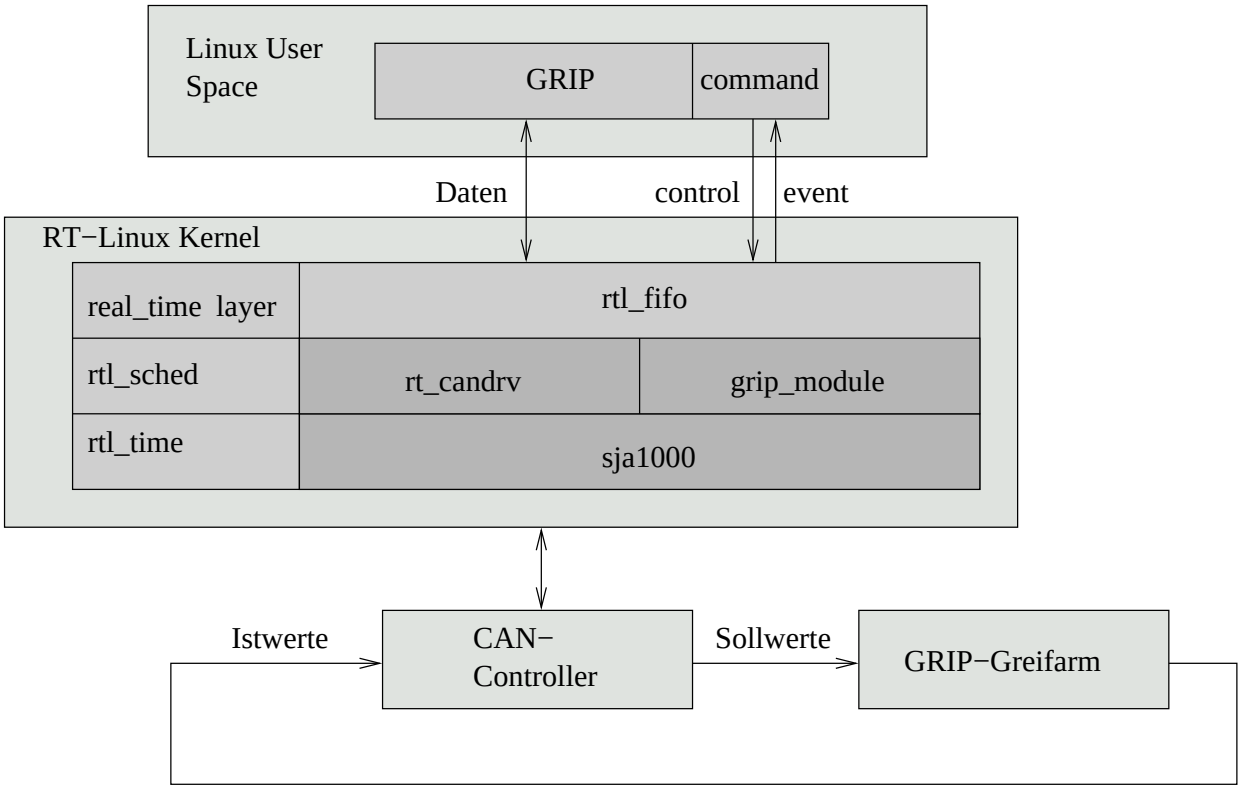

Abbildung 5.2: Architektur des Steuerungssystems (Version 1)

- Interrupt-Handler
  Wird eine Nachricht empfangen so löst der CAN-Controller einer Interrupt aus, der vom Interrupt-Handler empfangen wird und an die entsprende Funktion zum Lesen von Nachrichten aus dem Controller weitergeleitet wird.

- Message-Handler
  Message-Handler dienen dazu Threads die zurzeit nicht in Aktion sind, welche zum Beispiel auf den Beginn ihrer nächsten Ausfürungsperiode warten, über den Zustand eines FIFOs zu informieren. Wird zum Beispiel eine zu sendende Nachricht in den Ausgangs-FIFO gestellt, so informiert der Handler den Thread daß eine neue Übertragung ansteht. Dieser tritt dann sofort in Aktion und sendet die Botschaft.

- FIFOs
  Insgesamt werden 4 FIFOs benötig. Einen internen FIFO, der dazu dient eingehende Nachrichten zu sammeln und an den Interpolator weiterzuleiten. Ein Empfangsthread, der die selben Nachrichten enthält wie der interne FIFO; er kann vom User-Space aus gelesen werden. Einen Sende-FIFO, in welchen zu übertragende Nachrichten gestellt werden. Sowie einen Kontroll- und Nachrichten-FIFO. Diese FIFOs dienen zur Versändigung mit der User-Space Applikation. Es können Kommandos wie Start, Stop an den Kernel gegeben werden, und umgekehrt kann das Anwendungsprogrramm über aufgetretene Ereignisse informiert werden.

5.3 Initialisierung und kontinuierlicher Betrieb

Bevor ein kontinuierlicher Betrieb mit dem GRIP Roboter möglich ist, muß zuerst eine Initilalsierung vorgenommen werden. Dazu gehören:

- Senden eines NMT-Reset an alle Achsen (Funktion `reset_all_nodes`)
- Senden eines NMT-Start an alle Achsen (Funktion `start_all_nodes`)
- Parametrierung der einzelnen Achsen

Wurde der letzte Parameter gesendet, so kann der kontinuierliche Betrieb beginnen, indem eine `SYNC`-Nachricht gesendet wird. Der kontinuierliche Betrieb, hat dann immer folgenden Ablauf:

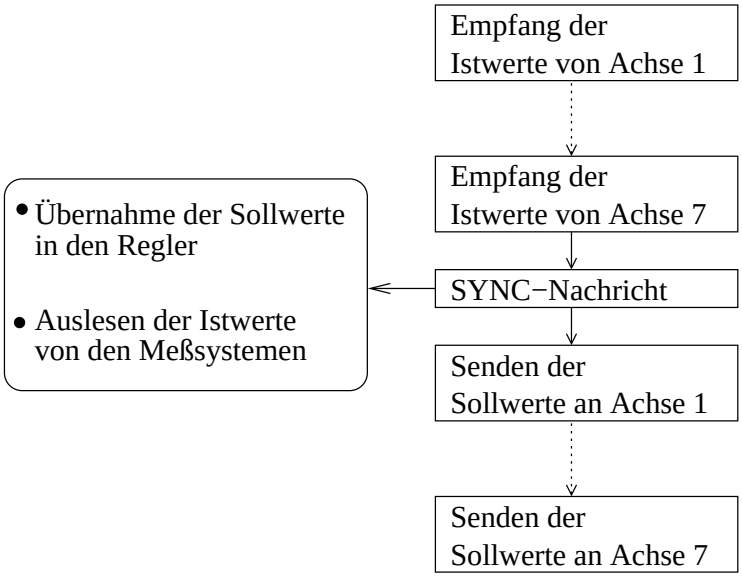

Abbildung 5.3: Kontinuierlicher Betrieb

5.4 Funktionsweise

Den Zusammenbau der einzelnen Komponenten zur Architektur des Steuerungssystems, zeigt die Abbildung 5.4.
Man erkennt die drei Threads die vom RT-Scheduler in periodischen Abständen aufgerufen werden. Der Thread `rt_receive_thread` dient dazu eingehende Nachrichten zu empfangen und auszuwerten. Dazu wird in regelmäßig die Funktion `ReadFromChip()` aufgerufen, die eingegangene Nachrichten aus dem FIFO des CAN-Controllers liest. Die Nachrichten werden dann in den Enpfangs-FIFO gestellt. Dies hat den Sinn, daß eine User-Space Applikation Zugriff auf die ankommenden Nachrichten hat, und sie gegebenenfalls auf dem Bildschirm ausgeben kann. Anschließend wird die Botschaft anhand des Identifiers dem entsprechenden Sender zugordnet, und dessen Istwerte werden aktualisiert.

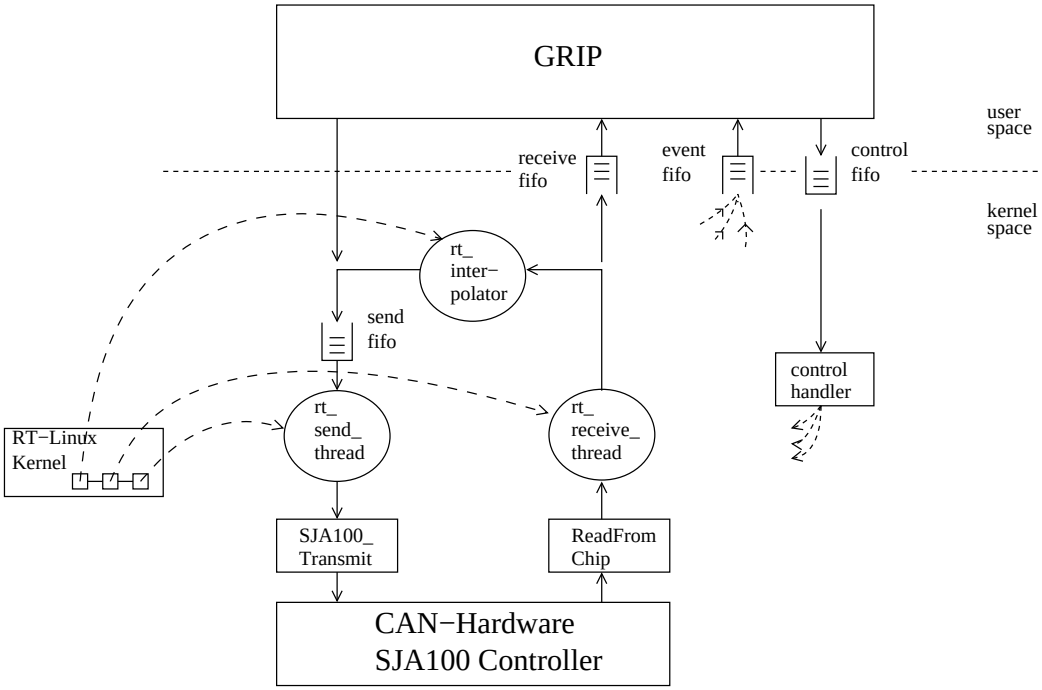

Abbildung 5.4: Architektur des Steuerungssystems (Version 2)

Der Thread `rt_send_thread` wird ebenfalls in festen Zeitabständen aktiviert, und verschickt zu sendende Botschaften aus dem Sende-FIFO, durch einen Aufruf der Funktion `SJA1000_Transmit()` auf den Bus.
Zwischen diesen beiden Threads vermittelt der `rt_interpolator` -Thread. Dieser berechnet aus den eingegangen Ist-Werten, neue Sollwerte für die einzelnen Achsen, und stellt sie zum Versenden in den Sende-FIFO.
Bei der hier sehr einfach durchgeführten Sollwertgenerierung, wird lediglich so vorgegangen, daß eintreffende Botschaften des CAN-Joysticks ausgewertet werden, und je nach Auslenkung des Joystick, an die entsprechend Achse eine Solldrehzahl vorgegeben wird. Es findet keine Interpolation im eigentlichen Sinne statt, daß Soll- und Istwerte miteinander verglichen werden, und aus der Regeldifferenz neue Führungsgrößen für die Achsen berechnet werden. Hier wurde lediglich eine einfache Art der Steuerung über eine Drehzahl-Vorgabe realisiert.
Das Programm GRIP generiert Signale und stellt sie in den `control_fifo`. Dieser wird vom Control_Handler ausgewertet, und leitet die Signale an die Threads weiter. Im Gegenzug können die Threads Signale an GRIP senden, indem sie entsprechende Nachrichten in den `event_fifo` stellen.

5.4.1 Aufruf des Echtzeit-Moduls

Der Echtzeitteil wird innerhalb eines Standard-Linux Moduls implementiert. Dieses Modul kann nach dem Kompilieren mit dem Befehl `insmod` zum Kernel hinzugefügt, und mittels `rmmod` wieder entfernt werden. Dieses Modul enthält eine Initialisierungs-, sowie eine entsprechende Funktion zur Deinstallation, und den Code für die zu erzeugenden Tasks. Wird der Befehl `insmod`

ausgeführt, so wird automatisch die Initialisierungsroutine `init_module` ausgeführt, deren Aufgabe es ist, alle erforderlichen Initialisierungsschritte zu übernehmen. Das heißt FIFOs anlegen, Tasks erzeugen, Parameter für Tasks definieren, Interrupthandler anlegen, usw. Die entsprechende Routine lautet beim entfernen des Moduls `cleanup_module`.
Der folgende Source-Code enthält diese beiden Routinen, für das Steuerungsystem.

```
int init_module(void)
{
        pthread_attr_t attr;
        struct sched_param sched_param;

        /* use default values, or the module parameters
         */
        BaseAdr = IO ? IO : BaseAdr;
        irq = IRQ ? IRQ : irq;
        baudrate = BR ? BR : baudrate;


        /* check if requested memory region can be used
         */
        if (check_region( BaseAdr, IO_RANGE)) {
                rtl_printf("rt_candrv.o: Unable to open port: 0x%lx\n",BaseAdr);
                return -ENODEV;
        }
        else {  /* get our resources */
                request_region( BaseAdr,IO_RANGE,"IXXAT CAN");
        }
        baseptr = ioremap(BaseAdr, IO_RANGE);
        /* initialize the C200 CAN-Card
         */
        if (CANInit((unsigned long)baseptr, baudrate))
        {
                rtl_printf("rt_candrv.o: Can't init CAN\n");
                return -1;
        }

        /* initialize the rt-fifos
         */
        if ( rtf_create(CONTROL_FIFO, CONTROL_FIFO_SIZE) < 0)
        {
          rtl_printf("rt_candrv.o: Could not install control fifo\n");
          return -ENODEV;
        }
        if ( rtf_create(EVENT_FIFO, EVENT_FIFO_SIZE) < 0)
        {
          rtl_printf("rt_candrv.o: Could not install event fifo\n");
          return -ENODEV;
        }
        if ( rtf_create(RECEIVE_FIFO, RECEIVE_FIFO_SIZE) < 0)
        {
          rtl_printf("rt_candrv.o: Could not install receive fifo\n");
          return -ENODEV;
```

```
}
if ( rtf_create(SEND_FIFO,SEND_FIFO_SIZE) < 0)
{
  rtl_printf("rt_candrv.o: Could not install send fifo\n");
  return -ENODEV;
}
if ( rtf_create(INT_RECEIVE_FIFO,INT_RECEIVE_FIFO_SIZE) < 0)
{
  rtl_printf("rt_candrv.o: Could not install internal receive fifo\n");
  return -ENODEV;
}

/* install the realtime interrupt handler
 */
if (irq != 0)
{

        if (rtl_request_irq(irq,intr_handler))
        {
                rtl_printf("rt_candrv.o:  can't get Interrupt %i\n",irq);
                        goto err;
                        return -1;
        }
        rtl_hard_enable_irq(irq);
        // rtl_printf("Interrupthandler enabled, IRQ=%i\n", irq);
}

/* initialize the message handler
 */
if ( rtf_create_handler(CONTROL_FIFO, grip_control_handler) )
{
  rtl_printf("rt_candrv.o: Could not install message handler\n");
  goto err;
  return -EINVAL;
}

/* initialize the send_fifo handler
 */
if (rtf_create_handler(SEND_FIFO, send_fifo_handler) )
{
  rtl_printf("rt_candrv.o: Could not install receive fifo handler\n");
  return -ENODEV;
}

//rtl_printf("candrv on CPU %d\n",rtl_getcpuid());

/* intialize the kernel threads
 */
pthread_attr_init (&attr);
pthread_attr_setcpu_np(&attr, 0);
sched_param.sched_priority = 1;
pthread_attr_setschedparam (&attr, &sched_param);
```

```
        if ( pthread_create (&threadsend,        &attr, rt_send_thread, (void *)1 ))
        {
          rtl_printf("rt_candrv.o: Failed to create rt_send_thread\n");
          return -EAGAIN;
        }
        if ( pthread_create (&threadrecv, &attr, rt_receive_thread, (void *)1 ))
        {
          rtl_printf("rt_candrv.o: failed to create rt_receive_thread\n");
          return -EAGAIN;
        }
        if ( pthread_create (&threadevalpos, &attr, rt_evalpos_thread, (void *)1 ))
        {
          rtl_printf("rt_candrv.o: failed to create rt_eval_pos_thread\n");
          return -EAGAIN;
        }

        rtl_printf("rt_candrv.o: Init module sucessfull\n");
        return 0;
}



void cleanup_module(void)
{
        /* delete threads
          */
        pthread_delete_np(threadevalpos);
        pthread_delete_np(threadsend);
        pthread_delete_np(threadrecv);

        /* removes the interrupthandler
          */
        if (irq != 0)
        {
                rtl_hard_disable_irq(irq);
                rtl_free_irq(irq);
        }

        /* destroy rt-fifos
          */
        rtf_destroy(SEND_FIFO);
        rtf_destroy(RECEIVE_FIFO);
        rtf_destroy(INT_RECEIVE_FIFO);
        rtf_destroy(EVENT_FIFO);
        rtf_destroy(CONTROL_FIFO);

        /* release allocated memory-region
          */
        release_region(BaseAdr,IO_RANGE);
        iounmap(baseptr);

        rtl_printf("rt_candrv.o: Removing module on CPU %d", rtl_getcpuid());
```

```
        rtl_printf(" sucessfull\n");
}
```

Die Initialisierung verläuft folgendermaßen:

- Wurden Werte zur Hardwarekonfiguration, als Parameter beim Aufruf des Moduls übergeben. Wenn ja, verwende diese. Wenn nein, verwende Default -Werte. Die Default-Werte sind:
  - Basisadresse = $0xD4000$
  - Interrupt = 6
  - Baudrate = $250K$
- Der zu belegende I/O-Bereich darf von keiner anderen Resource verwendet werden. Dies wird mit `check_region` geprüft. Ist der Bereich frei, so wird er mit `request_region` reserviert.
- Einblenden des Physikalischen Speichers des CAN-Controllers, in den virtuellen Adressraum von Linux. Anschließend kann mittels folgender Befehle auf die Register des Controllers zugegriffen werden.
  - `unsigned readb(address);`
  - `void writeb(unsigned value, address);`
  - `unsigned readw(address)`
  - `void writew(unsigned value, address);`
  - `unsigned readl(address)`
  - `void writel(unsigned value, address);`
- Initialisierung der CAN-Schnittstelle. Dies geschieht durch ein Aufruf der Funtkion `CANInit()`. Dabei wird ein Reset auf dem Baustein ausgelöst. Die Baudrate, sowie andere zur Initialisierung notwendige Parameter werden geschrieben.
- Erstellen der benötigten FIFOs mittels `rtf_create()`.
- Registrieren des Interrupt-Handlers. Dieser reagiert auf Interrupts des CAN-Controllers indem er die angegebene ISR ausführt.
- Registrieren der Message-Handler für den Sende- und Control-FIFO.
- Erzeugen der Threads.

Beim Entfernen des Moduls werden die erzeugten Threads wieder entfernt, die FIFOs gelöscht, und der belegte I/O-Bereich wird wieder freigegeben.

5.4.2 Beschreibung der Threads

Folgende drei Threads verwendet

- rt_receive_thread
- rt_send_thread
- rt_interpolator_thread

Alle Threads werden bei der Initialisierung angelegt, und bei Eintreffen eines Start-Signals aus dem Programm GRIP, periodisch aufgerufen. Dies geschieht mit dem Befehl

```
pthread_make_periodic(pthread p, hrtime_t start_time, hrtime_t
period)
```

Als Parameter wird der Name des Threads, der Startzeitpunkt der periodischen Ausführung, sowie die Periodendauer mit angegeben. Zeitangaben erfolgen in ns.
Der Thread `rt_receive_thread` dient dazu eingehende Nachrichten zu empfangen und auszuwerten. Dazu wird in regelmäßig die Funktion `ReadFromChip()` aufgerufen, die eingegangene Nachrichten aus dem FIFO des CAN-Controllers liest. Die Nachrichten werden ausgewertet und in den Empfangs-FIFO gestellt. Folgendes Fragment zeigt den wichtigsten Teil dieses Threads.

```
while(1) {
        while(0 != ReadFromChip((unsigned long)baseptr, &b[0])) {

                canmsg.id = (unsigned char)b[0];
                canmsg.id = (canmsg.id << 3) + ((unsigned char)b[1]>>5);
                canmsg.datalen = (b[1] & ((unsigned char)0x1f));
                memcpy(&canmsg.data[0], &b[2], 8);

                rtf_put(RECEIVE_FIFO, &canmsg, sizeof(canmsg));

                eval_can_message(&canmsg);
        }
        pthread_wait_np();
}
```

Der Thread `rt_send_thread` versendet CAN-Botschaften. Dies zeigt der folgende Ausschnitt:

```
while(1) {
        while(1) {
                ret = rtf_get(SEND_FIFO, &canmsg, sizeof(canmsg));

                if (ret != sizeof(canmsg))
                        break;

        while ((tries-- > 0) && (SJA1000_Transmit((unsigned long)baseptr,
                        canmsg.id, canmsg.datalen, &canmsg.data[0])))
              rtl_delay(1000);
```

```
            }
            pthread_wait_np();
    }
```

Der Interpolator-Thread befindet sich in einer Endlosschleife, und arbeitet nach folgendem Ablauf:

- Sync-Nachricht senden
- Sollwerte berechnen
- Sollwerte senden

5.4.3 Das Interface des CAN-Geräte Treibers

Der Geräte-Treiber unterstützt die Operation des CAN-Controllers im BasicCAN-Mode mit 11 Bit Identifier. Für die Verwendung von 29 Bit Identifieren nach CAN 2.0B muß in den PeliCAN-Mode geschaltet werden. In diesem Fall müssen alle verwendeten Routinen umgeschrieben werden, da eine andere Registerbelegung vorliegt.

CAN-Botschaften werden durch folgende Struktur definiert:

```
typedef struct {
        unsigned short id;
        unsigned short port;
        char           datalen;
        unsigned char  data[8];
} sCanMsg;
```

Dabei bedeutet

`id`	CAN-Identifier
`port`	gibt an auf welchem Kanal der CAN-Karte die Nachricht gesendet werden soll (hier immer 0)
`datalen`	Anzahl der Datenbytes (0 .. 8)
`data`	CAN-Nutzdaten

Folgende Funktionen sind in der Datei `sja1000.c` implementiert.

`SJA1000_Init():`
Ist die Routine die zur Initialisierung aufgerufen wird. Es werden die Werte für Bus-Timing, Acceptance Code und Acceptance Mask geschrieben. Weiterhin erfolgt die Konfiguration der Betriebsart im Control- sowie im Output Contol Register.

`SJA1000_Status():`
Liefert Status des CAN-Controllers

`SJA000_Reset():`
schaltet in den Reset-Mode

`SJA1000_Start():`
: schaltet in Operating-Mode

`SJA1000_Transmit():`
: sendet die übergebene Nachricht

`ReadFromChip():`
: liest Daten aus dem Empfangsregister. Die Daten weden als Struktur `sCanMsg` zurückgegeben.

`SetBaudrate():`
: dient zur Einstellung der Baudrate

`ClearOverrun():`
: Setzt das Data-Overrun Flag zurück

`CanSetMask():`
: Schreibt Acc-Code und Acc-Mask

`CanSetInterruptFlags():`
: zur Einstellung der Interrupt-Flags im Control-Register

5.4.4 GRIP-spezifische Funktionen

Zur Anbindung des Roboterarms sind folgende Funktionen vorhanden:

`eval_can_message():`
: Wird vom `rt_receive_thread` aufgerufen, und wertet die ankommenden Nachrichten entsprechend ihrem Identifiers aus.

`eval_axis_message():`
: Wird bei Ankunft einer Botschaft von einer der Achsen aufgerufen. Auswertung der Botschaft, und Aktualisierung der Ist-Werte.

`write_axis_pos():`
: Schickt für jede Achse Sollwerte an die Antriebe

`write_sync_msg():`
: Sendet eine Sync-Nachricht

`write_drive_param():`
: Sendet Antriebsparameter an die Achsen. Muss für jede Achse aufgerufen werden

`start_all_nodes():`
: Startet alle CAN-Knoten

`stop_all_nodes():`
: Stoppt alle CAN-Knoten

`reset_all_nodes()`:
Setzt alle CAN-Knoten zurück

`enable_node()`:
Setzt die Betriebsart einer einzelnen Achse

`disable_node()`:
Achse stillsetzten

`enable_all_nodes()`:
Setzt Betriebsart aller Achsen

`disable_all_nodes()`:
Alle Achsen stillsetzen

Um Joystick-Daten auszuwerten, werden folgende Funtkionen verwendet:

`eval_joystick_message()`:
Auswerten einer Joystick Nachricht

`write_jst_set_pdo_msg()`:
Aktiviert oder Deaktiviert die Nachrichten vom Joystick

`write_jst_set_mode_msg()`:
CAN-Joystick parametrieren

`write_jst_reset()`:
CAN-Joystick zurücksetzen

5.4.5 Steuerung und Kontrolle

Die Kontrolle des gesamten Steuerungssystems soll durch ein Programm im Linux User-Space realisiert werden. Es wurde das Programm GRIP implementiert, welches über zwei FIFOs mit dem Kernel-Modul `rt_candrv` kommunizieren kann.

- Der `control_fifo` dient dazu Signale an das Kernel-Modul zu übergeben.
- Der `event_fifo` arbeitet umgekehrt und übergibt Signale vom Kernel zum User-Space.

Um auf die Signale reagiern zu können, befindet sich im Kernel- sowie im User-Space ein Handler, der die FIFOs überwacht.
Die folgenden Signale sind implementiert:

- START: Die Steuerung wird gestartet, indem die Ausführung der Threads auf periodisch gestellt wird.
- STOP: Die Steuerung wird gestoppt, indem der Ausführungszeitpunkt auf unendlich, und die Periodendauer der Threads auf Null gesetzt wird.
- STATUS: Liefert den aktuellen Status des CAN-Controllers.

Kapitel 6

Zusammenfassung und Ausblick

Am Ende meiner Studienarbeit ist es möglich alle Achsen des GRIP-Greifarms mit Hilfe des CAN-Joysticks zu steuern.
Es wurde auf dem PC/104-System eine funktionsfähiges RT-Linux Plattform installiert, welche in der Lage ist ”harte” Echtzeitanforderungen zu erfüllen. Des weiteren wurde ein Echtzeit-Modul entwickelt, das in Verbindung mit einem CAN-Gerätetreiber die Kommunikation zwischen Greifarm und Steuerungssystem übernehmen kann. In dieses Modul wurde eine einfache Steuerungsfunktion implementiert mit der es möglich ist Joystickdaten auszulesen, und daraus Lagesollwerte zu generieren, welche an die einzelnen Achsen geschickt werden.
Dieses Steuerungsystem kann in späteren Entwicklungen um eine Transformation erweitert werden, die zwischen Raumkoordinaten und Roboterachskoordinaten umrechnet, und so bei alleiniger Vorgabe der Greiferposition eine simultane Bewegung aller Achsen ermöglicht.

Kapitel 7

Source-Code

7.1 sja1000

sja1000.h

```
#ifndef _SJA1000_H
#define _SJA1000_H
/*
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA       02111-1307, USA.*/

#ifndef bool
#define bool int
#endif


/*   Adress-Offset's der CAN-Register       */

#define   SJA1000_CTRL        0
#define   SJA1000_CMND        1
#define   SJA1000_STS         2
#define   SJA1000_INTER       3
#define   SJA1000_ACCC        4
#define   SJA1000_ACCM        5
#define   SJA1000_BT0         6
#define   SJA1000_BT1         7
#define   SJA1000_OUT_CTRL    8
```

```
#define   SJA1000_TEST        9
#define   SJA1000_TXDB1       10
#define   SJA1000_TXDB2       11
#define   SJA1000_TXDATA0     12
#define   SJA1000_RXDB1       20
#define   SJA1000_RXDB2       21
#define   SJA1000_RXDATA0     22
#define   SJA1000_RXDATA1     23
#define   SJA1000_RXDATA2     24
#define   SJA1000_RXDATA3     25
#define   SJA1000_RXDATA4     26
#define   SJA1000_RXDATA5     27
#define   SJA1000_RXDATA6     28
#define   SJA1000_RXDATA7     29
#define   SJA1000_DUMMY       30
#define   SJA1000_CLOCK_DIV   31


/*   Bedeutung der Bits im CAN-Control-Register      */

#define   CR_RR          0x01      /*   Reset Request    */
#define   CR_RIE         0x02      /*   Receive Interrupt Enable      */
#define   CR_TIE         0x04      /*   Transmit Interrupt Enable      */
#define   CR_EIE         0x08      /*   Error Interrupt Enable      */
#define   CR_OIE         0x10      /*   Overrun Interrupt Enable      */
#define   CR_S           0x40      /*   Sync   */

/*   Bedeutung der Bits im CAN-Command-Register   */

#define   CMR_TR         0x01      /*   Transmission Request      */
#define   CMR_AT         0x02      /*   Abort Transmission      */
#define   CMR_RRB        0x04      /*   Release Receive Buffer      */
#define   CMR_CDO        0x08      /*   Clear Data Overrun   */
#define   CMR_GTS        0x10      /*   Sleep- und Wake-up-Bit */

/*   Bedeutung der Bits im CAN-Status-Register       */
#define   SR_RBS         0x01      /*   Receive Buffer Status       */
#define   SR_DOS         0x02      /*   Data Overrun Status */
#define   SR_TBS         0x04      /*   Transmit Buffer Status       */
#define   SR_TCS         0x08      /*   Transmit Complete Status      */
#define   SR_RS          0x10      /*   Receive Status      */
#define   SR_TS          0x20      /*   Transmit Status      */
#define   SR_ES          0x40      /*   Error Status      */
#define   SR_BS          0x80      /*   Bus Status    */

/*   Bedeutung der Bits im CAN-Interrupt-Register        */
#define   IR_RI          0x01      /*   Receive Interrupt       */
#define   IR_TI          0x02      /*   Transmit Interrupt       */
#define   IR_EI          0x04      /*   Error Interrupt       */
#define   IR_DOI         0x08      /*   Data Overrun Interrupt      */
#define   IR_WUI         0x10      /*   Wake-Up Interrupt     */

/* DSCRRegister */
```

```
#define   DSCR2_RTR    0x10      /* Bitnummer des remoteflags im DSR2 */
#define   DSCR2_DLC    0x0f      /* Bereich des DLC */

#define   C2RTR_FLAG   0x10
#define   IO_RANGE     0xFF

/* Bus Timing Values */
#define BTR0_VALUE_5kBAUD 0x63
#define BTR1_VALUE_5kBAUD 0x1c

#define BTR0_VALUE_10kBAUD 0x31
#define BTR1_VALUE_10kBAUD 0x1c

#define BTR0_VALUE_20kBAUD 0x18
#define BTR1_VALUE_20kBAUD 0x1c

#define BTR0_VALUE_50kBAUD 0x09
#define BTR1_VALUE_50kBAUD 0x1c

#define BTR0_VALUE_100kBAUD 0x04
#define BTR1_VALUE_100kBAUD 0x1c

#define BTR0_VALUE_125kBAUD 0x03
#define BTR1_VALUE_125kBAUD 0x1c

#define BTR0_VALUE_250kBAUD 0x01
#define BTR1_VALUE_250kBAUD 0x1c

#define BTR0_VALUE_400kBAUD 0x01
#define BTR1_VALUE_400kBAUD 0x16

#define BTR0_VALUE_500kBAUD 0x00
#define BTR1_VALUE_500kBAUD 0x1c

#define BTR0_VALUE_800kBAUD 0x00
#define BTR1_VALUE_800kBAUD 0x16

#define BTR0_VALUE_1000kBAUD 0x40
#define BTR1_VALUE_1000kBAUD 0x23


/*   lokale Funktionen     */
int             CanInit(unsigned long BaseAdr, int BaudRate);
void            SJA1000_Status(unsigned long BaseAdr);
bool            SJA1000_Reset(unsigned long BaseAdr);
bool            SJA1000_Init(unsigned long BaseAdr);
bool            SJA1000_Start(unsigned long BaseAdr);
unsigned char   SJA1000_Transmit(unsigned long BaseAdr, int id,
                                 char dlc, const unsigned char   *p_src);
int             ReadFromChip(unsigned long BaseAdr, unsigned char  *b);

int             SetBaudrate(unsigned long BaseAdr, int BaudRate);
void            ClearOverrun(unsigned long BaseAdr);
```

```
void        CanSetMask(unsigned long BaseAdr,unsigned char ACR,
                       unsigned char AMR);
void        CanSetInterruptFlags(unsigned long BaseAdr,
                                 unsigned char interrupt_enable);

#endif
```

sja1000.c

```
/* This is a RTLinux-module for SJA1000-based CAN-cards.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA       02111-1307, USA.*/

#include <linux/ioport.h>
#include <asm/io.h>


#include "sja1000.h"



void SJA1000_Status(unsigned long BaseAdr)
{
        unsigned char cr;

        if (readb(BaseAdr+SJA1000_CTRL) & CR_RR) {
          rtl_printf("Reset - Mode\n");
        }
        else
          rtl_printf("Operating-Mode\n");

        cr=readb(BaseAdr + SJA1000_CTRL);
            /* Reset Controller */
        writeb(cr | 0x01, BaseAdr + SJA1000_CTRL);
        rtl_printf("SJA1000-Control-Register:%x\n",readb(BaseAdr+SJA1000_CTRL));
        rtl_printf("SJA1000-Status-Register: %x\n",readb(BaseAdr+SJA1000_STS));
        rtl_printf("SJA1000-Int-Control:   %x\n",readb(BaseAdr+SJA1000_INTER));
        rtl_printf("SJA1000-Acceptance Code: %x\n",readb(BaseAdr+SJA1000_ACCC));
        rtl_printf("SJA1000-Acceptance Mask: %x\n",readb(BaseAdr+SJA1000_ACCM));
        rtl_printf("SJA1000-Bus Timing 0:  %x\n",readb(BaseAdr+SJA1000_BT0));
        rtl_printf("SJA1000-Bus Timing 1:  %x\n",readb(BaseAdr+SJA1000_BT1));
        rtl_printf("SJA1000-Output-Control: %x\n",readb(BaseAdr+SJA1000_OUT_CTRL));
          /* Set the Controller back to normal operation mode; */
        writeb(cr, BaseAdr + SJA1000_CTRL);
}
```

```
bool   SJA1000_Reset(unsigned long BaseAdr)
{
        /*  Hardware - Reset */
       writeb(0x1, BaseAdr+ 0x100);
        /*  Reset-Zustand setzen     */
       writeb((readb(BaseAdr+SJA1000_CTRL) | CR_RR), BaseAdr + SJA1000_CTRL);
        /*  Gesetzter Wert zurueckliefern     */
       return(!(readb(BaseAdr+SJA1000_CTRL) & CR_RR));

}


bool   SJA1000_Init(unsigned long BaseAdr)
{
        int CTRL_init      = 0x1E;
        int ACCC_init      = 0x00;
        int ACCM_init      = 0xFF;
        int OUT_CTRL_init = 0x5E;

          /*  Flag's und Reset-Request im Ctrl-Reg. setzen      */
        writeb( (CTRL_init | CR_RR), BaseAdr + SJA1000_CTRL);

          /* Laesst sich der Wert nicht schreiben ?      */
        if((readb(BaseAdr+SJA1000_CTRL) & 0x1f) != ((CTRL_init & 0x1f) | CR_RR))
          /*  Error: CAN laesst sich nicht initialisieren      */
          return(1);

        writeb(ACCC_init,BaseAdr+SJA1000_ACCC);       /*  Acc-Code setzen   */
        writeb(ACCM_init,BaseAdr+SJA1000_ACCM);       /*  Acc-Mask setzen   */
        writeb(OUT_CTRL_init,BaseAdr+SJA1000_OUT_CTRL); /* Output-Ctrl setzen    */

  return(0);                                  /*  O.K.  */
}


bool   SJA1000_Start(unsigned long BaseAdr)
{
          /*  Runnig-Mode setzen   */
        writeb((readb(BaseAdr+SJA1000_CTRL) & ~CR_RR),BaseAdr+SJA1000_CTRL);
          /*  Gesetzter Wert zurueckliefern      */
        return ((readb(BaseAdr+SJA1000_CTRL) & CR_RR) != 0);
}


int CANInit(unsigned long BaseAdr, int BaudRate)
{
    int tries=100;

         /* CAN-Controller reset */
        SJA1000_Reset(BaseAdr);
```

```
        /* CAN-Controller initialisieren */
        if (SJA1000_Init(BaseAdr)) {
          rtl_printf("rt_candrv.o:  Fehler CAN-Controller Initialisierung\n");
          return 1;
        }

        /* Baudrate einstellen */
        if (SetBaudrate(BaseAdr, BaudRate) != 0) {
          rtl_printf("rt_candrv.o:  Fehler Initialisierung BaudRate");
          return 1;
        }

        /* Print Status Information */
        // SJA1000_Status(BaseAdr);

        /* CAN-Controller starten */
        while((tries--) && (SJA1000_Start(BaseAdr)))
          rtl_delay(20);
        if (tries == 0) {
          rtl_printf("rt_candrv.o:  Fehler CAN-Controller Start\n");
          return 1;
        }
    return 0;
}



void SJA1000_CanObjWr(unsigned long Adr, unsigned char *src, int cnt)
{

    int i;
    for(i=0; i< cnt; i++) {
      //        rtl_printf("%x ", src[i]);
        writeb(src[i], Adr+i);
    }
}

unsigned char  SJA1000_Transmit(unsigned long BaseAdr, int id, char dlc,
                                const unsigned char *p_src)
{
          /*  Ist CAN sendebereit ?  */
       if((readb(BaseAdr+SJA1000_STS) & (SR_BS | SR_TBS)) != SR_TBS) {
         //            rtl_printf("CAN nicht sendebereit\n");
           return(1);
        }

          /*  Objekt in CAN eintragen,  Descriptor zusammenbauen  */
        writeb((unsigned char )(id >> 3),BaseAdr+SJA1000_TXDB1);

        if ((dlc <= 8) && (dlc >= 0 )) {
          writeb((unsigned char )(((unsigned char )id << 5) | dlc),
                 BaseAdr+SJA1000_TXDB2);
                    /*  Daten unsigned char s kopieren  */
```

```
            SJA1000_CanObjWr(BaseAdr+SJA1000_TXDATA0,(unsigned char *)p_src,dlc);
        }
        else                /* als remote frame verschicken */
            writeb((unsigned char )(((unsigned char )id << 5) | DSCR2_RTR),
                    BaseAdr+SJA1000_TXDB2);
             /*  Sendeauftrag setzen     */
        writeb(CMR_TR,BaseAdr+SJA1000_CMND);
          //        rtl_printf("\nrt_candrv.o:  Objekt erfolgreich eingetragen\n");
           /*  Objekt erfolgreich eingetragen      */
        return(0);
}


void ReadRCVBuf(unsigned long BaseAdr, unsigned char *data)
{
        short i;
        /* sind immer 10 unsigned char s lang! */
        for (i=0; i<10;i++)
              *data++ = readb(BaseAdr + SJA1000_RXDB1 + i);
        /*  Receive-Buffer wieder freigeben      */
        writeb(CMR_RRB,BaseAdr+SJA1000_CMND);
}



int ReadFromChip(unsigned long BaseAdr, unsigned char   *b)
{
        int ret=0;
        int IRQState = readb(BaseAdr+SJA1000_INTER);

                /* es wurde ein unsigned char     empfangen */
        if (IRQState & IR_RI) {
                if(readb(BaseAdr+SJA1000_STS) & SR_RBS) {
                        ReadRCVBuf(BaseAdr, b);
                        ret = 1;
                }
        }
        if (IRQState & IR_DOI)
        if (readb(BaseAdr + SJA1000_STS) & SR_DOS) {
                    /*  Overrun Status loeschen! */
                writeb(CMR_CDO,BaseAdr + SJA1000_CMND);
        }
        if (IRQState & IR_EI) {
                rtl_printf("rt_candrv.o:  Error!!!\n");
                //                CANInit(BaseAdr, 250);
        }
        return ret;
}


int SetBaudrate(unsigned long BaseAdr, int BaudRate)
{
        unsigned char cr;
```

```
        cr=readb(BaseAdr + SJA1000_CTRL);
            /* Reset Controller */
        writeb(cr | 0x01, BaseAdr + SJA1000_CTRL);
            /* now set the baudrate: */
        switch (BaudRate)
        {
        case 10:
          writeb(BTR0_VALUE_10kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_10kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 20:
          writeb(BTR0_VALUE_20kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_20kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 50:
          writeb(BTR0_VALUE_50kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_50kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 100:
          writeb(BTR0_VALUE_100kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_100kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 125:
          writeb(BTR0_VALUE_125kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_125kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 250:
          writeb(BTR0_VALUE_250kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_250kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 500:
          writeb(BTR0_VALUE_500kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_500kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 800:
          writeb(BTR0_VALUE_800kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_800kBAUD, BaseAdr + SJA1000_BT1);
          break;
        case 1000:
          writeb(BTR0_VALUE_1000kBAUD, BaseAdr + SJA1000_BT0);
          writeb(BTR1_VALUE_1000kBAUD, BaseAdr + SJA1000_BT1);
          break;
        default :
          return -EINVAL;
          break;
        }
          // Set the Controller back to normal operation mode;
        writeb(cr, BaseAdr + SJA1000_CTRL);
    return 0;
}

void ClearOverrun(unsigned long BaseAdr)
```

```
{
        writeb(0x08, BaseAdr + SJA1000_CMND);
}


void CanSetMask(unsigned long BaseAdr,unsigned char ACR,unsigned char AMR)
{
  unsigned char cr;

        cr=readb(BaseAdr + SJA1000_CTRL);
          /* Reset Controller*/
        writeb(cr|0x01, BaseAdr + SJA1000_CTRL);
          // the 8 bost significant bits of the id of a can msg are
          // checked uing AMR and ACR bits.
          // AMR Mask the relevant bits: bit=0-> bit is relevant, bit=1-> don´t care
          // set AMR to 0xff, if you don't want to use hardware-filtering
        writeb(AMR, BaseAdr + SJA1000_ACCM);
          /* ACR relevant bits set or cleared */
        writeb(ACR, BaseAdr + SJA1000_ACCC);
          /* Set the Controller back to normal operation mode; */
        writeb(cr, BaseAdr + SJA1000_CTRL);

}

void CanSetInterruptFlags(unsigned long BaseAdr,unsigned char interrupt_enable)
{
        writeb(0x01, BaseAdr + SJA1000_CTRL); /* Reset Controller */
           /* Set the Controller back to normal */
           /* operation mode and set interrupts; */
        writeb(interrupt_enable & 0x1e, BaseAdr + SJA1000_CTRL);
}
```

7.2 rt_candrv

rt_candrv.h

```
#ifndef __RT_CANDRV_H_
#define __RT_CANDRV_H_
/* This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA        02111-1307, USA.*/
```

```
/*
 *      Strukture of CAN-Messages
 */
typedef struct {
        unsigned short id;
        unsigned short port;
        char           datalen;
        unsigned char  data[8];
} sCanMsg;


/*
 *     FIFOS used between GRIP and candrv.o
 */
#define SEND_FIFO              0
char*    SEND_FIFO_FILE=       "/dev/rtf0";
#define RECEIVE_FIFO           1
char*    RECEIVE_FIFO_FILE=    "/dev/rtf1";
#define INT_RECEIVE_FIFO       2
char*    INT_RECEIVE_FIFO_FILE= "/dev/rtf2";

#define CONTROL_FIFO           9     /* passing messages to real time module */
char*    CONTROL_FIFO_FILE=    "/dev/rtf9";
#define EVENT_FIFO             10    /* getting events from rt-module */
char*    EVENT_FIFO_FILE=      "/dev/rtf10";


#define CONTROL_FIFO_SIZE  1000   /* byte size for fifo buffer */
#define EVENT_FIFO_SIZE    1000
#define RECEIVE_FIFO_SIZE  1000
#define SEND_FIFO_SIZE     1000
#define INT_RECEIVE_FIFO_SIZE 1000

/* Interpolationstakt
 */
#define SENDPDO_TIME       1000*1000*10

/* Sende-, Empfangstakt
 */
#define SEND_TIME          1000*1000 /* every milli sec */
#define RECEIVE_TIME       1000*1000 /* every milli sec */

/*
 *     control messages passed to the real time module
 */
#define START_CONTROL      'a'
#define STOP_CONTROL       'b'
#define STATUS_CONTROL     'c'


/*
```

```
 *      event messages passed to user space
 */
#define INVALID_MESSAGE     'a'


//id grip_control_event_msg( unsigned char message );

int CANWrite(sCanMsg canmsg);

#endif //_RT_CANDRV_H_
```

rt_candrv.c

```
/*  This is a RT-Linux module, used to communicate
  with an IXXAT SJA1000 Can-Controller Card

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA       02111-1307, USA.*/

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <rtl.h>
#include <rtl_fifo.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_time.h>


/*      memory base address und interrupt line of the sja1000 card card
 */
static unsigned long BaseAdr      = 0xD4000;
static unsigned int irq           = 6;
static int baudrate               = 250;

/*      pointer to base address of Can–Controller, used by ioremap
 */
void   *baseptr;

/*      module parameters that can be given to the module with "insmod"
 */
static unsigned long IO = 0;
static int IRQ = 0;
static int BR = 0;
MODULE_PARM (IO, "l");
MODULE_PARM_DESC (IO, "base IO-Memory-Adress (default: 0xD4000)");
MODULE_PARM (IRQ, "i");
MODULE_PARM_DESC (IRQ, "number of Interrupt-line (default: 5)");
MODULE_PARM (BR, "i");
MODULE_PARM_DESC (BR, "Baudrate of CAN-Bus (default: 250)");
MODULE_AUTHOR ("Rainer Sigle, rainer.sigle@autip.de");
MODULE_DESCRIPTION ("This is a RT-Linux module, communicating with a passive
                  IXXAT PC/104 CAN-Card with SJA1000 Controller");
```

```
#include "sja1000.c"
#include "rt_candrv.h"
#include "grip_module.c"


 /*        Deklaration of the used threads
  */
pthread_t       threadsend,threadrecv,threadinterpol;



 /*        If RTL_DEBUG is defined every RT-thread prints information of
  *        incoming or outgoing messages
  */
//#define RTL_DEBUG



 /*         CANWrite puts messages in the SEND-FIFO
  */
int CANWrite(sCanMsg canmsg)
{
        int ret;

        ret = rtf_put(SEND_FIFO, &canmsg, sizeof(canmsg));
        pthread_wakeup_np(threadsend);
        return ret;
}
/*
int CANWrite(int port, int id, int len, unsigned char *data)
{
        sCanMsg canmsg;
        int ret;

        canmsg.id = id; canmsg.port = port; canmsg.datalen = len;
        if ((len<=8) && (len>0))
          memcpy(&canmsg.data[0], data, len);
        ret = rtf_put(SEND_FIFO, &canmsg, sizeof(canmsg));
        pthread_wakeup_np(threadsend);
        return ret;
}
*/

void grip_init()
{
        hrtime_t        t;
        int             sleep = 1000*1000*100;
        UNS08           LaufSDOKanal = 0;
        UNS08           LaufSDOParameter = 0;

        /* Initialisierung der zu benützenden Variablen,
```

```
 */
init_variables();

/* NMT-Nachrichten an alle Achsen
 */
reset_all_nodes();
start_all_nodes();

 /* Sync-Nachricht senden, warten
  */
write_sync_msg();
 t = gethrtime();
 while (gethrtime() < t + sleep);
StartSendSDO = TRUE;

 /*   Parametrierung der Antriebe
  */
while(1) {
if (StartSendSDO == TRUE) {
  if (LaufSDOKanal < CAN_NUMBEROFAXES) {
    if (Achse_vorhanden[LaufSDOKanal + 1] == TRUE) {
      if (LaufSDOParameter < PARAMETERANZAHL) {
        write_drive_param(LaufSDOKanal , LaufSDOParameter);
            t = gethrtime();
            while (gethrtime() < t + sleep);
          LaufSDOParameter = LaufSDOParameter + 1; }
      else {
          rtl_printf("Kernel: Status:  Achse %i ist aktiv\n", LaufSDOKanal + 1);
          LaufSDOKanal = LaufSDOKanal + 1;
          LaufSDOParameter = 0;  }
    }
    else {
        LaufSDOKanal = LaufSDOKanal + 1;
        LaufSDOParameter = 0;  }
  }
  else {
      LaufSDOKanal = 0;
      LaufSDOParameter = 0;
      StartSendSDO = FALSE;
      StartSendPDO = TRUE;
      break;
      }
   }
}

 /*   Joystick parametrieren
  */
 write_jst_set_pdo_msg(1);

 /*   Betriebsart der Achsen einstellen
  */
 enable_all_nodes(DREHZAHL);
```

```
        StartSendPDO = TRUE;
}


void grip_stop()
{
        disable_all_nodes();
        write_axis_pos();
        stop_all_nodes();
        write_jst_set_pdo_msg(0);
}



static void *rt_receive_thread(void *dummy)
{
        unsigned char       b[10];
        sCanMsg     canmsg;

#ifdef RTL_DEBUG
        char        Buffer[100];
        int         i;
        char        *ptr;
#endif
        pthread_wait_np();

        while(1) {
                if ((unsigned long)baseptr) {
                    while(0 != ReadFromChip((unsigned long)baseptr,&b[0])) {

                        canmsg.id = (unsigned char)b[0];
                        canmsg.id = (canmsg.id << 3) + ((unsigned char)b[1]>>5);

                        canmsg.datalen = (b[1] & ((unsigned char)0x1f));
                        memcpy(&canmsg.data[0], &b[2], 8);
                        rtf_put(RECEIVE_FIFO, &canmsg, sizeof(canmsg));

#ifdef RTL_DEBUG
                        ptr = &Buffer[0];
                        ptr += sprintf(Buffer,"%03x ",canmsg.id);
                        for (i=0;i<canmsg.datalen;i++)
                                ptr += sprintf(ptr," %02x",b[i+2]);
                        rtl_printf("recv: %s\n",Buffer);
#endif
                        eval_can_message(&canmsg);
                    }
                }
        pthread_wait_np();
        }
}
```

```
static void *rt_send_thread(void *dummy)
{
        sCanMsg   canmsg;
        int       tries;
        int       ret;

#ifdef RTL_DEBUG
        int       i;
#endif

        pthread_wait_np();
        while(1) {
                while(1) {
                        ret = rtf_get(SEND_FIFO, &canmsg, sizeof(canmsg));
                        if (ret != sizeof(canmsg))
                                break;

#ifdef RTL_DEBUG
                        rtl_printf("send: %03x", canmsg.id);
                        if ((canmsg.datalen < 0) || (canmsg.datalen > 8)) {
                                rtl_printf(" REMOTE\n");
                        }
                        else {
                                for(i=0;i<canmsg.datalen;i++)
                                        rtl_printf(" %02x",canmsg.data[i]);
                                rtl_printf("\n");
                        }
#endif
                tries = 100;
                while ((tries-- > 0) && (SJA1000_Transmit((unsigned long)baseptr,
                        canmsg.id, canmsg.datalen, &canmsg.data[0])))
                  pthread_wait_np();
                }
        pthread_wait_np();
        }
    return NULL;
}



static void *rt_interpolator_thread(void *dummy)
{
        int state1 = 0, state2 = 1, tries = 0, i=10, j=0;

        pthread_wait_np();
        /* Initialisierung und Parametrierung der Achsen
          */
        grip_init();

        /* Zyklischer Betrieb
          */
        if (StartSendPDO == TRUE)
             while(1) {
```

```
pthread_wait_np();
write_sync_msg();

if (state1) {
    if ((Joystick_x <= 0xFF) && (Joystick_flag == 0))
      SollDrehzahl[1] = Joystick_x;
    if ((Joystick_x <= 0xFF) && (Joystick_flag == 1))
      SollDrehzahl[1] = - Joystick_x;

    if ((Joystick_y <= 0xFF) && (Joystick_flag == 0))
      SollDrehzahl[2] = Joystick_y;
    if ((Joystick_y <= 0xFF) && (Joystick_flag == 2))
      SollDrehzahl[2] = - Joystick_y;

    if ((Joystick_z <= 0xFF) && (Joystick_flag == 0))
      SollDrehzahl[3] = Joystick_z;
    if ((Joystick_z <= 0xFF) && (Joystick_flag == 4))
    SollDrehzahl[3] = - Joystick_z;
  }

  if (state2) {
    if ((Joystick_x <= 0xFF) && (Joystick_flag == 0))
      SollDrehzahl[4] = Joystick_x;
    if ((Joystick_x <= 0xFF) && (Joystick_flag == 1))
      SollDrehzahl[4] = - Joystick_x;

    if ((Joystick_y <= 0xFF) && (Joystick_flag == 2))
      SollDrehzahl[5] = Joystick_y;
    if ((Joystick_y <= 0xFF) && (Joystick_flag == 0))
      SollDrehzahl[5] = - Joystick_y;

    if ((Joystick_z <= 0xFF) && (Joystick_flag == 4))
      SollDrehzahl[6] = Joystick_z;
    if ((Joystick_z <= 0xFF) && (Joystick_flag == 0))
      SollDrehzahl[6] = - Joystick_z;
  }

  if (Joystick_button == 2)
        SollDrehzahl[7] = 20;
  if (Joystick_button == 4)
        SollDrehzahl[7] = - 20;

  /* Abfrage zur Umschaltung Achsen 123 oder 456
   */
  if ((Joystick_button == 1) && (state1==1) && (tries++ >= 100)) {
        state1 = 0; state2 = 1; tries = 0;
  }
  if ((Joystick_button == 1) && (state2==1) && (tries++ >= 100)) {
        state1 = 1; state2 = 0; tries = 0;
  }

  /* Antriebsstatuswort in Ordnung ?
```

```
                    */
                    if (i-- == 0) {
                      i = 10000;
                      for (j = 0; j < CAN_NUMBEROFAXES; j++) {
                          if (Antriebsstatus[j] == 128) {
                              disable_node(j);
                              write_axis_pos();
                              enable_node(j, DREHZAHL);
                              write_axis_pos();
                              i = 1;
                          }
                      }
                    }
                    /* Sende Sollwerte
                     */
                    pthread_wait_np();
                    write_axis_pos();
                }
        return NULL;
}



unsigned int intr_handler(unsigned int irq ,struct pt_regs *p)
{
        unsigned char b[10];
        int packetArrived = 0;

        rtl_printf("rt_candrv: int arrived\n");
        if ((unsigned long)baseptr)
        {
                while(0 != ReadFromChip((unsigned long)baseptr,&b[0])) {
                        rtf_put(INT_RECEIVE_FIFO, b, 10);
                        packetArrived = 1;
                }
        }

        rtl_hard_enable_irq(irq);
        if (packetArrived) {
                pthread_wakeup_np (threadrecv);
        }
        return 0;
}


int send_fifo_handler(unsigned int fifo)
{
        if (fifo == SEND_FIFO) {
                pthread_wakeup_np (threadsend);
                return 0;
        }
        else
                return -EINVAL;
```

```
}


void write_event_msg(unsigned char message)
{
            /*
             *  Event message handling, send to event message fifo
             *  defined in rt_candrv.h.
             */
          rtf_put(EVENT_FIFO, &message, 1);
}


int grip_control_handler(unsigned int fifo)
{
          unsigned char message;

          while (rtf_get(CONTROL_FIFO, &message, 1) > 0) {
            switch(message) {

                /*  start cyclic interpolation by making thread periodic with
                 *  interpolation time in ns
                 */
              case START_CONTROL:
                   rtl_printf("rt_candrv.o: START Received\n");

                   pthread_make_periodic_np(threadsend, gethrtime(), SEND_TIME);
                   pthread_make_periodic_np(threadrecv, gethrtime(), RECEIVE_TIME);
                   pthread_make_periodic_np(threadinterpol, gethrtime(), SENDPDO_TIME);
                   break;

               /*  stops grip robot, by setting interpolation time to inifinity
                *  and period to 0
                */
              case STOP_CONTROL:
                   rtl_printf("rt_candrv.o: STOP Received\n");
                   grip_stop();

                   pthread_make_periodic_np(threadrecv, HRTIME_INFINITY, 0);
                   pthread_make_periodic_np(threadsend, HRTIME_INFINITY, 0);
                   pthread_make_periodic_np(threadinterpol, HRTIME_INFINITY, 0);
                   break;

              case STATUS_CONTROL:
                   rtl_printf("rt_candrv.o: STATUS Received\n");
                   SJA1000_Status(BaseAdr);
                   break;

              default:     write_event_msg(INVALID_MESSAGE);
              }
          }
     return 0;
}
```

```
int init_module(void)
{
        pthread_attr_t attr;
        struct sched_param sched_param;

        /* use either the default values, or the module parameters
         */
        BaseAdr = IO ? IO : BaseAdr;
        irq = IRQ ? IRQ : irq;
        baudrate = BR ? BR : baudrate;


        /* check if requested memory region can be used
         */
        if (check_region( BaseAdr, IO_RANGE)) {
                rtl_printf("rt_candrv.o: Unable to open port: 0x%lx\n",BaseAdr);
                return -ENODEV;
        }
        else {  /* get our resources */
                request_region( BaseAdr,IO_RANGE,"IXXAT CAN");
                /* rtl_printf("rt_candrv: Registered IO-memory: 0x%lx - 0x%lx\n",
                                                BaseAdr, BaseAdr + IO_RANGE - 1); */
        }
        baseptr = ioremap(BaseAdr, IO_RANGE);
        /* initialize the C200 CAN-Card
         */
        if (CANInit((unsigned long)baseptr, baudrate))
        {
                rtl_printf("rt_candrv.o: Can't init CAN\n");
                goto err;
                return -1;
        }

        /* initialize the rt-fifos
         */
        if ( rtf_create(CONTROL_FIFO, CONTROL_FIFO_SIZE) < 0) {
          rtl_printf("rt_candrv.o: Could not install control fifo\n");
          return -ENODEV;
        }
        if ( rtf_create(EVENT_FIFO, EVENT_FIFO_SIZE) < 0) {
          rtl_printf("rt_candrv.o: Could not install event fifo\n");
          return -ENODEV;
        }
        if ( rtf_create(RECEIVE_FIFO, RECEIVE_FIFO_SIZE) < 0) {
          rtl_printf("rt_candrv.o: Could not install receive fifo\n");
          return -ENODEV;
        }
        if ( rtf_create(SEND_FIFO,SEND_FIFO_SIZE) < 0) {
```

```
  rtl_printf("rt_candrv.o: Could not install send fifo\n");
  return -ENODEV;
}
if ( rtf_create(INT_RECEIVE_FIFO,INT_RECEIVE_FIFO_SIZE) < 0) {
  rtl_printf("rt_candrv.o: Could not install internal receive fifo\n");
  return -ENODEV;
}

/* install the realtime interrupt handler
 */
if (irq != 0) {
        if (rtl_request_irq(irq,intr_handler)) {
                rtl_printf("rt_candrv.o:  can't get Interrupt %i\n",irq);
                        goto err;
                        return -1;
        }
        rtl_hard_enable_irq(irq);
        // rtl_printf("Interrupthandler enabled, IRQ=%i\n", irq);
}

/* initialize the message handler
 */
if ( rtf_create_handler(CONTROL_FIFO, grip_control_handler) ) {
  rtl_printf("rt_candrv.o: Could not install message handler\n");
  goto err;
  return -EINVAL;
}

/* initialize the send_fifo handler
 */
if (rtf_create_handler(SEND_FIFO, send_fifo_handler) ) {
  rtl_printf("rt_candrv.o: Could not install receive fifo handler\n");
  goto err;
  return -ENODEV;
}

//rtl_printf("candrv on CPU %d\n",rtl_getcpuid());

/* intialize the kernel threads
 */
pthread_attr_init (&attr);
pthread_attr_setcpu_np(&attr, 0);
sched_param.sched_priority = 1;
pthread_attr_setschedparam (&attr, &sched_param);


if ( pthread_create (&threadsend,       &attr, rt_send_thread, (void *)1 )) {
  rtl_printf("rt_candrv.o: Failed to create rt_send_thread\n");
  goto err;
  return -EAGAIN;
}
if ( pthread_create (&threadrecv, &attr, rt_receive_thread, (void *)1 )) {
  rtl_printf("rt_candrv.o: failed to create rt_receive_thread\n");
```

```
            goto err;
            return -EAGAIN;
        }
        if ( pthread_create (&threadinterpol, &attr, rt_interpolator_thread, (void *)1 )) {
            rtl_printf("rt_candrv.o: failed to create rt_interpolator_thread\n");
            goto err;
            return -EAGAIN;
        }

        rtl_printf("rt_candrv.o: Init module sucessfull\n");
        return 0;

        err:
        if (irq != 0) {
                rtl_hard_disable_irq(irq);
                rtl_free_irq(irq);
        }
        release_region(BaseAdr,IO_RANGE);
        iounmap(baseptr);
        rtl_printf("rt_candrv.o: Error Routine ended\n");
        return -1;
}



void cleanup_module(void)
{
        /* Delete threads
         */
        pthread_delete_np(threadinterpol);
        pthread_delete_np(threadsend);
        pthread_delete_np(threadrecv);

        /* removes the interrupthandler
         */
        if (irq != 0) {
                rtl_hard_disable_irq(irq);
                rtl_free_irq(irq);
        }

        /* release rt-fifos
         */
        rtf_destroy(SEND_FIFO);
        rtf_destroy(RECEIVE_FIFO);
        rtf_destroy(INT_RECEIVE_FIFO);
        rtf_destroy(EVENT_FIFO);
        rtf_destroy(CONTROL_FIFO);

        /* releases allocated memory-region
         */
        release_region(BaseAdr,IO_RANGE);
        iounmap(baseptr);
```

```
        rtl_printf("rt_candrv.o: Removing module on CPU %d", rtl_getcpuid());
        rtl_printf(" sucessfull\n");

}
```

7.3 grip_module

grip_module.h

```
#ifndef _GRIP_MODULE_H
#define _GRIP_MODULE_H
/*
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA        02111-1307, USA.*/


/* SYNC - Telegramm - ID */
#define SYNCTXId              0x80

/* Netzwerkmanagment-Telegramm ID , Broadcast-Message */
#define NMTZeroMsgId          0x0

/* Empfangstelegramme fuer Antriebsparameter Achsen */
#define SDORXId               0x581
#define SDORXId_1   SDORXId + 1
#define SDORXId_2   SDORXId + 2
#define SDORXId_3   SDORXId + 3
#define SDORXId_4   SDORXId + 4
#define SDORXId_5   SDORXId + 5
#define SDORXId_6   SDORXId + 6
#define SDORXId_7   SDORXId + 7

/* Sendetelegramme fuer Antriebsparameter Achsen       */
#define SDOTXId               0x601
#define SDOTXId_1   SDOTXId + 1
#define SDOTXId_2   SDOTXId + 2
#define SDOTXId_3   SDOTXId + 3
#define SDOTXId_4   SDOTXId + 4
#define SDOTXId_5   SDOTXId + 5
#define SDOTXId_6   SDOTXId + 6
```

```
#define SDOTXId_7   SDOTXId + 7

/* Empfangstelegramme fuer Istwerte und Status der Achsen       */
#define PDORXId             0x181
#define PDORXId_1   PDORXId + 1
#define PDORXId_2   PDORXId + 2
#define PDORXId_3   PDORXId + 3
#define PDORXId_4   PDORXId + 4
#define PDORXId_5   PDORXId + 5
#define PDORXId_6   PDORXId + 6
#define PDORXId_7   PDORXId + 7

/* Sendetelegramme fuer Sollwerte und Steuerwort der Achsen */
#define PDOTXId             0x201
#define PDOTXId_1   PDOTXId + 1
#define PDOTXId_2   PDOTXId + 2
#define PDOTXId_3   PDOTXId + 3
#define PDOTXId_4   PDOTXId + 4
#define PDOTXId_5   PDOTXId + 5
#define PDOTXId_6   PDOTXId + 6
#define PDOTXId_7   PDOTXId + 7

/* Empfangstelegramme fuer Antriebsparameter */
#define EMCYRXId              0x81
#define EMCYRXId_1 EMCYRXId + 1
#define EMCYRXId_2 EMCYRXId + 2
#define EMCYRXId_3 EMCYRXId + 3
#define EMCYRXId_4 EMCYRXId + 4
#define EMCYRXId_5 EMCYRXId + 5
#define EMCYRXId_6 EMCYRXId + 6
#define EMCYRXId_7 EMCYRXId + 7

/* Betriebsmodi der CAN-Antriebe */
#define STILLSETZEN             0x0
#define DREHZAHL                0x9
#define POSITION                0xA
#define POSITION_VORSTEUER      0xB
#define DREHMOMENT              0xC
#define DREHZAHL_VORSTEUER      0xD

/* Steuerflags fuer CAN-Antrieb       */
#define CAN_ENABLE_IN           0x08
#define CAN_BREAK_RELEASE       0x10
#define CAN_ERROR_QUITT         0x20
#define CAN_POS_CALIBRATE       0x40
#define CAN_BOOTSTRAP           0x80

/* Statusflags fuer CAN-Antrieb       */
#define CAN_DRIVE_ERROR         0x80

/* erstes Byte eines SDO-Telegramms */
#define SDOSendByte             0x2f
```

```
/* Empfangstelegramm fuer aktuelle Daten des Joystick */
#define PDO_JST_DATA_RCV      0x400

/* Antwort des Joystick auf erfolgte Konfigurierung */
#define PDO_JST_KONFIG_RCV    0x52

/* Local-ID des Penny & Giles CAN Joysticks */
#define JOYSTICK_LOCAL_ID     0x00

/* default Host ID */
#define HOST_ID               0x51

#define BUTTON0               0x01
#define BUTTON1               0x08
#define BUTTON2               0x04

/* ANzahl Achsen des GRIP-Roboters */
#define CAN_NUMBEROFAXES       7
#define PARAMETERANZAHL       16

/* utilities*/
#define TRUE                   1
#define FALSE                  0


/* vereinfachte Typdefinitionen*/
#define BOOLEAN          unsigned  char
#define UNS08            unsigned  char
#define UNS16            unsigned  short
#define UNS32            unsigned  long
#define SGN08            signed  char
#define SGN16            signed  short
#define SGN32            signed  long
#define REAL32           float
#define REAL64           double



/* ======================Globale Variablen ==============================*/

enum _CAN_MSG_TYPE
{
  CAN_PDO, CAN_SDO, CAN_EMCY
};

/* Deklaration der Prozeßdatenobjekte */
  sCanMsg PDOTX_1;
  sCanMsg PDOTX_2;
  sCanMsg PDOTX_3;
  sCanMsg PDOTX_4;
  sCanMsg PDOTX_5;
  sCanMsg PDOTX_6;
  sCanMsg PDOTX_7;
```

```
   sCanMsg  PDOTX_8;

/* Deklaration der Servicedatenobjekte */
   sCanMsg  SDOTX_1;
   sCanMsg  SDOTX_2;
   sCanMsg  SDOTX_3;
   sCanMsg  SDOTX_4;
   sCanMsg  SDOTX_5;
   sCanMsg  SDOTX_6;
   sCanMsg  SDOTX_7;
   sCanMsg  SDOTX_8;

/* Sync-Message */
   sCanMsg  SyncMsg;

/* NMTMessage */
   sCanMsg  NMTZeroMsg;

/* Parametrierung Joystick */
   sCanMsg  JoystickMsg;

/* Istwerte */
   SGN16  IstPosition[CAN_NUMBEROFAXES];
   SGN16  IstDifPosition[CAN_NUMBEROFAXES];
   SGN16  IstDrehzahl[CAN_NUMBEROFAXES];
   SGN16  IstMoment[CAN_NUMBEROFAXES];
   UNS16  Antriebsstatus[CAN_NUMBEROFAXES];

/* Sollwerte */
   SGN16  SollPosition[CAN_NUMBEROFAXES];
   SGN16  SollDrehzahl[CAN_NUMBEROFAXES];
   SGN16  SollMoment[CAN_NUMBEROFAXES];

/* Variablen für Joystick */
   UNS08      Joystick_activated;
   UNS08      Joystick_button;
   UNS08      Joystick_flag;
   UNS16      Joystick_x;
   UNS16      Joystick_y;
   UNS16      Joystick_z;

/* Sonstiges */
   UNS08  Achse_vorhanden[CAN_NUMBEROFAXES];
   UNS08  SDOReady[CAN_NUMBEROFAXES];
   UNS08  StartSendSDO;
   UNS08  StartSendPDO;

//    UNS08 count=0;
/* Parameter */
 UNS08 Kv[CAN_NUMBEROFAXES] = {40, 50, 40, 40, 40, 40, 40};
 UNS08 Kp[CAN_NUMBEROFAXES] = {255, 200, 200, 100, 100, 100, 100};
 UNS08 Ki[CAN_NUMBEROFAXES] = {100, 100, 100, 50, 50, 50, 50};
 UNS08 DrehzahlGrenzwert[CAN_NUMBEROFAXES] = {255,255,255,255,255,255,255};
```

```
 UNS08  StromGrenzwert[CAN_NUMBEROFAXES] = {100,255,255,50,255,30,100};
 UNS08  Zeitkonstante_Filter[CAN_NUMBEROFAXES] = {200, 200, 200, 200, 200, 200, 200};
 UNS08  ReglerStruktur_Mn[CAN_NUMBEROFAXES] = {0, 0, 0, 0, 0, 0, 0};


/* ========================lokale Funktions-Prototypen======================*/

 /*       Allgemeine CAN-Treiberfunktioen
  */
BOOLEAN  eval_can_message(sCanMsg *canmsg);


 /*       Funktionen zur Anbindung der CAN-Antriebe
  */
void      eval_axis_message(UNS08 axis_id, sCanMsg *canmsg, UNS08 type);
void      write_axis_pos(void);

void      start_all_nodes(void);
void      stop_all_nodes(void);
void      reset_all_nodes(void);

void      enable_node(UNS08 node, UNS08 mode);
void      disable_node(UNS08 node);
void      enable_all_nodes(UNS08 mode);
void      disable_all_nodes(void);

void      write_sync_msg(void);
void      write_drive_param(UNS08 axis_id, UNS08 param_nr);
void      init_variables(void);
void      grip_init(void);
void      grip_stop(void);


 /*       Funktionen zur Anbindung des CAN-Joysticks
  */
void      eval_joystick_message (sCanMsg *canmsg);
void      write_jst_set_pdo_msg (UNS08 mode);
void      write_jst_set_mode_msg (UNS08 msg_rate);
void      write_jst_reset(void);


 /*       Hilfsfunktionen
  */
UNS16    TwoByteToWord( UNS08 ucByteLow, UNS08 ucByteHigh);
void      WordTo2Byte( UNS16 usWord, UNS08 *pucByteLow, UNS08 *pucByteHigh);

#endif   */ _GRIP_MODULE_H */
```

grip_module.c

```
/* These are the GRIP-specific functions for the RTLinux based module
    candrv.o

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA       02111-1307, USA.*/


#include "grip_module.h"


/* ***************************************************************************
 *
 * name : eval_can_message
 *
 * description : wertet die ankommenden CAN-Nachrichten entsprechend
 *                   ID aus
 *
 * ***************************************************************************/
BOOLEAN eval_can_message(sCanMsg *canmsg)
{
  switch(canmsg->id)
  {

/* Empfangstelegramme IST-Werte der Achsen */
    case  PDORXId_1:
      eval_axis_message(1, canmsg, CAN_PDO);
      break;
    case  PDORXId_2:
      eval_axis_message(2, canmsg, CAN_PDO);
      break;
    case  PDORXId_3:
      eval_axis_message(3, canmsg, CAN_PDO);
      break;
    case  PDORXId_4:
      eval_axis_message(4, canmsg, CAN_PDO);
      break;
    case  PDORXId_5:
      eval_axis_message(5, canmsg, CAN_PDO);
      break;
    case  PDORXId_6:
```

```
        eval_axis_message(6, canmsg, CAN_PDO);
        break;
      case PDORXId_7:
        eval_axis_message(7, canmsg, CAN_PDO);
        break;
/* Empfangstelegramme des Joystick */
      case PDO_JST_DATA_RCV:
        eval_joystick_message(canmsg);
        break;
      case PDO_JST_KONFIG_RCV:
        eval_joystick_message(canmsg);
        break;
      case HOST_ID: break;

/* Empfangstelegramme fuer Antriebsparameter */
      case SDORXId_1:
        eval_axis_message(1, canmsg, CAN_SDO );
        break;
      case SDORXId_2:
        eval_axis_message(2, canmsg, CAN_SDO );
        break;
      case SDORXId_3:
        eval_axis_message(3, canmsg, CAN_SDO );
        break;
      case SDORXId_4:
        eval_axis_message(4, canmsg, CAN_SDO );
        break;
      case SDORXId_5:
        eval_axis_message(5, canmsg, CAN_SDO );
        break;
      case SDORXId_6:
        eval_axis_message(6, canmsg, CAN_SDO );
        break;
      case SDORXId_7:
        eval_axis_message(7, canmsg, CAN_SDO );
        break;

/* Fehler-Telegramme */
      case EMCYRXId:
        break;
      case EMCYRXId_1:
        eval_axis_message(1, canmsg, CAN_EMCY );
        break;
      case EMCYRXId_2:
        eval_axis_message(2, canmsg, CAN_EMCY );
        break;
      case EMCYRXId_3:
        eval_axis_message(3, canmsg, CAN_EMCY );
        break;
      case EMCYRXId_4:
        eval_axis_message(4, canmsg, CAN_EMCY );
        break;
      case EMCYRXId_5:
```

```
        eval_axis_message(5, canmsg, CAN_EMCY );
        break;
      case EMCYRXId_6:
        eval_axis_message(6, canmsg, CAN_EMCY );
        break;
      case EMCYRXId_7:
        eval_axis_message(7, canmsg, CAN_EMCY );
        break;

      default:        // Unbekannte CAN Nachricht empfangen.
          rtl_printf("Unknown CAN-message received -> CAN-ID: %x\n", canmsg->id);
          return (FALSE);
          break;
  }

  return (TRUE);
} /* Ende: eval_can_message () */



/* ***************************************************************************
 *
 * name : eval_axis_message
 *
 * description : wertet die CAN-Message für jede Achse aus
 *
 * ************************************************************************** */
void eval_axis_message (UNS08 axis_id, sCanMsg *canmsg, UNS08 type)
{

  switch (type)
  {
    case CAN_PDO:

      IstPosition[axis_id]
      = (SGN16)TwoByteToWord( canmsg->data[0], canmsg->data[1] );

      IstDrehzahl[axis_id]
      = (SGN16)TwoByteToWord( canmsg->data[2], canmsg->data[3] );

      IstMoment[axis_id]
      = (SGN16)TwoByteToWord( canmsg->data[4], canmsg->data[5] );

      Antriebsstatus[axis_id] = canmsg->data[6];

      Achse_vorhanden[axis_id] = TRUE;
      /*
      msg_recv[axis_id] = TRUE;
    */


#ifdef RTL_DEBUG
```

```
        rtl_printf("Achse %i: Ist-Pos=%d, Ist-DZ=%d, Ist-Mom=%d, Stat=%d\n",
                   axis_id, IstPosition[axis_id], IstDrehzahl[axis_id],
                   IstMoment[axis_id], Antriebsstatus[axis_id]);
#endif

        break;

     case CAN_SDO:
#ifdef RTL_DEBUG
        rtl_printf("SDO-Object received ->ID=%x\n",canmsg->id);
#endif
        SDOReady[axis_id] = TRUE;

        break;

     case CAN_EMCY:
#ifdef RTL_DEBUG
        rtl_printf("Emergency-Object received -> ID = %x\n", canmsg->id);
#endif
        break;

     default:
#ifdef RTL_DEBUG
        rtl_printf("Unknown axis message received\n");
#endif
        break;
   }
} /* Ende: eval_axis_message () */




/* ***************************************************************************
 *
 * name : eval_joystick_message
 *
 * description :    wertet die CAN-Message fuer den Joystick    aus
 *
 * ************************************************************************** */
void eval_joystick_message(sCanMsg *canmsg)
{


          if (canmsg->data[0] != JOYSTICK_LOCAL_ID) {
            rtl_printf("Falsche Joystick LocalID (should never happen !)\n");
            return;
          }

          switch (canmsg->id)
          {
          case   PDO_JST_KONFIG_RCV:
            /* Antwort des Joystick auf erfolgte Konfigurierung */
```

```
            if (canmsg->data[1] == 0x80) { /* SETUP_CMD */
                switch (canmsg->data[2])
                {
                  case 0x10:   /* SET_MODE */
                    break;
                  case 0x12:   /* SET_PDO */
                    Joystick_activated = TRUE;
                    break;
                }
            }
            break;
          case  PDO_JST_DATA_RCV:
              /*  Joystick-Tasten auswerten      */
            //        rtl_printf("joystick-message received\n");

            Joystick_button = canmsg->data[2];
            Joystick_flag   = canmsg->data[3];
            Joystick_x      = canmsg->data[4];
            Joystick_y      = canmsg->data[5];
            Joystick_z      = canmsg->data[6];


#ifdef RTL_DEBUG
            rtl_printf("x=%x, y=%x, z=%x, flag=%i, Button=%i\n", Joystick_x,
              Joystick_y, Joystick_z, Joystick_flag, Joystick_button);
#endif
          }
} /* Ende: eval_joystick_message () */



/* ***************************************************************************
 *
 * name : write_axis_pos
 *
 * description :    schickt für jede Achse Sollwerte an die Antriebe
 *
 * ************************************************************************** */
void write_axis_pos ()
{
        WordTo2Byte(SollPosition[1], &(PDOTX_1.data[0]), &(PDOTX_1.data[1]));
        WordTo2Byte(SollPosition[2], &(PDOTX_2.data[0]), &(PDOTX_2.data[1]));
        WordTo2Byte(SollPosition[3], &(PDOTX_3.data[0]), &(PDOTX_3.data[1]));
        WordTo2Byte(SollPosition[4], &(PDOTX_4.data[0]), &(PDOTX_4.data[1]));
        WordTo2Byte(SollPosition[5], &(PDOTX_5.data[0]), &(PDOTX_5.data[1]));
        WordTo2Byte(SollPosition[6], &(PDOTX_6.data[0]), &(PDOTX_6.data[1]));
        WordTo2Byte(SollPosition[7], &(PDOTX_7.data[0]), &(PDOTX_7.data[1]));

        WordTo2Byte(SollDrehzahl[1], &(PDOTX_1.data[2]), &(PDOTX_1.data[3]));
        WordTo2Byte(SollDrehzahl[2], &(PDOTX_2.data[2]), &(PDOTX_2.data[3]));
        WordTo2Byte(SollDrehzahl[3], &(PDOTX_3.data[2]), &(PDOTX_3.data[3]));
        WordTo2Byte(SollDrehzahl[4], &(PDOTX_4.data[2]), &(PDOTX_4.data[3]));
        WordTo2Byte(SollDrehzahl[5], &(PDOTX_5.data[2]), &(PDOTX_5.data[3]));
```

```
        WordTo2Byte(SollDrehzahl[6], &(PDOTX_6.data[2]), &(PDOTX_6.data[3]));
        WordTo2Byte(SollDrehzahl[7], &(PDOTX_7.data[2]), &(PDOTX_7.data[3]));

        WordTo2Byte(SollMoment[1], &(PDOTX_1.data[4]), &(PDOTX_1.data[5]));
        WordTo2Byte(SollMoment[2], &(PDOTX_2.data[4]), &(PDOTX_2.data[5]));
        WordTo2Byte(SollMoment[3], &(PDOTX_3.data[4]), &(PDOTX_3.data[5]));
        WordTo2Byte(SollMoment[4], &(PDOTX_4.data[4]), &(PDOTX_4.data[5]));
        WordTo2Byte(SollMoment[5], &(PDOTX_5.data[4]), &(PDOTX_5.data[5]));
        WordTo2Byte(SollMoment[6], &(PDOTX_6.data[4]), &(PDOTX_6.data[5]));
        WordTo2Byte(SollMoment[7], &(PDOTX_7.data[4]), &(PDOTX_7.data[5]));

        if (Achse_vorhanden[1] == TRUE)
          CANWrite(PDOTX_1);
        if (Achse_vorhanden[2] == TRUE)
          CANWrite(PDOTX_2);
        if (Achse_vorhanden[2] == TRUE)
          CANWrite(PDOTX_3);
        if (Achse_vorhanden[2] == TRUE)
          CANWrite(PDOTX_4);
        if (Achse_vorhanden[2] == TRUE)
          CANWrite(PDOTX_5);
        if (Achse_vorhanden[2] == TRUE)
          CANWrite(PDOTX_6);
        if (Achse_vorhanden[2] == TRUE)
          CANWrite(PDOTX_7);

        return;
} /* Ende: write_can_axis_pos () */




/* ***************************************************************************
 *
 * name : write_jst_set_pdo_msg
 *
 * description :      mode = 0 deaktiviert die CAN-Nachrichten vom Joystick
 *                    mode = 1 aktiviert die CAN-Nachrichten vom Joystick
 *
 * *************************************************************************** */
void write_jst_set_pdo_msg(UNS08 mode)
{

        JoystickMsg.data[0]         = JOYSTICK_LOCAL_ID;
        JoystickMsg.data[1]         = 0x80;
        JoystickMsg.data[2]         = 0x12;
        JoystickMsg.data[3]         = mode;

        CANWrite(JoystickMsg);   CANWrite(JoystickMsg);

    return;
```

```
} /* Ende: write_jst_set_pdo_msg () */



/* ***************************************************************************
 *
 * name : write_jst_set_mode_msg
 *
 * description :     CAN Joystick parametrieren
 *
 * *************************************************************************** */
void write_jst_set_mode_msg (UNS08 msg_rate)
{

        JoystickMsg.id          = HOST_ID;
        JoystickMsg.datalen = 7;
        JoystickMsg.data[0] = JOYSTICK_LOCAL_ID;
        JoystickMsg.data[1] = 0x80;         /* SETUP_CMD                          */
        JoystickMsg.data[2] = 0x10;         /* SUB_CMD SET_MODE                   */
        JoystickMsg.data[3] = 0x00;         /* StartMode, 0=Manual, 1= Automatic  */
        JoystickMsg.data[4] = 0x00;         /* NodeGuard, 0=off, 1=10ms...255=2550ms */
        JoystickMsg.data[5] = msg_rate;     /* MsgRate, 0=10ms ... 9=100ms        */
        JoystickMsg.data[6] = 0x00;         /* MsgDelay, 0...9                    */

        CANWrite(JoystickMsg);

        return;

} /* Ende: write_jst_set_mode_msg () */



/* ***************************************************************************
 *
 * name : write_jst_reset
 *
 * description :     CAN-Joystick zuruecksetzen
 *
 * *************************************************************************** */
void write_jst_reset()
{

        JoystickMsg.datalen             = 3;
        JoystickMsg.data[0]             = JOYSTICK_LOCAL_ID;
        JoystickMsg.data[1]             = 0x50;     /* NET_CMD */
        JoystickMsg.data[2]             = 0x01;     /* SUB_CMD NET_MODE */

        CANWrite(JoystickMsg);

        return;
} /* Ende: write_jst_reset () */
```

```
/* ************************************************************************
 *
 * name : write_sync_msg
 *
 * description :     verschickt SYNC-Nachrichten an den CAN-Treiber
 *
 * ********************************************************************** */
void write_sync_msg()
{

    CANWrite(SyncMsg);

    return;
} /* Ende: write_can_sync_msg () */



/* ************************************************************************
 *
 * name : write_drive_param
 *
 * description :     sendet CAN-Antriebsparameter
 *                   wird immer pro Achse aufgerufen
 *
 * ********************************************************************** */
void write_drive_param(UNS08 axis_id, UNS08 param_nr)
{
        UNS08 Parameterliste[PARAMETERANZAHL];
        UNS16 id;

        switch(param_nr) {
        case (0): Parameterliste[0] = DrehzahlGrenzwert[axis_id]; break;
        case (1): Parameterliste[1] = StromGrenzwert[axis_id]; break;
        case (2): Parameterliste[2] = Kv[axis_id]; break;
        case (3): Parameterliste[3] = 0; break;
        case (4): Parameterliste[4] = Kp[axis_id]; break;
        case (5): Parameterliste[5] = Ki[axis_id]; break;
        case (6): Parameterliste[6] = Zeitkonstante_Filter[axis_id]; break;
        case (7): Parameterliste[7] = ReglerStruktur_Mn[axis_id]; break;
        default:    Parameterliste[param_nr] = 0; break;
        }


        id = SDORXId;
        switch(axis_id) {
        case (0): SDOTX_1.data[0] = SDOSendByte;
                  memcpy(&SDOTX_1.data[1], &id, 2);
                  SDOTX_1.data[3] = param_nr;
                  SDOTX_1.data[4] = Parameterliste[param_nr];
                  CANWrite(SDOTX_1);
                  break;
```

```
            case (1): SDOTX_2.data[0] = SDOSendByte;
                      memcpy(&SDOTX_2.data[1], &id, 2);
                      SDOTX_2.data[3] = param_nr;
                      SDOTX_2.data[4] = Parameterliste[param_nr];
                      CANWrite(SDOTX_2);
                      break;
            case (2): SDOTX_3.data[0] = SDOSendByte;
                      memcpy(&SDOTX_3.data[1], &id, 2);
                      SDOTX_3.data[3] = param_nr;
                      SDOTX_3.data[4] = Parameterliste[param_nr];
                      CANWrite(SDOTX_3);
                      break;
            case (3): SDOTX_4.data[0] = SDOSendByte;
                      memcpy(&SDOTX_4.data[1], &id, 2);
                      SDOTX_4.data[3] = param_nr;
                      SDOTX_4.data[4] = Parameterliste[param_nr];
                      CANWrite(SDOTX_4);
                      break;
            case (4): SDOTX_5.data[0] = SDOSendByte;
                      memcpy(&SDOTX_5.data[1], &id, 2);
                      SDOTX_5.data[3] = param_nr;
                      SDOTX_5.data[4] = Parameterliste[param_nr];
                      CANWrite(SDOTX_5);
                      break;
            case (5): SDOTX_6.data[0] = SDOSendByte;
                      memcpy(&SDOTX_6.data[1], &id, 2);
                      SDOTX_6.data[3] = param_nr;
                      SDOTX_6.data[4] = Parameterliste[param_nr];
                      CANWrite(SDOTX_6);
                      break;
            case (6): SDOTX_7.data[0] = SDOSendByte;
                      memcpy(&SDOTX_7.data[1], &id, 2);
                      SDOTX_7.data[3] = param_nr;
                      SDOTX_7.data[4] = Parameterliste[param_nr];
                      CANWrite(SDOTX_7);
                      break;
            }
    return;

} /* Ende: write_can_drive_param () */



/* ***************************************************************************
 *
 * name : reset_all_nodes
 *
 * description : setzt alle Can-Knoten zurück
 *
 * ************************************************************************** */
void reset_all_nodes()
{
```

```
        NMTZeroMsg.data[0] = 129;
        NMTZeroMsg.data[1] =   0;

        CANWrite(NMTZeroMsg);
} /* Ende: reset_all_nodes() */



/* ************************************************************************
 *
 * name : start_all_nodes
 *
 * description : startet alle Can-Knoten
 *
 * ************************************************************************ */
void start_all_nodes()
{
        NMTZeroMsg.data[0] =   1;
        NMTZeroMsg.data[1] =   0;

        CANWrite(NMTZeroMsg);
} /* Ende: start_all_nodes() */



/* ************************************************************************
 *
 * name : stop_all_nodes
 *
 * description : stoppt alle Can-Knoten
 *
 * ************************************************************************ */
void stop_all_nodes()
{
        NMTZeroMsg.data[0] =   2;
        NMTZeroMsg.data[1] =   0;

        CANWrite(NMTZeroMsg);
} /* Ende: stop_all_nodes() */


/* ************************************************************************
 *
 * name : enable_node
 *
 * description :
 *
 * ************************************************************************ */
void enable_node(UNS08 node, UNS08 mode)
{
        switch(node)
        {
            case (1): PDOTX_1.data[6] = mode; break;
```

```
            case (2): PDOTX_2.data[6] = mode; break;
            case (3): PDOTX_3.data[6] = mode; break;
            case (4): PDOTX_4.data[6] = mode; break;
            case (5): PDOTX_5.data[6] = mode; break;
            case (6): PDOTX_6.data[6] = mode; break;
            case (7): PDOTX_7.data[6] = mode; break;
        }
}

/* ***************************************************************************
 *
 * name : disable_node
 *
 * description :
 *
 * *************************************************************************** */
void disable_node(UNS08 node)
{
        switch(node)
       {
            case (1): PDOTX_1.data[6] = STILLSETZEN; break;
            case (2): PDOTX_2.data[6] = STILLSETZEN; break;
            case (3): PDOTX_3.data[6] = STILLSETZEN; break;
            case (4): PDOTX_4.data[6] = STILLSETZEN; break;
            case (5): PDOTX_5.data[6] = STILLSETZEN; break;
            case (6): PDOTX_6.data[6] = STILLSETZEN; break;
            case (7): PDOTX_7.data[6] = STILLSETZEN; break;
            case (8): PDOTX_8.data[6] = STILLSETZEN; break;
        }
}


/* ***************************************************************************
 *
 * name : enable_all_nodes
 *
 * description :
 *
 * *************************************************************************** */
void enable_all_nodes(UNS08 mode)
{
        PDOTX_1.data[6] = mode;
        PDOTX_2.data[6] = mode;
        PDOTX_3.data[6] = mode;
        PDOTX_4.data[6] = mode;
        PDOTX_5.data[6] = mode;
        PDOTX_6.data[6] = mode;
        PDOTX_7.data[6] = mode;
        PDOTX_8.data[6] = mode;
}

/* ***************************************************************************
 *
```

```
 * name : disable_all_nodes
 *
 * description :
 *
 * ************************************************************************ */
void disable_all_nodes()
{
        PDOTX_1.data[6] = STILLSETZEN;
        PDOTX_2.data[6] = STILLSETZEN;
        PDOTX_3.data[6] = STILLSETZEN;
        PDOTX_4.data[6] = STILLSETZEN;
        PDOTX_5.data[6] = STILLSETZEN;
        PDOTX_6.data[6] = STILLSETZEN;
        PDOTX_7.data[6] = STILLSETZEN;
        PDOTX_8.data[6] = STILLSETZEN;
}



/* ************************************************************************
 *
 * name : TwoByteToWord
 *
 * description :    Composes a Word value consisting of the given Highbyte
 *                  and Lowbyte. Returns the composed Word.
 *
 * ************************************************************************ */
UNS16 TwoByteToWord( UNS08 ucByteLow, UNS08 ucByteHigh)
{
        UNS16 usWord;
        UNS08 *puc;
        puc = (UNS08*)&usWord;

        *puc = ucByteLow;

        puc++;
        *puc = ucByteHigh;
        return usWord;
}



/* ************************************************************************
 *
 * name : WordTo2Byte
 *
 * description : Dispatches the bytes of a short value to 2 single bytes
 *
 * ************************************************************************ */
void WordTo2Byte( UNS16 usWord, UNS08 *pucByteLow, UNS08 *pucByteHigh )
{
        UNS08* puc;
        puc = (UNS08*)&usWord;
```

```
        *pucByteLow = *puc;

        puc++;
        *pucByteHigh = *puc;
        return;
}



/* ***************************************************************************
 *
 * name : init_variables
 *
 * description : make all the initialization nessecary to start
 *                    communication: reset arrays, initalizes variables, ...
 *
 * ************************************************************************** */
void init_variables()
{
  UNS08 i;
  /*
   * reset Arrays
   */
        for (i=0; i < CAN_NUMBEROFAXES; i++)
        {
          SollPosition[i] = 0;
          SollDrehzahl[i] = 0;
          SollMoment[i] = 0;
          IstPosition[i] = 0;
          IstDifPosition[i] = 0;
          IstDrehzahl[i] = 0;
          IstMoment[i] = 0;
          Antriebsstatus[i] = 0;

          Achse_vorhanden[i] = FALSE;
          SDOReady[i] = FALSE;
          StartSendSDO = FALSE;
          StartSendPDO = FALSE;
          Joystick_activated = FALSE;
        }

        PDOTX_1.id = PDOTXId_1;
        PDOTX_2.id = PDOTXId_2;
        PDOTX_3.id = PDOTXId_3;
        PDOTX_4.id = PDOTXId_4;
        PDOTX_5.id = PDOTXId_5;
        PDOTX_6.id = PDOTXId_6;
        PDOTX_7.id = PDOTXId_7;

        SDOTX_1.id = SDOTXId_1;
        SDOTX_2.id = SDOTXId_2;
        SDOTX_3.id = SDOTXId_3;
        SDOTX_4.id = SDOTXId_4;
```

```
SDOTX_5.id = SDOTXId_5;
SDOTX_6.id = SDOTXId_6;
SDOTX_7.id = SDOTXId_7;

PDOTX_1.datalen = 8;
PDOTX_2.datalen = 8;
PDOTX_3.datalen = 8;
PDOTX_4.datalen = 8;
PDOTX_5.datalen = 8;
PDOTX_6.datalen = 8;
PDOTX_7.datalen = 8;

SDOTX_1.datalen = 8;
SDOTX_2.datalen = 8;
SDOTX_3.datalen = 8;
SDOTX_4.datalen = 8;
SDOTX_5.datalen = 8;
SDOTX_6.datalen = 8;
SDOTX_7.datalen = 8;

PDOTX_1.port = 0;
PDOTX_2.port = 0;
PDOTX_3.port = 0;
PDOTX_4.port = 0;
PDOTX_5.port = 0;
PDOTX_6.port = 0;
PDOTX_7.port = 0;

SDOTX_1.port = 0;
SDOTX_2.port = 0;
SDOTX_3.port = 0;
SDOTX_4.port = 0;
SDOTX_5.port = 0;
SDOTX_6.port = 0;
SDOTX_7.port = 0;

for (i=0; i < 8; i++)
{
  PDOTX_1.data[i] = 0;
  PDOTX_2.data[i] = 0;
  PDOTX_3.data[i] = 0;
  PDOTX_4.data[i] = 0;
  PDOTX_5.data[i] = 0;
  PDOTX_6.data[i] = 0;
  PDOTX_7.data[i] = 0;
}

for (i=0; i < 8; i++)
{
  SDOTX_1.data[i] = 0;
  SDOTX_2.data[i] = 0;
  SDOTX_3.data[i] = 0;
  SDOTX_4.data[i] = 0;
```

```
            SDOTX_5.data[i] = 0;
            SDOTX_6.data[i] = 0;
            SDOTX_7.data[i] = 0;
          }

          /* Sync-Nachricht und NMT-Nachricht definieren */
          SyncMsg.id = SYNCTXId;
          SyncMsg.port = 0;
          SyncMsg.datalen = 0;

          NMTZeroMsg.id = NMTZeroMsgId;
          NMTZeroMsg.port = 0;
          NMTZeroMsg.datalen = 2;

          JoystickMsg.id = HOST_ID;
          JoystickMsg.port = 0;
          JoystickMsg.datalen = 4;

          for (i=0; i < 8; i++) {
            SyncMsg.data[i] = 0;
            NMTZeroMsg.data[i] = 0;
            JoystickMsg.data[i] = 0;
          }
          StartSendSDO = FALSE;
          StartSendPDO = FALSE;

    return;
}
```

7.4 grip

grip.h

```
#ifndef _GRIP_H
#define _GRIP_H
/*
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA       02111-1307, USA.*/
```

```
struct  cmd
{
        char    *c_name;                /* command name */
        char    *c_help;                /* help message */
        int     (*c_handler)();         /* routine to do the work */
};

#endif /* GRIP_H */
```

grip.c

```
/* This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA        02111-1307, USA.*/

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>
#include <unistd.h>
#include <pthread.h>

#include "grip.h"
#include "commands.c"


int         fromatty;
char        cmdline[20];
int         margc;
char        *margv[20];
struct      cmd *getcmd();
struct      cmd cmdtab[];

pthread_t       event_fifo_id, receive_fifo_id;

/*
 * compares the given command with those in cmdtab
 */
struct cmd *getcmd(register char *name)
{
        register char *p, *q;
        register struct cmd *c, *found;
        register int nmatches, longest;

        longest = 0;
        nmatches = 0;
        found = 0;
        for (c = cmdtab; (p = c->c_name); c++) {
                for (q = name; *q == *p++; q++)
                        if (*q == 0)                /* exact match? */
                                return(c);
                if (!*q) {                          /* the name was a prefix */
```

```
                        if (q - name > longest) {
                                longest = q - name;
                                nmatches = 1;
                                found = c;
                        } else if (q - name == longest)
                                nmatches++;
                }
        }
        if (nmatches > 1)
                return((struct cmd *)-1);
        return(found);
}


/*
 * Slice a string up into argc/argv.
 */
void makeargv()
{
        register char *cp;
        register char **argp = margv;
        int n=0;

        margc = 0;
        for (cp = cmdline; *cp && (cp - cmdline) < sizeof(cmdline) && n < 20; n++) {
                while (isspace(*cp))
                        cp++;
                if (*cp == '\0')
                        break;
                *argp++ = cp;
                margc += 1;
                while (*cp != '\0' && !isspace(*cp))
                        cp++;
                if (*cp == '\0')
                        break;
                *cp++ = '\0';
        }
        *argp++ = 0;
}


/*
 *      Command parser.
 */
void cmdscanner()
{
        register struct cmd *c;

        for (;;) {
                if (fromatty) {
                        printf("grip> ");
                        fflush(stdout);
                }
```

```
                if (fgets(cmdline, sizeof(cmdline), stdin) == 0)
                        quit();
                if (cmdline[0] == 0 || cmdline[0] == '\n')
                        break;
                makeargv();
                c = getcmd(margv[0]);
                if (c == (struct cmd *)-1) {
                        printf("Ambiguous command\n");
                        continue;
                }
                if (c == 0) {
                        printf("Invalid command\n");
                        continue;
                }
                (*c->c_handler)(margc, margv);
        }
}


#define HELPINDENT (sizeof ("directory"))
/*
 * Help command.
 */
int help(int argc, char *argv[])
{
        register struct cmd *c;

        if (argc == 1) {
                register int i, j, w;
                int columns, width = 0, lines;
                extern int NCMDS;

                printf("Commands are:\n\n");
                for (c = cmdtab; c->c_name; c++) {
                        int len = strlen(c->c_name);

                        if (len > width)
                                width = len;
                }
                width = (width + 8) &~ 7;
                columns = 80 / width;
                if (columns == 0)
                        columns = 1;
                lines = (NCMDS + columns - 1) / columns;
                for (i = 0; i < lines; i++) {
                        for (j = 0; j < columns; j++) {
                                c = cmdtab + j * lines + i;
                                if (c->c_name)
                                        printf("%s", c->c_name);
                                if (c + lines >= &cmdtab[NCMDS]) {
                                        printf("\n");
                                        break;
                                }
```

```
                                w = strlen(c->c_name);
                                while (w < width) {
                                        w = (w + 8) &~ 7;
                                        putchar('\t');
                                }
                        }
                }
                return 0;
        }
        while (--argc > 0) {
                register char *arg;
                arg = *++argv;
                c = getcmd(arg);
                if (c == (struct cmd *)-1)
                        printf("?Ambiguous help command %s\n", arg);
                else if (c == (struct cmd *)0)
                        printf("?Invalid help command %s\n", arg);
                else
                        printf("%-*s\t%s\n", HELPINDENT,
                                c->c_name, c->c_help);
        }
        return 0;
}


/*
 *  MAIN-Routine of program GRIP
 *  locks for command-line arguments, intializes the command parser
 *  creates fifos and tasks
 */
int main(int argc, char *argv[])
{
        register struct cmd *c;
        char *name;

        name = argv[0];

        if (--argc > 0) {
                c = getcmd(*++argv);
                if (c == (struct cmd *)-1) {
                        printf("?Ambiguous command\n");
                        exit(1);
                }
                if (c == 0) {
                        printf("?Invalid command\n");
                        exit(1);
                }
                (*c->c_handler)(argc, argv);
                exit(0);
        }
        fromatty = isatty(fileno(stdin));
        if (init_fifos() == 0) {
          if (pthread_create(&event_fifo_id, NULL, event_fifo_task, (void*)0 )) {
```

```
            printf("Can't create event task\n");
            return -1;
          }
          if (pthread_create(&receive_fifo_id, NULL, receive_fifo_task, (void*)0 )) {
            printf("Can't create receive task\n");
            return -1;
          }

          for (;;)
            {
              cmdscanner();
            }
        }
        else
          return -1;
    return 0;
}
```

7.5 commands

/ This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.*

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA./*

```
#include "rt_candrv.h"


/*
 * grip – command tables
 */
int     start(), stop(),    quit(), help(), status();

char    starthelp[]   = "start the realtime threads by making thread periodic";
char    stophelp[]    = "stops grip robot by setting execution
                         time to infinity and period to 0";
char    quithelp[]    = "exit grip";
char    helphelp[]    = "gets help on commands";
char    statushelp[] =  "show status of grip robot";

struct cmd cmdtab[] = {
```

```
        { "start",      starthelp,      start },
        { "stop",       stophelp,       stop },
        { "help",       helphelp,       help },
        { "quit",       quithelp,       quit },
        { "status",     statushelp,     status },
        { "?",          helphelp,       help },
        { 0 },
};

int     NCMDS = sizeof (cmdtab) / sizeof (cmdtab[0]);


/*
 *      Initialise variables for fifos
 */
int receive_fifo, send_fifo, control_fifo, event_fifo;

/*
 *      initialize fifos from grip.c
 */
static int init_fifos( void )
{
        /* Initialise the receive fifo
         */
        receive_fifo = open(RECEIVE_FIFO_FILE, O_WRONLY );
        if (receive_fifo      < 0 ) {
            printf( "Error opening %s Receive Fifo\n", RECEIVE_FIFO_FILE );
            printf( "Maybe the module isn't loaded\n");
            return -1;
        }
        /* Initialise the send fifo
         */
        send_fifo = open(SEND_FIFO_FILE, O_WRONLY );
        if (send_fifo     < 0 ) {
            printf( "Error opening %s Send Fifo\n", SEND_FIFO_FILE );
            printf( "Maybe the module isn't loaded\n");
            return -1;
        }
        /* Initialise the control message fifo to candrv
         */
        control_fifo = open(CONTROL_FIFO_FILE, O_WRONLY );
        if (control_fifo      < 0 ) {
            printf( "Error opening %s Control Fifo\n", CONTROL_FIFO_FILE );
            printf( "Maybe the module isn't loaded\n");
            return -1;
        }
        /* Initialise the event message fifo from candrv
         */
        event_fifo = open(EVENT_FIFO_FILE, O_RDONLY );
        if (event_fifo      < 0 ) {
          printf( "Error opening %s Event Fifo\n", EVENT_FIFO_FILE );
          printf( "Maybe the module isn't loaded\n");
          return -1;
```

```
        }
    return 0;
}

/*
 *    cleanup fifos from rt module
 */
static int cleanup_fifos( void )
{
        if (close(receive_fifo)) {
          printf("Error closing Receive-Fifo\n");
          return -1;
        }
        if (close(send_fifo)) {
          printf("Error closing Send-Fifo\n");
          return -1;
        }
        if (close(control_fifo)) {
          printf("Error closing Control-Fifo\n");
          return -1;
        }
        if (close(event_fifo)) {
          printf("Error closing Event-Fifo\n");
          return -1;
        }
        return 0;
}


/*
 *    Sends control messages
 */
inline void write_control_message( unsigned char message )
{
        write(control_fifo, &message, 1 );
}

/*
 *    receive fifo task, evaluates messages passed from
 *    the real time module.
 */
void *receive_fifo_task()
{
        sCanMsg canmsg;
        int     i;

        while (1) {
          if (read(receive_fifo, &canmsg, sizeof(canmsg)) == sizeof(canmsg)) {
            printf("recv: %03x", canmsg.id);
            if ((canmsg.datalen < 0) || (canmsg.datalen > 8))
              printf(" REMOTE\n");
            else {
              for (i=0; i < canmsg.datalen; i++)
```

```
                printf(" %02x", canmsg.data[i]);
              printf("\n");
            }
          }
        }
}
/*
 *      event fifo task, evaluates messages passed from
 *      the real time module.
 */
void *event_fifo_task()
{
        unsigned char message;

        while (1) {
          if (read(event_fifo, &message, 1) == 1) {
            switch(message)
            {
              case INVALID_MESSAGE:
                printf("Message %c not valid\n",message);
                break;

              default:
                printf("Message %c not possible\n",message);
            }
          }
        }
}


int start(void)
{
        printf("start\n");
        write_control_message(START_CONTROL);
        return 0;
}


int stop(void)
{
        printf("stop\n");
        write_control_message(STOP_CONTROL);
        return 0;
}


int status(void)
{
        printf("status\n");
        write_control_message(STATUS_CONTROL);
        return 0;
}
```

```
/*
 * Exit grip
 */
int quit(void)
{
        if (cleanup_fifos() == 0) {
            printf("bye!\n");
            exit(0);
        }
        else
           exit(-1);
}
```

7.6 Makefile

```
MYCFLAGS=-g -Wall -pipe -D_POSIX_THREADS -D_POSIX_THREAD_SAFE_FUNCTIONS \
        -D_REENTRANT -Wno-uninitialized -lpthread

TARGET=/mnt/fisw34

include rtl.mk

all: rt_candrv.o candrv.o grip

grip: grip.c
        $(CC) ${INCLUDE} ${MYCFLAGS} -o grip grip.c

rt_candrv.o: rt_candrv.c sja1000.c sja1000.h rt_candrv.h grip_module.c grip_module.h
        $(CC) ${INCLUDE} ${CFLAGS} -c rt_candrv.c

candrv.o: candrv.c sja1000.c sja1000.h rt_candrv.h
        $(CC) ${INCLUDE} ${CFLAGS} -c candrv.c

cantest: cantest.c
        $(CC) ${INCLUDE} ${MYCFLAGS} -o cantest cantest.c

clean:
        rm -f *.o grip cantest

dist:
        tar zcvf grip.tgz Makefile *.c *.h rtl.mk

copy:
        cp -dpR rt_candrv.o ${TARGET}/root
```

Literaturverzeichnis

[1] HiTEX Automation. *HiCO 486 Handbuch*.

[2] HiTEX Automation. *HiCOETS2-DOC Handbuch*.

[3] Ulrich Laible & Gottfried Bauer. *Einsatz von CAN zur Steuerung eines Roboterarms*, Juni 2001.

[4] Konrad Etschberger. *Controller-Area-Network*. Carl Hanser Verlag München Wien, zweite edition, 2000.

[5] FSMLabs. *RTLinux Installation Instructions*, 2001.

[6] Penny & Giles. *Joystickelektronik PGJC6LK*, 1999.

[7] IXXAT Automation GmbH. *PC-I 04/104-Handbuch*.

[8] Robert Bosch GmbH. *CAN Specification Version 2.0*, 1991.

[9] Wittenstein Motion Control GmbH. *Technische Dokumentation Antriebsbaureihe TPS und SLS*, 2000.

[10] Ulrich Laible. *Dokumentation für das Projekt GRIP*, 2001.

[11] M-Systems. *IM-DOC-21 Installation Manual: Using the DiskOnChip© with Linux OS*, 1999.

[12] M-Systems. *DiskOnChip© TrueFFS© driver for Linux*, 2001.

[13] Uri Pomerantz. *Linux Kernel Module Programming Guide*, 1999.

[14] G. Pritschow. *Steuerungstechnik 2*, ws 01/02 edition. Skript zur Vorlesung.

[15] Allesandro Rubini. *Linux Device Drivers*. O'Reilly & Associates Inc., second edition, 2001.

[16] Philips Semiconductors. *Determination of the Bit Timing Parameters for the CAN Controller SJA1000*, 1997. Application Note AN97046.

[17] Philips Semiconductors. *Data Sheet SJA1000 Stand-alone CAN controller*, 1999.

[18] Andrew Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[19] Beck Böhme Dziadska Kunitz Magnus Schröter Verworner. *Linux-Kernel-Programmierung Algorithmen und Strukturen der Version 2.2*. Addison-Wesley, fünfte edition, 1999.

[20] Peter Wurmsdobler. *A Sample Control Application employing Real Time Linux*.

[21] Victor Yodaiken. *The RTLinux Manifesto*, 2000.

[22] Victor Yodaiken and Michael Barabanov. *RTLinux Version 2*, 1999.

Abbildungsverzeichnis

Tabellenverzeichnis