Mikrocontroller Programmierung und Application Programming Interface zur Steuerung von CAN-Geräten mit EPICS

Diplomarbeit

Version1.1

von

Linda Vanina Fouedjio

Hochschule Darmstadt

Fachbereich Elektrotechnik/Informationstechnik –

UNIVERSITY OF APPLIED SCIENCES

Referat:	Prof. Dr. Klaus Schaefer
Korreferat:	Prof. Dr. Antje Wirth
Betreuer GSI:	Dr. Michael Traxler und Dr. Peter Zumbruch

Abgegeben am: _____ _____

Datum Unterschrift

Erklärung

Ich versichere hiermit, dass ich meine Diplomarbeit mit dem Thema

„Mikrocontroller Programmierung und Application Programming Interface zur Steuerung von CAN-Geräten mit EPICS“ verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Darmstadt, 29.04.09)

(Unterschrift)

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, mit deren Hilfe und Unterstützung diese Diplomarbeit zustande gekommen ist.

Bei Herrn Prof. Dr. Klaus Schaefer und Frau Prof. Dr. Antje Wirth der Hochschule Darmstadt bedanke ich mich für die Unterstützung und Betreuung im Rahmen dieser Arbeit.

Besonderer Dank gilt meinen Betreuern, den Herren Dr. Holger Brand, Dr. Michael Traxler und Dr. Peter Zumbruch bei der Firma GSI–Helmholtzzentrum für Schwerionenforschung, nicht nur für die interessante Themenstellung, sondern auch für ihre engagierte und lehrreiche Betreuung, die oftmals über den Inhalt der Arbeit hinausging.

Weiterer Dank gebührt meinem Freund und meinen engen Freunden für ihre moralische

Unterstützung. Zuletzt geht mein Dank an meine Eltern und Geschwister, die mir von sehr fern immer viel Mut geben.

Kurzfassung

Bei dem Detektorsystem High Acceptance DiElectron Spectrometer (HADES), das im Rahmen einer internationalen Kollaboration beim Helmholtzzentrum für **S**chwerionenforschung(GSI) in Darmstadt entstand, handelt es sich um ein hochauflösendes Dileptonen-Spektometer der zweiten Generation. Wesentliche Aufgabe von HADES ist die Untersuchung der Änderung der Eigenschaften von Hadronen (Baryonen und Mesonen) über deren Zerfall in Elektronen und Positronen in heißer und dichter Kernmaterie. Die Kenntnis dieser Eigenschaften unter diesen extremen Bedingungen ist Voraussetzung zum Verständnis z. B. der Vorgänge in Neutronensternen.

Bis zum Erscheinen dieser Arbeit ist die Überwachung der Teilsysteme des HADES Detektorsystems mit einem Control Area Network(CAN)-Interface mit Hilfe eines alten Modells Versa Modular Eurocard(VME) realisiert. Die Zielsetzung dieser Arbeit ist, das auf VME basierte System durch ein an der GSI entwickeltes HADControl Board zu ersetzen. Das HADControl besteht im Wesentlichen aus einem System on Chip (SoC) und einem Mikrocontroller mit verschiedenen Schnittstellen. In dieser Arbeit geht es um die Implementierung der Ansteuerung des CAN Interfaces mittels eines Application Programming Interfaces(API) im Mikrocontroller.

Inhaltsverzeichnis

Seite

Abkürzungsverzeichnis

Abkz	Erläuterung der Abkürzung
API	Application Programming Interface
AVR	Advanced Virtual RISC
CA	Channel Access
CR	Collision Resolution
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DLC	Data Length Code
CSMACD	Carrier Sense Multiple Access Collision Detection
EEPROM	Electrically Erasable Programmable Read only Memory
EPICS	Experimental Physics and Industrial Control System
ESR	Experimentierspeicherring
ETRAX	Ethernet Token Ring Achse
FOPI	4π, Name eines Experiments bei der GSI
HADES	High Acceptance DiElectron Spectrometer
IDE	Identifier Extension
IFS	Intermission Frame Space
IOC	Input Output Control
ISR	Interrupt Service Routine
I2C	Inter Integrated Circuit
JTAG	Joint Test Action Group
LAN	Local Area Network
MEDM	Motif Editor and Display Manager
Mob	Message Object Block
NRZ	Non Return to Zero
PC	Personal Computer
RAM	Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
SIS	Schwerionensynchrotron
SoC	System on Chip
UART	Universal Asynchronous Receiver Transmitter
UNILAC	Universal Linear Accelerator
VME	Versa Module Eurocard

1 Einleitung

1.1 GSI Helmholtzzentrum für Schwerionenforschung

Das GSI–Helmholtzzentrum für Schwerionenforschung wurde als Gesellschaft für Schwerionenforschung (GSI) 1969 gegründet und hat als Gesellschafter die Bundesrepublik Deutschland und das Land Hessen. Das Forschungsinstitut hat ca.1050 Mitarbeiter davon 300 Wissenschaftler und Ingenieure. Darüber hinaus hat die GSI 1000 Gastwissenschaftler aus dem In- und Ausland pro Jahr. Die Schwerpunkte der GSI liegen in den Forschungsgebieten Kernphysik, Atomphysik, Biophysik, Kernchemie, Plasmaphysik und Materialforschung. Dazu betreibt die GSI eine weltweit einmalige Beschleunigungsanlage für Ionenstrahlen. Diese ermöglicht es den Forschern, immer wieder neue und faszinierende Entdeckungen in der Grundlagenforschung zu machen. Die Beschleunigeranlage besteht aus Ionenquellen, einem Universellen Linear Beschleuniger (UNILAC), einem Schwerionensynchrotron (SIS), einem Experimentierspeicherring (ESR) und anderen Teilanlagen. Die beschleunigten Ionen werden benötigt, um Experimente an den Experimentierplätzen durchzuführen. Die Experimentierplätze sind zum Beispiel FOPI, HADES, LAND und Therapie. Das implementierte API soll in dem HADES Experiment benutzt werden. HADES ist einer der größten Experimentierplätze der GSI. Die Abbildung 1.1 zeigt die Beschleunigeranlage der GSI mit den Experimentplätzen.

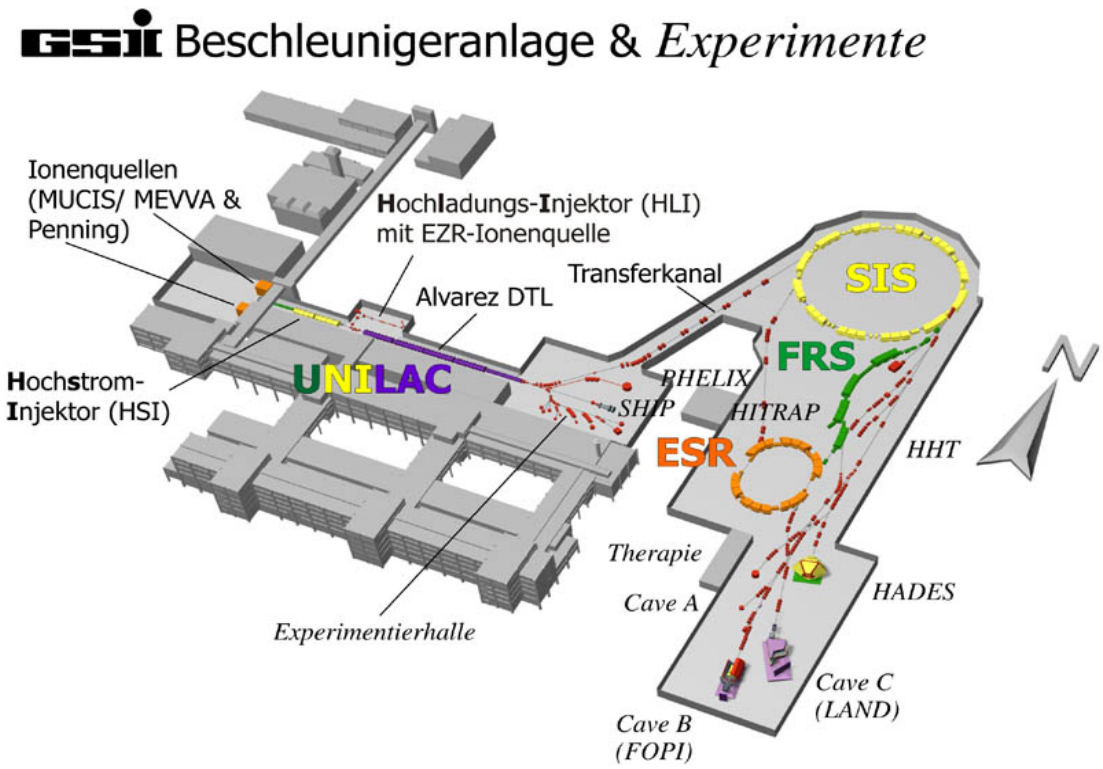

Abbildung 1:1 Darstellung der Beschleunigeranlage der GSI mit den Testplätzen [1]

1.2 Motivation und Problemstellung

Beim Betrieb des HADES-Experiments entsteht starke ionische Strahlung. Die ionisierende Strahlung verbietet den Zutritt zum Experimentbereich, d.h. es gibt keine Überwachung auf der Detektorkomponenten und anderen Geräten.

Um während des Experimentbetriebs verschiedene Parameter des Detektors unter anderen Spannungen, Strömen, Temperaturen, sowie die Steuerung und die Regelung der Geräte zu kontrollieren, wird eine Fernsteuerung benutzt.

Viele kontrollierte Geräte haben ein CAN Interface. Die Steuerung von CAN-Geräten soll in Verbindung mit TCP/IP basierten Kontrollsystemen verwirklicht werden, deshalb ist eine Schnittstelle zwischen CAN und TCP/IP für die Steuerung und Kontrolle von solchen Geräten notwendig. Die Idee ist, dieses Vorhaben mit einem kleinen und preiswerten HADControl-Board zu verwirklichen. Für die Kommunikation und Ansteuerung bietet dieses Board sowohl ein CAN Interface als auch eine TCP/IP Interface an.

Aufbauend auf diesem API lässt sich dann vergleichsweise einfach ein Treibermodul für die an der GSI genutzte Kontrollsystemsoftware Experimental Physics and Industrial Control System (EPICS) erstellen, wodurch ein Zugriff auf CAN-Geräte via Internet möglich ist.

Das HADControl besteht aus einem Mikrocontroller mit integriertem CAN Controller und einem SoC(CPU), wobei das SoC via TCP/IP angesprochen werden kann. Dabei erfolgt die Kommunikation der beiden Einheiten über eine interne serielle Schnittstelle. Die Kommunikation zwischen EPICS und HADControl-Board bzw. SoC soll über TCP/IP erfolgen.

1.3 Ziel der Arbeit

Die Zielsetzung dieser Arbeit ist die Implementierung einer Software-Routine auf dem Mikrocontroller, die die Ansteuerung eines CAN Interfaces mittels eines Application Programming Interfaces (API) ermöglicht. Dabei soll die Central Processing Unit des SoC via serieller Schnittstelle auf CAN-Geräte zugreifen können.

Dafür ist zuerst die Erstellung eines Kommunikationsprotokolls für das HADControl-Board zwischen den o.g. Einheiten notwendig. Auf Basis dieses Kommunikationsprotokolls wird ein API implementiert. Das implementierte API soll über diese interne serielle Schnittstelle zwischen Mikrocontroller und CPU angesprochen werden.

Als nächstes muss das EPICS Kontrollsystem über TCP/IP im SoC zum Laufen gebracht werden. Mit Hilfe dieser Integration soll das erste API anhand des EPICS-Treibers angesteuert werden.

2 Grundlagen

Vor der Realisierung der ersten Teilaufgabe, u.a. die Implementierung eines Application Programming Interface für die Steuerung von CAN-Geräten mit EPICS, ist die allgemeine Beschreibung von einigen Grundbegriffen wie CAN, Mob usw. erforderlich. Diese Begriffe werden in diesem Kapitel näher erläutert.

2.1 CAN - Control Area Network

Der folgende Abschnitt wird aus [12] entnommen.

Das CAN-Protokoll, *Control Area Network* Protokoll, ist ein Kommunikationsprotokoll, das ursprünglich für den Austausch von Informationen innerhalb eines Kraftfahrzeugs gedacht war. Heute sind fast 80% der CAN-Anwendung im Automobilbereich angesiedelt und der verbleibende Anteil wird in der Automatisierungstechnik eingesetzt.

Die Abbildung 2:1 zeigt das grundsätzliche CAN-verteilte System. Der CAN-Bus ist eine Drahtleitung, an der beliebig viele CAN-Geräte(Steuergeräte) parallel angeschlossen werden können. Die beiden Leitungen des Busses sind als CAN-High (CANH) und CAN-Low (CANL) bezeichnet. Der Bus ist an beiden Enden mit jeweils 120 Ohm Widerstand abgeschlossen. Die maximale Spannungsdifferenz U_D = CANH – CANL beträgt 0,5V.

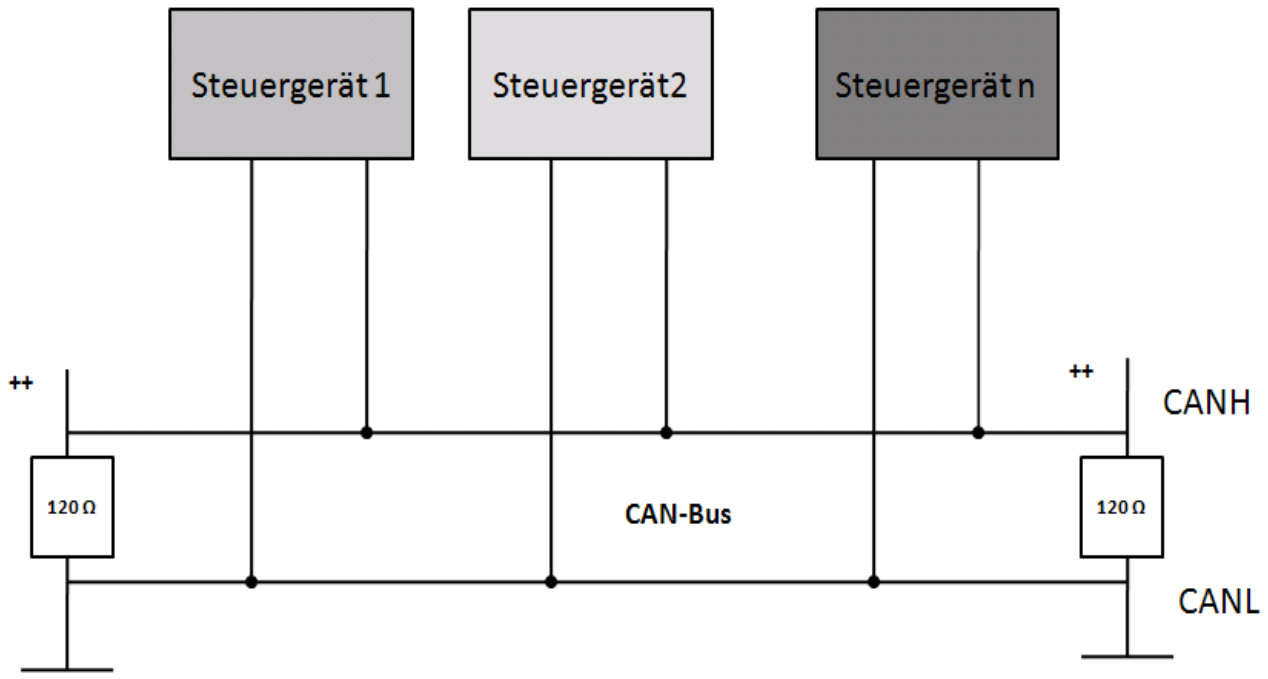

Abbildung 2:1 Physikalische Verbindung an dem CAN-Bus

Für CAN wurde eine Multi-Master-Hierarchie aus Gründen der Sicherheit und Verfügbarkeit gewählt. Falls ein Knoten defekt ist, bricht damit nicht das Gesamtsystem zusammen. In Anpassung an die Multi-Master-Eigenschaften wird bei CAN eine CSMACD + CR Mediumzugriffstechnik angewendet. Das heißt jeder Knoten, der auf das Kommunikationsmedium zugreifen möchte, wartet bis der Bus frei ist und startet dann die Übertragung einer Botschaft mit dem Aussenden des ersten Startbits, das zur Synchronisation aller Partner gedacht ist.

Jedes CAN-Telegramm besteht aus einer standardmäßig vordefinierten Bit-Struktur, unterteilt in bestimmte Felder. Die untere Tabelle 2:1 stellt den Aufbau einer CAN-Nachricht dar.

Start 1 Bit	Identifier 11 Bit	RTR 1 Bit	IDE 1 Bit	r0 1 Bit	DLC 4 Bit	DATA 0..8 Byte	CRC 15 Bit	ACK 12Bit	EOF+IFS 1 0 Bit

Tabelle 2:1 Aufbau des Standard-Frames nach Standard CAN 2.0A

Die eigentliche Kollisionserkennung und -auflösung wird bitweise über dem Identifikationsfeld ausgeführt. Diese Art der bitweisen Arbitrierung setzt dementsprechend voraus, dass die Implementierung der Bit-Werte 0 und 1 folgendermaßen vorgenommen wird. Die logische 0 als dominanter Bit-Wert mit hoher Priorität und die logische 1 als rezessiver Bit- Wert mit niedriger Priorität. Die Kommunikation bei CAN ist grundsätzlich Broadcast-orientiert. Ein Erzeuger von Informationen, der den Buszugriff gewonnen hat, sendet seine Information an alle anderen Knoten. Jeder Knoten empfängt diese Botschaft und prüft sie auf Fehler. Ob die so empfangene Botschaft jedoch auch weiter an den Rechner gegeben wird, hängt davon ab, ob der Name der empfangenen Botschaft durch den im Empfänger spezifizierten Botschaftsfilter akzeptiert wird.

Die Botschaften werden durch ihre Identifier gekennzeichnet, die gleichzeitig die Priorität einer Botschaft beinhalten. Der Identifier kennzeichnet die Quelle der Nachricht und nicht den Knoten.

Die Fehlerbehandlungstechnik bei CAN ist auf einem 15-Bit-CRC mit einer Hamming-Distanz von 6 basiert. Damit können 5 Einzelbitfehler pro Botschaft erkannt werden.

CAN sieht ein globales Quittungsfeld vor, das dem Sender die Information gibt, dass wenigstens einer der Knoten, oder auch keiner, die Botschaft richtig empfangen hat.

Die Bitcodierung ist NRZ. Das bedeutet, dass beim Informationstransfer nur eine geringe Bandbreite gefordert wird.

Die Bitrate ist programmierbar zwischen 5 kBit/s und 1 MBit/s.

2.2 Message Object Block

Der folgende Abschnitt wird aus [12]entnommen.

Der *Message Object Block* oder die *Mailbox* dienen als Puffer zwischen zwei parallelen Prozessoren. Diese Mailbox ist im CAN-Controller integriert, und besteht aus zwei Implementierungsformen: die sogenannten BASIC CAN- und FULL CAN-Strukturen. Der Unterschied zwischen den beiden Formen liegt in der Art, wie die vom Bus empfangenen Botschaften ausgefiltert werden, bevor sie in die Empfangs-Mailboxen geschrieben und ignoriert werden. Die Botschaften, die den Empfangsfilterprozess passiert haben, gelangen über die Mailbox zu der eigentlichen Anwendung.

Der Mikrocontroller auf das HADControl-Board ist ein AT90CAN128 der Firma Atmel mit integriertem FULL CAN-Controller. Somit handelt es sich in den folgenden Abschnitten und Kapiteln um CAN-Botschaften mit FULL CAN-Struktur.

Bei FULL CAN können die Mailboxen beliebig als Empfänger oder Sender programmiert sein. Jede der Mailboxen kann eine komplette Sende- oder Empfangsbotschaft speichern. Das beinhaltet den eindeutigen Identifier, die entsprechenden Daten, die Datenlänge und den Interrupt, der den erfolgreichen Empfang oder das Versenden einer Botschaft kennzeichnet.

Die Abbildung 2:2 stellt den allgemeinen Aufbau des CAN-Controllers dar.

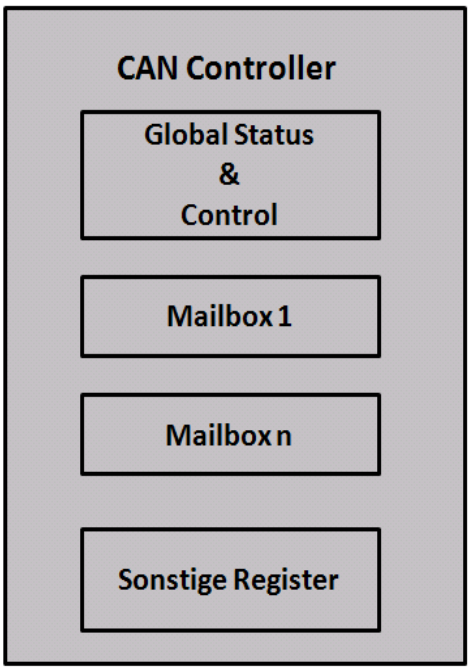

Abbildung 2:2 Allgemein gültiger Aufbau eines CAN-Controllers [12]

Im Folgenden wird die Funktionalität jeder Komponente des CAN-Controllers kurz beschrieben.

- Im globalen Status- und Kontrollregister werden Funktionen wie z.B. ein Software-Reset oder die Einstellung der Bit-Timing-Parameter eingetragen.
- Im Mailbox-Bereich sind die einzelnen Botschaften abgelegt.
- Bei dem sonstigen Register werden die Einstellungen vorgenommen, die im eigentlichen Sinn mit CAN nichts zu tun haben.

Jede der Mailboxen im CAN-Controller ist wiederum in 3 Bereichen aufgeteilt. Die Abbildung 2:3 zeigt die Struktur einer Mailbox.

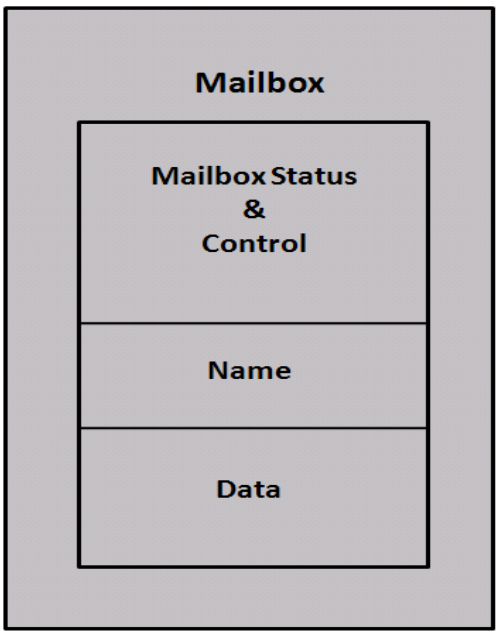

Abbildung 2:3 Allgemein gültiger Aufbau einer Mailbox [12]

Die Komponenten der Mailbox haben folgende Eigenschaften:

- Im Status- und Kontrollregister sind Status- und Kontrollinformationen vorhanden, die direkt zu einer Mailbox gehören.
- Der Name ist verknüpft mit der Priorität des Telegramms. Er ist der Identifier, der auf dem Bus übertragen wird.
- Der Bereich Data beinhaltet die Nutzdaten des Telegramms.

2.3 Interrupt

Der folgende Abschnitt wird aus [2] und [4] entnommen.

Weil es sich beim Großteil dieser Aufgabe um die Programmierung eines Mikrocontrollers handelt, ist das Verhalten des Interrupts im Mikrocontroller zu erläutern.

Der Interrupt oder die Unterbrechung ist eine Routine, die auf Ein-/Ausgabe-Ereignisse schnell reagieren kann, während ein anderer Programmcode abgearbeitet wird.

Bei bestimmten Ereignissen in einem Prozessor wird ein Interrupt ausgelöst. Dabei wird das Hauptprogramm angehalten und das Unterprogramm, die so genannten Unterbrechungsroutine ISR, aufgerufen. Wenn die Bearbeitung des Unterprogramms beendet ist, läuft das Hauptprogramm ganz normal weiter.

Bei Mikrocontrollern können Interrupts auf einige Ereignisse auftreten, z.B. während einer seriellen Übertragung, während eines Zählvorgangs und während einer Messung.

Am besten soll die ISR schnell beendet werden, sonst können z.B. Daten während einer seriellen Übertragung verloren gehen oder beim Timer Zählzyklen verloren gehen.

Im Mikrocontroller existieren verschiedene Interrupts, die wie andere Funktionen im Mikrocontroller behandelt werden müssen.

Bei AVR ist die Globale Interruptsteuerung durch das I-Bit im Register SREG definiert. Es ist ein Hauptschalter und kann global alle Interrupts ein- und ausschalten. Falls das I-Bit gesetzt ist und die Interrupts freigegeben sind, dann wird das Interrupt ausgeführt.

Im Gegensatz zum Globalen Interrupt, sind lokale Interrupts über Maskenbits in mehreren Registern definiert. Die jeweilige Interruptquellen werden individuell ein- und ausgeschaltet.

Eine ISR-Funktion wird nur ausgeführt, wenn die Interrupts global frei geschaltet sind, das individuelle Maskenbit gesetzt ist und das Interrupt eintritt.

Die Abbildung 2:4 beschreibt den allgemeinen Programmablauf eines Interrupts.

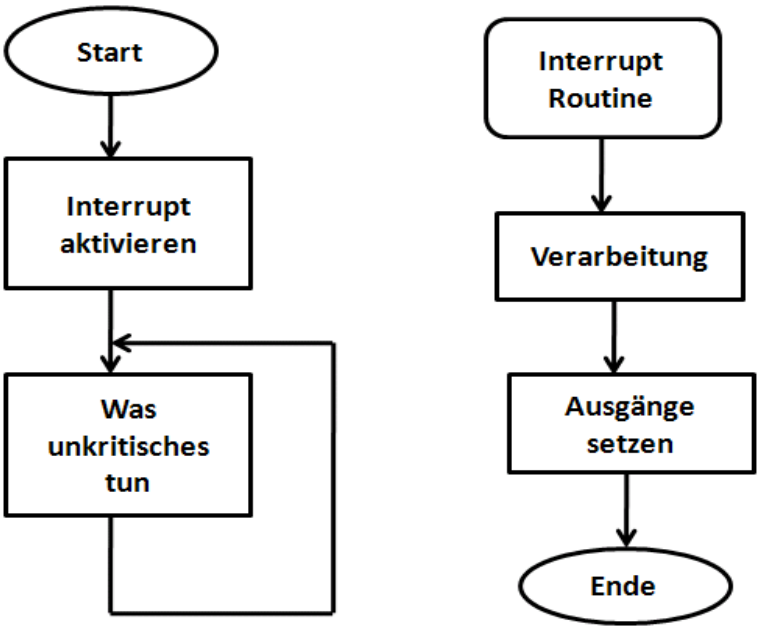

Abbildung 2:4 Programmablauf von Interruptsteuerung [4]

2.4 Treiber

Ein Treiber oder Driver ist ein Modul, welches die Interaktion mit angeschlossenen oder eingebauten Geräten steuert. Ein Driver Support ist eine Software-Schicht zwischen Code und Treiber, welche die Kommunikation mit Devices ermöglicht.

3 Lösungskonzept

Um das gestellte Problem Implementierung eines Application Programming Interface für die Streurung von CAN-Geräten mit EPICS zu lösen, ist zuerst der Lösungsansatz zu definieren, dann ein Überblick über das gesamte zu haben und danach eine kurze Beschreibung über die verschiedenen benutzten Komponenten zu erläutern.

3.1 Lösungsansätze

Das implementierte Application Programming Interface im Mikrocontroller soll in der Lage sein sowohl die Kommandos aus der seriellen Schnittstelle als auch die Daten aus CAN-Geräten zu interpretieren. Um dieses Ergebnis zu erreichen werden drei Ansätze definiert.

3.1.1 Kommunikationsprotokoll

Das erste Teil ist die Definition eines Protokolls für die serielle Schnittstelle.

3.1.2 Implementierung der API

Auf Basis das definierte Kommunikationsprotokoll wird die API implementiert.

3.1.3 Konfiguration einer EPICS Device Support

Im letzen Teil soll das implementierte API von einem EPICS Client direkt angesprochen und ausgelesen werden.

3.2 Überblick über das gesamte System

Die Abbildung 3:1 zeigt den Aufbau des HADControl-Boards und die für die Realisierung dieser Arbeit zusammenhängenden Subsysteme.

Ein EPICS Client hat den Zugang zum Server durch das so genannte Channel Access Protocol. Der Input Output Control(IOC)-Server verwaltet die Prozessvariablen, und das Device Support Modul stellt die Verbindung zum Mikrocontroller her. Der Wert dieser Prozessvariablen wird durch die serielle Schnittstelle an den Mikrocontroller gesendet. Der Mikrocontroller interpretiert diesen Wert und sendet ihn weiter an die verschiedenen CAN-Geräte. Die Daten von CAN-Geräten werden ebenfalls über den CAN-Bus vom Mikrocontroller empfangen, interpretiert und weiter an die CPU gesendet.

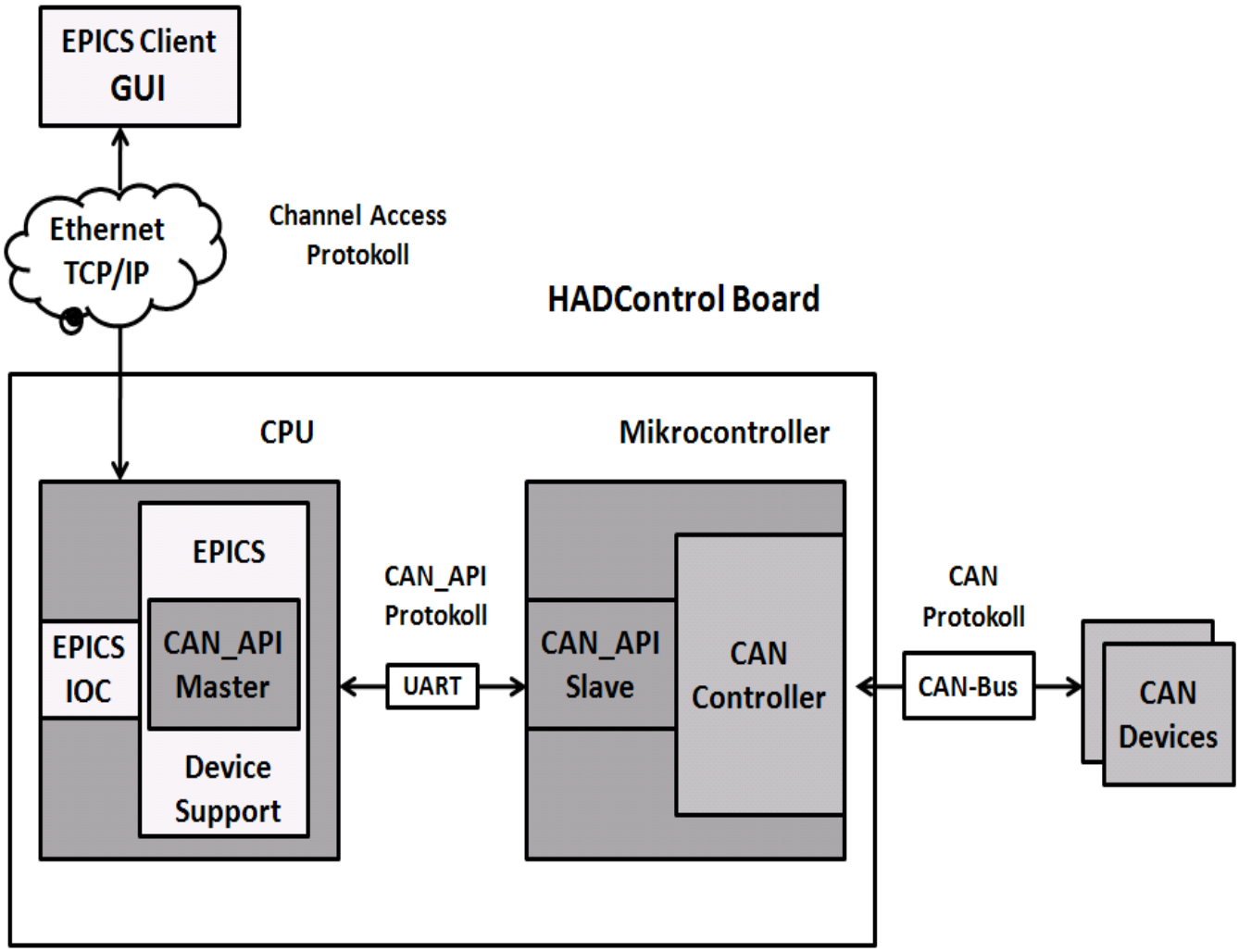

Abbildung 3:1 Schaltbild für die Steuerung von CAN-Geräten mit EPICS-Befehle

Zur Konzipierung werden die verschiedenen Systeme im kommenden Abschnitt beschrieben. Die verwendeten Systeme sind in drei Subsysteme unterteilt.

3.3 Verwendete Hardware

3.3.1 HADControl Board

Das HADControl ist ein Board, das an der GSI entwickelt wurde. Es besteht aus den folgenden Komponenten:

- Einem 8 Bit Mikrocontroller AT90CAN128 der Firma Atmel mit CAN und **JTAG** Schnittstelle, 128 kBytes **FLASH**, 4 kBytes **RAM** und 4 kBytes **EEPROM** -Speicher.
- Einem CAN Controller, der im Mikrocontroller integriert ist. Der CAN Controller besteht aus 15 Full Message Object mit getrenntem Identifier und Mask. Es unterschützt zwei Versionen von CAN Standards: CAN 2.0A mit 11 Bits Identifier und CAN 2.0B mit 29 Bits Identifier. Die verschiedenen Modes im AT90CAN128 sind Transmit, Receive, Automatic Reply und Frame Buffer Receive.
- Ein ETRAX SoC (Ethernet Token Ring Axis System on Chip) ist eine Central Processing Unit (CPU) der Firma AXIS Kommunikation. Sie basiert auf dem Code Reduced Instruction Set (CRIS). Das so genannte SoC-Modul besteht aus dem ETRAX 100LX MCM (Multi Chip Modul), mit 16 MB **SDRAM**, 4 MB FLASH, 10/100 MBits/s Ethernet und vier seriellen Schnittstellen mit hoher Geschwindigkeit bis zum 6250Kbit/s.

. Die Abbildung 3:2 zeigt das HADControl Board und seine Peripherie.

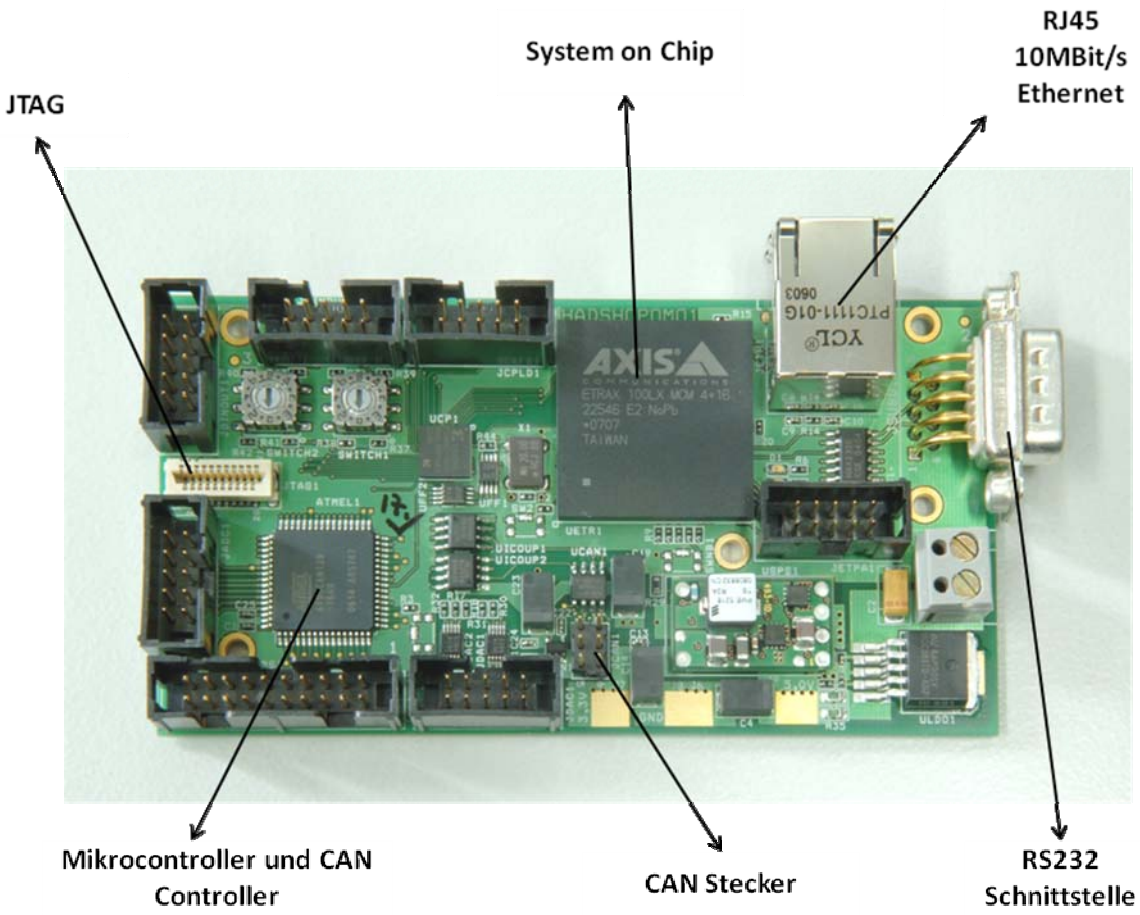

Abbildung 3:2 HADControl Board

3.3.2 CAN Devices

Unter CAN Devices versteht man die Geräten, die mindestens aus einer CAN-Schnittstelle besteht und nach dem CAN-Protokoll mit anderen Modulen kommunizieren.

3.4 Verwendete Software

3.4.1 C-Programmiersprache

C ist eine imperative Programmiersprache, die vom Informatiker Dennis Ritchie im Jahr 1970 entwickelt wurde. C steht auf fast allen Computersystemen zur Verfügung. Das Haupteinsatzgebiet von C liegt in der Systemprogrammierung. C ermöglicht direkt Speicherzugriff und damit bietet sie die Möglichkeit an, Hardware direkt anzusprechen.

3.4.2 Experimental Physics and Industrial Control System

Das EPICS ist ein Kontrollsystem mit folgenden Eigenschaften:

- Es ist eine Kontrollsystem Architektur: Ein Client/Server Modell mit einem effizienten Kommunikationsprotokoll (Channel Access) für den Austausch einfacher Daten.
- Eine Software Werkzeugsammlung: Eine Sammlung von gemeinschaftlich entwickelten Software Werkzeugen zum Aufbau eines verständlichen und erweiterbaren Kontrollsystems.
- Der Input Output Control Server ist meist ein Computer auf dem ein iocCore läuft. Es ist eine Sammlung von EPICS Routinen, die benutzt werden um die Prozessvariablen zu publizieren und Echtzeitalgorithmen anzuwenden. Das iocCore wird im Gegensatz ein Record benutzen um die Prozessvariablen zu definieren.

3.5 Verwendete Protokolle

Für das ganze System werden vier verschiedene Protokolle definiert und benutzt.

3.5.1 Control Area Network-Protokoll

Das CAN-Protokoll ist in Kapitel 2:1 beschrieben.

3.5.2 CAN_API-Protokoll

Das CAN_API steht für die Kommunikation zwischen Mikrocontroller und den CAN-Geräten, dieses Protokoll wird mehr in dem nächsten Kapitel beschrieben.

3.5.3 UART-Protokoll

Der folgende Abschnitt wird aus [LAWR99]entnommen.Das Universal Asynchronous Receiver Transmitter Protokoll ist eine Einrichtung, die digitalen Datenstrom seriell zu übertragen ermöglicht. Es wird meistens für die Datenübertragung zwischen Mikrocontroller und PC genutzt. Die digitalen Datenströme sind mit einem fixen Rahmen aufgebaut, der aus einem Start-Bit, einem optionalen Parity-Bit, einem 5-9 Datenbit und einem Stopp-Bit besteht. Für die Übertragung von Daten sind zwei Pins am Mikrocontroller nötig. Ein sendender Pin und ein empfangender Pin.

Die Synchronisation bei dem Empfänger besteht aus einem Start- und Stopp-Bit und einer bestimmten Bitrate.

3.5.4 Channel Access-Protokoll

Das sogenannte Channel Access(CA) ist das Kommunikationsprotokoll, das für das EPICS-System angewendet ist, und mit dessen Hilfe auch Informationen über das Netzwerk verschickt werden.

Der Channel Access Server ist eine Software, die Zugriffe über Channel Access auf Process Variablen zur Verfugung stellt. Der IOC ist ein CA Server für EPICS.

Der Channel Access-Client ist eine Software, die über Channel Access auf Process Variable zugreift. Das MEDM ist ein CA Client für EPICS.

Die Abbildung 3:3 stellt das Netzwerkbasierte Client/Server für EPICS dar.

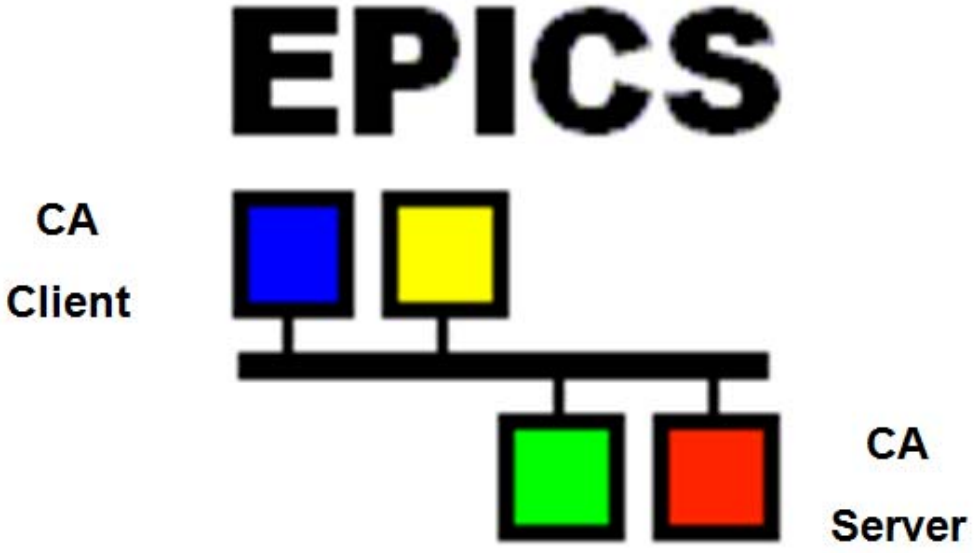

Abbildung 3:3 Netzwerkbasiertes Client/ Server Modell[10]

4 Lösungsweg

In diesem Kapitel werden die verschiedenen Schritte für die Lösung des gestellten Problems näher beschrieben.

4.1 Beschreibung des CAN_API-Protokolls

Das implementierte API wird in der Zukunft CAN_API genannt und soll über die serielle Schnittstelle zwischen Mikrocontroller und CPU angesprochen und gelesen werden. Das CAN_API besteht aus zwei Teilen. Eines ist der CAN_API–Master. Dieser CAN_API-Master ist ein Programm, das die API verwendet und das Protokoll kennt, es läuft auf der CPU und hat als Hauptaufgabe die Kommandos über eine serielle Schnittstelle zum Mikrocontroller zu senden. Das zweite Teil ist ein CAN_API-Slave, der im Mikrocontroller läuft und als Hauptaufgabe die angekommenen CPU-Kommandos und CAN-Daten bearbeitet.

Das CAN_API Modul ist in der Programmiersprache C implementiert. Das CAN_API-Protokoll beschreibt das Protokoll für die serielle Kommunikation, das heißt für den Datenaustausch zwischen der CPU und dem Mikrocontroller.

4.1.1 Datenformat

Die Daten bzw. Kommandos sind als String strukturiert und bestehen aus einzelnen Blöcken. Die verschiedenen Blöcke werden im Folgenden als Parameter bezeichnet. Jeder einzelne Parameter ist durch ein Leerzeichen getrennt.

4.1.2 Baudrate

Die Baudrate für die serielle Kommunikation ist bei 38400 Bit/Sekunde eingestellt. Die minimale Kommandolänge besteht aus 13 Bytes und die maximale aus 50 Bytes, d.h. für die Übertragung der Kommandos brauchen wir nur eine maximale Wartezeit von 3334 Mikrosekunden. Die Daten werden asynchron über die serielle Schnittstelle gesendet.

Die Clock Frequenz des Mikrocontrollers ist 5MHz. Diese Frequenz ist nicht für den Mikrocontroller AT90CAN128 vordefiniert, sie wird wegen anderer Hardware Komponenten des HAD-Control Boards gewählt. Mit dieser Frequenz kann eine maximale Baudrate von 38400 Bit/Sekunde benutzt werden. Mit der Clock Frequenz werden einige Baudraten gewählt und die

zugehörigen Fehlerraten berechnet. Bei der seriellen Übertragung mit 8 Bit Datenübertragung wird eine maximale Fehlerrate von +/- 2% für eine fehlerfreie Übertragung erlaubt.

Baudrate = Fclk/16*(UBRR +1)

UBBR = (Fclk/(16*Baudrate)) -1

Error [%] = (1- (Baudrate $_{\text{closet Match}}$/Baudrate))*100

Baudrate ist die Übertragungsgeschwindigkeit in Bit pro Sekunde

UBBR ist der Wert, der im Register des Mikrocontrollers gesetzt wird.

Fclk ist die Frequenz des Systems für Eingang bzw. Ausgang

Siehe [12] für mehr Information.

Die Tabelle 4:1 stellt die Ergebnisse und die Beurteilung für die 5MHz Clock Frequenz dar.

Baudrate (bps)	UBBR	Error(%)	Evaluation
4800	64	0.2	acceptable
9600	32	-1,4	acceptable
38400	7	1.7	optimal
57.6K	4	8,5	no acceptable
115.2K	2	-9.6	no acceptable

Tabelle 4:1 Berechnung von Fehlerprozent für die serielle Kommunikation

4.1.3 Parameterliste

Die gesendeten Kommandos von der CPU bzw. empfangenen Daten vom CAN-Bus bestehen aus einer Reihe von Parametern. Die Tabelle 4:1 stellt die Liste der verwendeten Parameter, deren Bedeutung sowie die erlaubten Werte dar.

Parametername	Bedeutung	Erlaubte Werte
Command-Name	Name des zu sendenden Kommandos	SEND/SUBS/USUB
Command-Response	Antwort auf Kommando	RECV/ERRA/ERRC/ERRM
DATA	Datenbyte	0x0 …FF
Error-Code	Nummer von Fehler	0…9
Error-Description	Beschreibung von Fehler	0x41…0x5A und 0x61..0x71
Length	Anzahl der Datenbyte	0…8
Message-ID	CAN Message ID	0x0 …FFFFFFF1
MOB-Number	Message Object Block	0...14
Range-ID	CAN Message ID-Mask	0x0 …FFFFFFF1
RTR	Remote Transmission Request	0/1

Tabelle 4:2 Liste von verwendete Parameter, Bedeutung und Werte für das CAN_API-Protokoll

Command-Name ist ein Schlüsselwort, das die eigentliche Aufgabe des Kommandos bestimmt. Es ist immer 4 Byte lang und wird von der CPU zum Mikrocontroller gesendet.

Command-Response ist ein Schlüsselwort, das den Antworttyp vom Mikrocontroller charakterisiert. Es ist ebenfalls, wie der Parameter "Command-Name" 4 Bytes lang. Die Commando-Response wird immer vom Mikrocontroller zur CPU versandt. Wenn die Kommunikation erfolgreich war, werden die Daten in Hexadezimal kodiert. Falls es sich jedoch um eine Fehlermeldung handelt, werden die Fehler in ASCII-Character kodiert.

DATA enthält die Daten des Telegramms, das versandt bzw. empfangen wird. Die Daten sind maximal 8 Bytes und minimal 0 Bytes lang.

Error-Code gibt die spezifische Fehlernummer sowohl für die serielle Kommunikation als auch für die CAN-Kommunikation.

Error-Decription gibt eine genaue Beschreibung des Fehlers in Form eine String-Text in Abhängigkeit von Error-Code.

Length enthält die Länge der Daten, die versandt bzw. empfangen werden. Die Länge kann maximal 8 sein, weil die CAN Message aus maximal 8 Bytes besteht.

Message-ID steht für Message Identifier und kennzeichnet den Inhalt der Nachricht. Der Empfänger entscheidet anhand des Identifiers, ob die Nachricht für ihn relevant ist oder nicht. Der

Identifier dient dann zur Priorisierung der Nachrichten. Er ist maximal 11 Bit lang für CAN-Controller 2.0A und 29 Bit lang für CAN-Controller 2.0B.

MOB-Number ist die Nummer des Übertragungskanals, die für die Übertragung von Daten auf den CAN-Bus benutzt wird.

Range-ID dient zur Maskierung von bestimmten Message-ID-Nummern. Die Range-ID besteht aus maximal 11 Bit für CAN-Controller 2.0A und 29 Bit lang CAN-Controller 2.0B.

RTR steht für Remote Transmission Request. Er unterscheidet zwischen Daten und Datenanforderungstelegrammen. RTR besteht aus einem einzelnen Bit. Ein gesetztes RTR-Bit kennzeichnet einen Remote-Frame (rezessiv). Mit Hilfe eines Remote-Frames kann ein Teilnehmer einen anderen auffordern, seine Daten zu senden.

4.1.4 Erläuterung von Command-Name und Response-Name

Die Abbildung 4:1 zeigt eine Befehls- und Antwortübersicht zwischen Mikrokontroller und CPU.

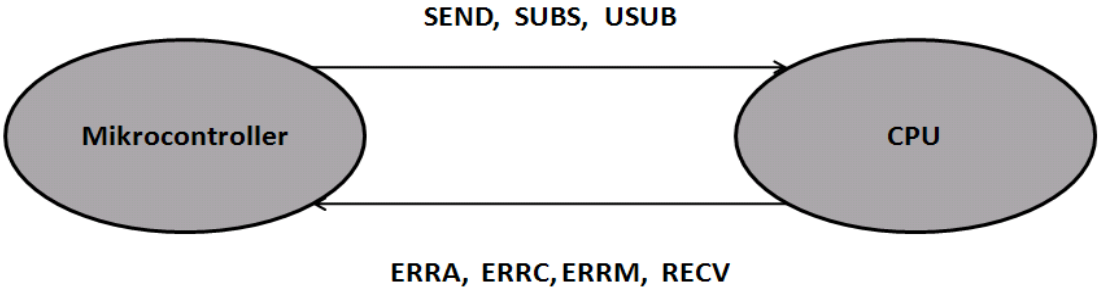

Abbildung 4:1 Sende- und Antwortschlüsselwörter

Im Folgenden wird erklärt, wie die Kommandos erweitert werden und wie die Antwort entschlüsselt wird.

4.1.4.1 Command-Name in Richtung CPU-Mikrocontroller

Der Command-Name kann aus einem dieser 3 verschiedenen Schlüsselwörter bestehen.

SEND

Kommando für das Lesen und Schreiben auf den CAN-Bus.

SUBS

Registrierung für das Lesen bestimmter Message-ID, die auf dem CAN-Bus vorhanden sind.

USUB

Abmeldung für das Lesen bestimmter Message-ID, die auf dem CAN-Bus vorhanden sind.

4.1.4.2 Command-Response in Richtung Mikrocontroller-CPU

Die Command-Response kann aus einem dieser 4 verschiedenen Schlüsselwörter bestehen.

RECV

dieses Schlüsselwort kennzeichnet die erfolgreiche Kommunikation zwischen der CPU und den CAN-Geräten durch den Mikrocontroller. Die Antwort besteht zuerst aus dem Schlüsselwort, dann die Nummer des Kanals, dann der Message-ID, danach der Länge von empfangenen Daten und am Ende dem Maximum von 8 folgenden Datenwörtern.

ERRA (Error im CAN_API-Protokoll)

bedeutet, dass ein Fehler während der seriellen Kommunikation aufgetreten ist. Nach dem Schlüsselwort folgen die Fehlernummer und dann die genaue Fehlerbeschreibung in Form von Klartext.

ERRC (Error in CAN-Protokoll)

bedeutet, dass ein Fehler während der CAN Kommunikation aufgetreten ist. Nach dem Schlüsselwort folgen die Fehlernummer und dann die genaue Fehlerbeschreibung in Form von Klartext.

ERRM (Error in M0b)

bedeutet, dass es zurzeit keinen freien Kanal auf dem CAN-Controller für die Ausführung des Kommandos gibt. Nach dem Schlüsselwort folgen die Fehlernummer und dann die genaue Fehlerbeschreibung in Form von Klartext.

4.1.5 Datenformat-Struktur

Hier wird kurz beschrieben, in welcher Reihenfolge die Kommandos bzw. Daten strukturiert werden sollen.

4.1.5.1 Struktur für Kommandos

Die Kommandos sind in folgender Form strukturiert:

“Commando-Name Message-ID Range-ID RTR Length DATA“

Falls ein Parameter zwischen Commando-Name und DATA ausfällt soll, wird er durch ein 0 ersetzt.

Beispiel:

Notwendige Parameter für das Lesen der Lüftergeschwindigkeit

“Commando-Name Message-ID Range-ID RTR Length“

Format: **“SEND 304 0 1 8 “**

Beispiel:

Notwendige Parameter für das Anmelden einer bestimmten Message-ID

“Commando-Name Message-ID Range-ID“

Format: **“SUBS 104 F77“**

Beispiel:

Notwendige Parameter für das Abmelden einer bestimmten Message-ID

“Commando-Name Message-ID Range-ID“

Format: **“USUB 104 000“**

4.1.5.2 Struktur der Daten

Die erste Struktur bezeichnet die empfangenen Daten aus dem CAN-Bus im Fall eines request-Fall, die zweite Struktur bezeichnet die empfangenen Daten aus dem CAN-Bus im Fall eines nicht request-Fall und die dritte die möglichen Fehler während der Kommunikation. Die MOB-Number, Message-ID, 8 CAN-Daten und der Anzahl von empfangenen CAN-Daten sind in Hexadezimal dargestellt.

“Commando-Response MOB-Number Message-ID Length DATA“

“Commando-Response Bestätigung“

“Commando-Response Error-Number Error-Description“

Beispiel:

Antwort auf die Lüftergeschwindigkeit besteht aus folgenden Parametern:

“Commando-Response Mob-Number Message-ID Length DATA“

Format: **“RECV 0 304 8 83 b8 31 32 31 ff ff ff “**

Beispiel:

Antwort ein

“Commando-Response Commando will carried out”

Format: **“RECV commando will carried out“**

Beispiel:

Fehler während der seriellen Kommunikation

“Commando-Response Error-Number Error-Description”

Format: **“ERRA 6 Data is out of Range“**

Beispiel:

Fehler während der CAN-Kommunikation

“Commando-Response Error-Number Error-Description”

Format: **“ERRC 1 CAN Channel is off “**

Beispiel:

Fehler für das Message Object Block

“Commando-Response Error-Number Error-Description”

Format: **“ERRM 2 Find no MOb“**

4.2 Implementierung des CAN_API-Slaves

Die Implementierung von CAN_API-Slaven soll im Mikrocontroller laufen. Dieser hat zwei Hauptrollen, einerseits soll er in der Lage sein die Befehle von der CPU zu interpretieren und durchzuführen, anderseits die angekommenen Daten von CAN-Geräten in eine CAN_API-Struktur einzupacken und an die CPU weiter zu leiten. Die Implementierung des CAN_API-Slaves ist durch Interrupt gesteuert. Die Abbildung 4:2 stellt seinen Programmablaufplan dar.

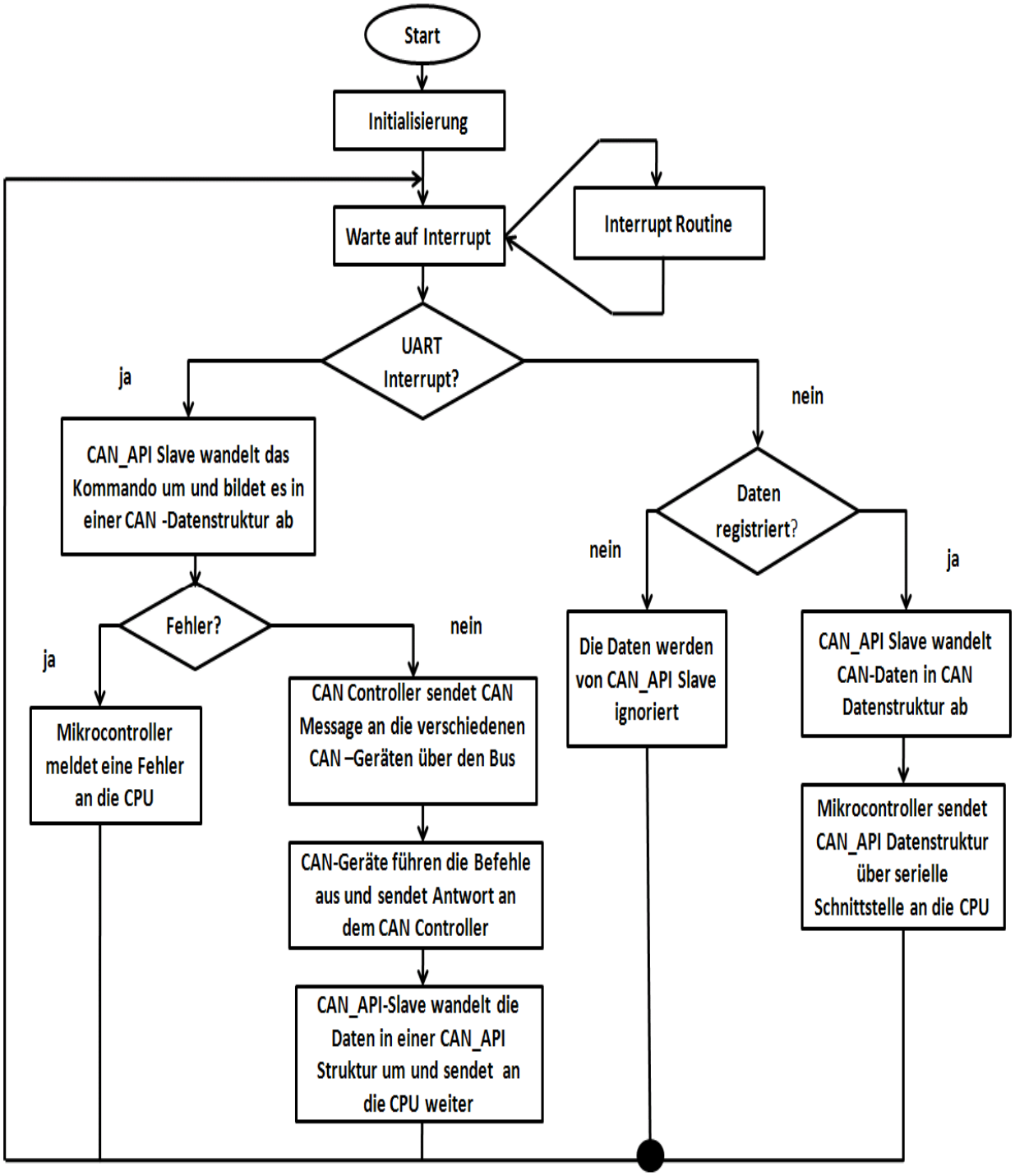

Abbildung 4:2 Programmablaufplan für den CAN_API-Slave

Die Tabelle 4:3 stellt die implementierten Funktionen für CAN_API-Master dar. Die rechte Spalte listet die Funktionen spezifisch für das CAN_API-Protokoll aus und die linke Spalte die Funktionen spezifisch für das CAN-Protokoll.

CAN_API Funktionen	**CAN Funktionen**
Convert _UartFormat_to_CanFormat Decrypt_Uart_String Choose_Function	CAN_Init CANIT_vect Convert_CanFormat_to_UartFormat
Communication_Error Convert_pack_canFrame_to_uartFormat General_Communication_Error Get_in_Ringbuffer Put_in_Ringbuffer Receive_Message Send_Message SIG_UART_RECV Subscribe_Message Timer0_Init Timer0A_Init	Get_BusState Get_CanError Get_FreeMob Get_Interrupt_of_Mob Init Wait_for_Can_Send_Finished Wait_for_Receive_Finished
UART0_Init UART0_Send_Message_String UART0_Transmit Unsubcribe_Message	

Tabelle 4:3 Liste von implementierten Funktionen für den CAN_API-Master

In diesem Abschnitt werden kurz die Hauptaufgaben der implementierten CAN_API-Master Funktionen erläutert. Erstmal werden die implementierten Funktionen spezifisch für das CAN_API, dann anschließend für das CAN erklärt.

4.2.1 Funktionalität der implementierten CAN_API Funktionen

`void Convert_UartFormat_to _CanFormat(char String[13][8])`

Diese Funktion besteht aus zwei Teilen. In dem ersten Schritt werden die einzelnen Inhalte der zweidimensionalen Array im Hexadezimal umgewandelt und dann an die verschiedenen Elemente einer Struktur weitergeleitet. Im zweiten Schritt werden die einzelnen Elemente der Struktur mit erlaubten Werten verglichen. Beim Fehlerauftritt wird die Funktion **Communication_Error** aufgerufen.

`void Decrypt_Uart_String(char decryptString[8])`

In dieser Funktion wird das empfangene CPU-Kommando in verschiedene Blöcke oder Parameter zerlegt. Diese einzelnen Parameter von dem empfangenen CPU-Kommando werden in einem zweidimensionalen Array gespeichert. Beim Auftreten eines Fehlers wird die Funktion **Communication_Error** aufgerufen.

`void Choose_Function(struct CPUFRAME *PtrCpu)`

ist abhängig von dem Commando-Name, das heißt der Inhalt der zugehörigen Funktion wird an der zweidimensionalen Array der ersten Position aufgerufen.

`uint8_t Communication_Error(const char Error[50], const uint8_t Error_Number, char Error_Typ[5])`

Die Funktion gibt die spezifischen Beschreibungen von allen Fehlern über CAN, UART und Mailbox. Die Eingabeparameter sind die Fehlerbeschreibung, die Fehlernummer und der Fehlertyp. Ein Rückgabewert 1 signalisiert einen Fehler und der 0 eine erfolgreiche Kommunikation.

`If Error_Typ = ERRA:` Die Funktion signalisiert einen Fehler während der seriellen Kommunikation. Sie ist zum Beispiel in der Funktion **Convert_UartFormat_to _CanFormat** aufgerufen, wenn ein Parameter außerhalb des erlaubten Bereichs liegt.

`If Error_Typ = ERRC:` Die Funktion signalisiert einen Fehler während der CAN Kommunikation. Sie wird automatisch in der Funktion **Get_CanError** bei jedem gesetzten Fehler-Bit aufgerufen

`If Error_Typ = ERRM:` Die Funktion signalisiert während der CAN Kommunikation einen Fehler falls kein freier Mob für die Übertragung von Nachrichten gefunden wird. Sie ist im Zusammenhang mit der definierten Funktion **Subscribe/Unsubscribe_Message.**

`uint8_t General_Communication_Error(const char Error[25], const uint8_t Error_Number)`

Die Funktion gibt die allgemeinen Fehlerbeschreibungen. Die Eingabeparameter sind die Fehlerbeschreibung und die Fehlernummer. Ein Rückgabewert 1 signalisiert einen Fehler und der 0 eine erfolgreiche Kommunikation.

`Int8_t Get_in_RingBuffer(void)`

Die Funktion wird die gespeicherte CAN-Daten aus dem Puffer auslesen. Der Rückgabewert 1 bedeutet, dass alle gespeicherten Daten ausgelesen sind. Der Rückgabewert 0 signalisiert, dass es noch nicht ausgelesenen Daten gibt.

`void Convert_pack_canFrame_to_uartFormat(struct CANFRAME *PtrCan)`

Die Funktion speichert die verschiedenen CAN-Daten in einem String. Der Eingangsparameter ist die definierte CAN-Struktur.

`Int8_t Put_in_RingBuffer(char putData[41])`

Die Funktion wird die empfangene CAN-Daten in einem Puffer speichern. Der Rückgabewert 1 bedeutet, dass der reservierte Puffer voll ist. Der Rückgabewert 0 signalisiert, dass es noch frei Speicherplatz gibt.

`void Receive_Message (struct CPUFRAME *PtrCpu)`

ist die zugehörige Funktion für den Commando-Namen "SEND" und mit gesetzten RTR. In der Funktion werden die benötigten Register mit den Strukturelementen initialisiert. Diese Funktion wird in der **Choose-Funktion** aufgerufen.

`void Send_Message (struct CPUFRAME *PtrCpu)`

ist die zugehörige Funktion für den Commando-Namen "SEND". In der Funktion werden die benötigten Register mit den Strukturelementen initialisiert. Diese Funktion wird in der **Choose-Funktion** aufgerufen.

`ISR(SIG_UART_RECV)`

ist eine Interrupt Service Routine Funktion zum Empfangen von Kommandos aus der seriellen Schnittstelle. Sie wird in dem Fall verlassen, falls ein Line Fine gefunden wurde.

`void Subscribe_Message(struct CPUFRAME *PtrCpu)`

ist die zugehörige Funktion für den Commando-Namen "SUBS". In der Funktion werden die benötigten Register mit den Strukturelementen initialisiert. Diese Funktion wird im **Choose_Function** aufgerufen.

`int8_t Timer0_Init(void)`

diese Funktion dient zur Initialisierung des ersten Zählers und wird in der Main-Funktion aufgerufen. Der Rückgabewert 1 signalisiert eine erfolgreiche Initialisierung.

`int8_t Timer0A_Init(void)`

diese Funktion dient zur Initialisierung des zweiten Zählers und wird in der Main-Funktion aufgerufen. Der Rückgabewert 1 signalisiert eine erfolgreiche Initialisierung.

`int8_t UART0_Init(void)`

diese Funktion initialisiert die Kommunikation zwischen CPU und Mikrocontroller und wird in der Main-Funktion aufgerufen. Die benutzte Übertragungsgeschwindigkeit wird hier gewählt. Der Rückgabewert 1 signalisiert eine erfolgreiche Initialisierung.

`int8_t UART0_Send_Message_String(char *tempstr)`

diese ist eine Hilfefunktion, sie ist verwendet, um alle Daten unter ein String-Format von dem Mikrocontroller nach der CPU zu senden.

`void UART0_Transmit(unsigned char c)`

diese Funktion sendet 8 Bit aus dem Mikrocontroller. Um diese Funktion zu nutzen, muss man sicher sein, dass die Daten nicht mehr als 8 Bit sind.

`void Usubscribe_Message(struct CPUFRAME *PtrCpu)`

ist die zugehörige Funktion für den Commando-Namen “USUB“. In der Funktion werden die benötigten Register mit den Strukturelementen initialisiert. Diese Funktion wird im **Choose_Function** aufgerufen.

4.2.2 Funktionalität der implementierten CAN Funktionen

`int8_t CAN_Init(void)`

zur Initialisierung der Kommunikation zwischen Mikrocontroller und CAN-Geräten. Die Übertragungsgeschwindigkeit wird in dieser Funktion definiert. Der Rückgabewert 1 bedeutet eine erfolgreiche Kommunikation und -1 signalisiert einen Fehler.

`ISR(CANIT_vect)`

ist eine Interrupt Service Routine Funktion zum Abarbeiten der von der CAN Kontrolleinheit empfangenen Daten aus dem CAN-Bus. Sie wird verlassen, falls die Übertragung beendet ist.

`void Convert_Can_Format_to_Uart_Format(struct CANFRAME *PtrCan)`

in dieser Funktion werden die empfangenen CAN-Daten in einem CPU-Daten-Format strukturiert. Die Funktion geht davon aus, dass die Ankunft einer Nachricht schon bei der Behandlung des Interrupts erkannt wird.

`void Get_BusStatus(void)`

Die Funktion gibt den CAN- Buszustand an, sie ist auch durch ein Timer-Interrupt gesteuert und wird automatisch jede 0.5 Sekunden aufgerufen.

`void Get_CanError(void)`

die Funktion gibt die verschiedenen Fehler während der CAN-Kommunikation an. Sie ist durch ein Timer-Interrupt gesteuert und wird automatisch jede 0.6 Sekunden aufgerufen.

`int8_t Get_CanFreeMob(void)`

ist eine kleine Routine, die die 15 Mailbox des CAN-Controllers scannt, und das erste freie Mob wählt, das für die Übertragung der Nachricht benutzt wird. Die Rückgabe -1 heißt, dass für die Übertragung kein freier Kanal gefunden wurde. Die Rückgabe freemob bedeutet, dass mindestens ein Mob für die Nachrichtübertragung gefunden wurde.

`uint8_t Get_Inerrupt_of_Mob(uint32_t bitmask)`

Die Funktion holt das Message Object Block, auf dem ein Interrupt aktiviert ist. Die Rückgabe mob bedeutet ein gültiges Message Object Block und NOMOB ein ungültiges Message Object Block.

`void Init(void)`

Aufräumen der benutzten Parameter nach dem Senden/Empfang einer Nachricht auf dem CAN-Bus.

`void Wait_for_Receive_Finished(void)`

die Funktion wartet auf die Bestätigung eines Busteilnehmers für das Senden von verschiedenen Kommandos in einem Remote-Request-Fall. Die Funktion wird im **Receive_Message** aufgerufen.

`void Wait_for_Send_Finished(void)`

die Funktion wartet auf die Bestätigung eines Busteilnehmers für das Senden von verschiedenen Kommandos in einem nicht Remote-Request-Fall. Die Funktion wird im **Send_Message** aufgerufen.

4.3 Implementierung des CAN_API-Masters

Die Implementierung des CAN_API-Masters soll in der CPU geladen werden. Das Programm steht für die Kommunikation über die serielle Schnittstelle zwischen der CPU und dem Mikrocontroller. Der CAN_API-Master hat die Aufgabe, die Befehle regelmäßig an den Mikrocontroller zu senden und die ankommenden Antworten auf der Terminal-Verbindung darzustellen. Das Programm ist unabhängig von EPICS.

Die Abbildung 4:3 stellt den Programmablaufplan für die Implementierung des CAN_API-Masters dar.

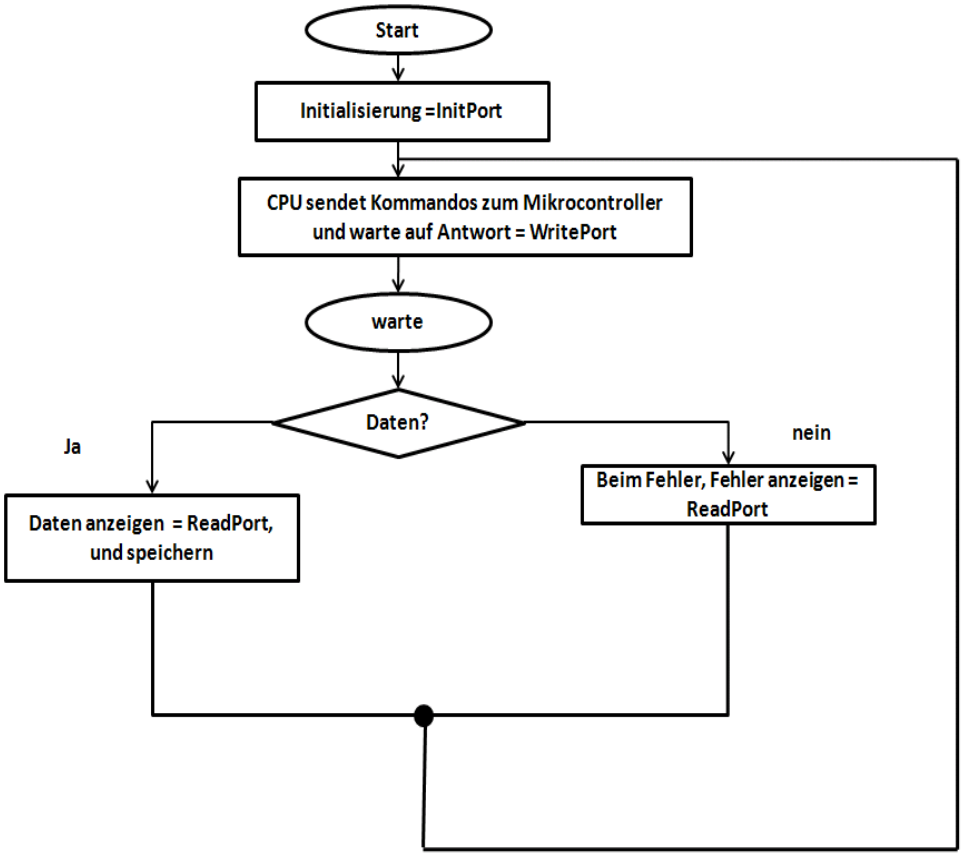

Abbildung 4:3 Programmablaufplan für CAN_API-Master

Die Implementierung des CAN_API-Masters besteht aus den folgenden drei Grundfunktionen.

Das benutze Integer-Eingabeparameter fd, der für die Definition der drei Funktionen benutzt wird, enthält die Beschreibung des definierten Ports für die Kommunikation zwischen der CPU und dem Mikrocontroller.

`int InitPort(int fd)`

> InitPort ist eine Funktion vom Datentyp Integer und liefert ein Integer als Rückgabeparameter. Die Konfiguration in dieser Funktion ist sehr entscheidend, weil die Konfiguration nicht nur die CPU sondern auch den Mikrocontroller betrifft. Als erstes wird definiert, durch welchen Port die beiden Module kommunizieren sollen. Danach kommen die Einstellungen wie Übertragungsrate, Datenformat usw. Der Rückgabewert 1 bedeutet, dass die Initialisierung erfolgreich war.

`int WritePort(int fd, char * chars)`

WritePort ist vom integer-Typ er nimmt als Eingabeparameter ein char und Integer Variablen und liefert ein Integer als Rückgabeparameter. In dieser Funktion wird der eigentliche String über den Port gesendet. Jeder String wird automatisch von einem “ Line Fine“ abgeschlossen. Der Rückgabewert 1 bedeutet, dass das Schreiben erfolgreich war.

`int ReadPort(void)`

Diese Funktion ist vom void-Typ und benötigt keinen Eingabeparameter und liefert ein Integer als Rückgabeparameter. Die Funktion soll automatisch alle empfangenen Daten auf dem Bildschirm anzeigen.

Die gelesenen Daten sind in einer Variablen von maximal 255 Charakteren kopiert. Die Variable ist vom Typ char und mit ein 0 Byte abgeschlossen. Wenn beim Lesen auf das Zeichen Line Fine gestoßen und ein Timeout erreicht wird, wird der Lesevorgang abgebrochen. Der Rückgabewert 1 bedeutet, dass das Lesen erfolgreich war.

4.4 Verbindung von HADControl mit EPICS

Damit das implementierte CAN_API für das HADControl von dem EPICS-Kontrollsystem angesprochen wird, soll erstens eine IOC in der CPU laufen, zweitens ist die Implementierung eines EPICS Device Support notwendig. Die Abbildung 4:4 zeigt die wesentlichen Softwarekomponenten einer IOC. Mit dem Einsatz des EPICS Kontrollsystems werden die Ergebnisse nicht mehr auf der Terminal-Verbindung wie im Abschnitt 4:3 dargestellt, sondern auf eine EPICS-grafische Benutzeroberfläche.

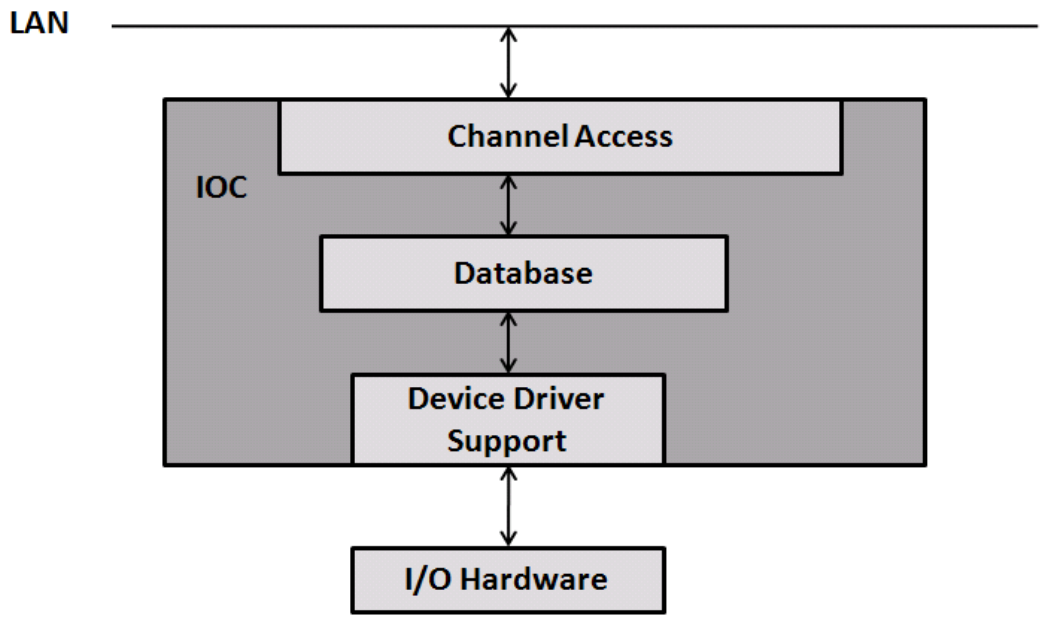

Abbildung 4:4 Physikalische Verbindung eines Devices mit EPICS [10]

- **LAN** ist das Netzwerk, über das die Kommunikation mit EPICS möglichst ist.
- **Channel Access** ist die Channel-Kommunikation für EPICS, sie wird genauer in Abschnitt 3.1.4 erläutert.
- **Database** definiert das sogenannte Record für jede einzelne Process Variable, die auf ein IOC implementiert ist. Das Record beschreibt unter anderem die Art der Verbindung mit einer Hardware.
- **Device Driver Support** ist ein Treiber, über den das Record mit einer Hardware kommuniziert.

Die Abbildung 4:5 stellt die physikalische Verbindung zwischen EPICS und dem HADControl Board dar.

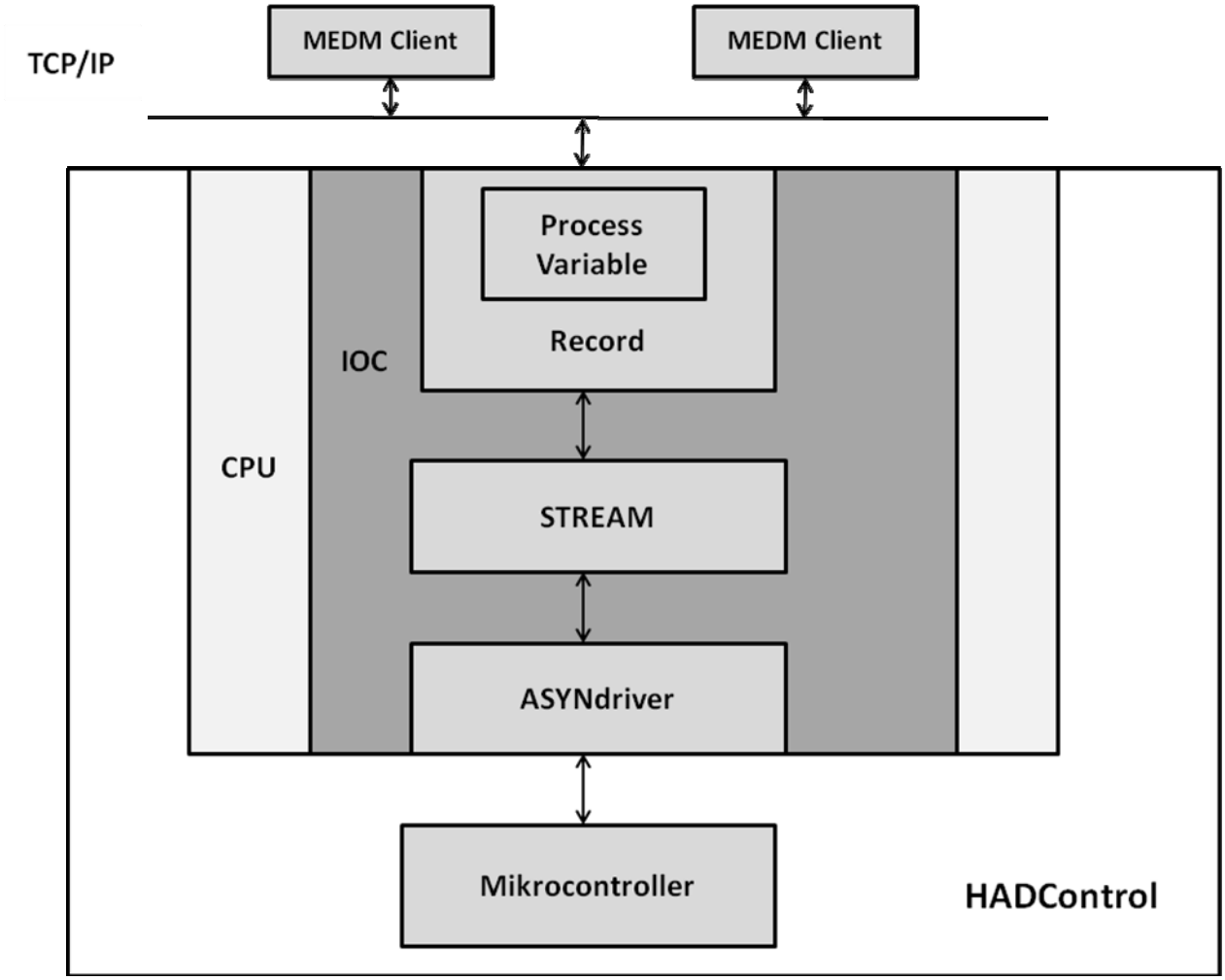

Abbildung 4:5 Physikalische Verbindung des HADControl-Boards mit EPICS

- **MEDM** steht für Motif Editor and Display Manager, es ist eine grafische Benutzeroberfläche für die Planung und Durchführung der Kontroll-Bildschirme.
- **Process Variable** ist ein Datensatz mit einem Namen, der über diese angesprochen werden kann. Die Prozess-Variable wird definiert über ein Record in der Datenbase der IOC.
- **STREAM** ist ein allgemeiner EPICS Device Support für Devices mit " Byte Stream". Er kontrolliert das Senden und Empfangen von Strings, einschließlich nicht druckbare Zeichen und null Bytes. Ein Beispiel für diese Art der Kommunikation ist die serielle Kommunikation. STREAM Device verwendet die Schnittstelle des ASYNdriver, der den Low-Level Support für diese Kommunikation durchführt. Er unterstützt alle Standard Record-Typen von EPICS.
- **ASYNdriver** ist der EPICS Device Support für asynchrone Kommunikation.

4.5 Dateikonfiguration

Damit das STREAM Device benutzt werden kann, muss eine Datei konfiguriert werden, unter anderem ein Protocol File, ein Record definiert werden, und ein IOC zum Laufen gebracht werden. Der Abschnitt 4.5.1 definiert das Protokoll des Devices.

4.5.1 Protocol File

Im Protocol File erfolgt die Konfiguration für die Kommunikation mit einem Device. Es wird ein einziges File für jedes Device benötigt. Es wird für jede Funktion ein Protokoll angelegt. Das Protokoll besteht aus einem Namen gefolgt von einer geschweiften Klammer.

Der nächste Block beschreibt die Konfiguration des Testprotokolls HadConCAN.proto. Das Protokoll ist debug, das im Record in Abschnitt 4.5.2 verwendet wird.

HadConCAN.proto

```
Terminator = LF;
LockTimeout = 4000;
ReplyTimeout = 500;
WriteTimeout = 3000;
ExtraInput = Ignore;
#Connect a stringout record to this to get
# a generic command interface.
#After processing finishes, the record contains the reply.
debug{
            ExtraInput = Ignore;
            out"%s";in"%39c"}
```

Hier wird die Bedeutung der Konfigurationen kurz erläutert.

Terminator

jeder String wird durch ein Line Feed abgeschlossen

LockTimeout

ist ein Integer Wert, der sich auf das out-Kommando bezieht. Es wird maximal 4000ms gewartet, bis die Kommunikation mit dem Device erstellt wird.

ReplyTimeout

ist ein Integer Wert, der sich auf das in-Kommando bezieht. Es wird maximal 500ms gewartet, bis das erste Byte der Antwort vom Device ankommt

WriteTimeout

ist ein Integer Wert, der sich auf das out-Kommando bezieht. Es wird maximal 3000ms gewartet, bis das Kommando gesendet wird.

ExtraInput

Bezieht sich auf das in-Kommando, alle zusätzliche Input Variablen werden ignoriert.

debug

ist das definierte Protokoll, es besteht aus out- und in Kommandos:

out wird benutzt um beliebige Strings an das Device zu senden.

in wird benutzt um beliebige Strings vom Device zu empfangen. Es werden maximal 39 Zeichen vom Device gelesen.

4.5.2 Record

Eine Process Variable wird über eine sogenannte Record-Definition in der Database beschrieben. Ein Record besteht aus einem Record-Typ und einem Record-Namen.

HadConCAN.db

```
Record(stringout, “$(HOSTNAME):DEBUG”){
field(DTYP, “STREAM”)
field(OUT, “@HadConCAN.proto  debug  HadConCAN”)
}
```

Stringout

ist der Record Typ

DTYP

steht für Device Typ und gibt den Namen des Device Typs.

OUT

steht für Output Link und definiert die Art der Kommunikation mit dem STREAM Device. Das Feld besteht aus:

HadConCAN.proto ist der Name für das Protocol File

debug ist das Protokoll im Protocol File

HadConCAN ist der sogenannte Bus-Name, dessen Definition in Abschnitt 4.5.3 erfolgt.

4.5.3 Startup Script für IOC

Das Input Output Control läuft auf der CPU, das IOC wird die Database **dbLoadDatabase** und damit das Record **dbLoadRecord** erzeugt wird. Durch die Records werden die Process Variablen bereitgestellt für eine Verbindung mit der Hardware über einen definierten Bus.

Die Bus Konfiguration wird in **drvAsynSerialPortConfigure** definiert. Der eigentliche Bus ist: dev/ttyS1.

Alle wichtigen Parameter für die serielle Kommunikation u. A. Baudrate, Format sind in **asynSetOption** gesetzt.

```
## Register all support components
dbLoadDatabase("dbd/streamTest.dbd")
streamTest_registerRecordDeviceDriver(pdbbase)

drvAsynSerialPortConfigure("HadConCAN","/dev/ttyS1",0,0,0)
   asynSetOption("HadConCAN",0,"baud","38400")
   asynSetOption("HadConCAN",0,"bits","8")
   asynSetOption("HadConCAN",0,"parity","none")
   asynSetOption("HadConCAN",0,"stop","1")
   asynSetOption("HadConCAN",0,"clocal","Y")
   asynSetOption("HadConCAN",0,"crtscts","N")

## Load record instances
dbLoadRecords("db/HadConCAN.db","HOSTNAME=${HOSTNAME}")
```

5 Test des gesamten Systems

Dieses Kapitel beschreibt die verschiedenen Tests des implementierten Application Programming Interfaces.

Im ersten Schritt wird das API mit dem implementierten CAN_API-Master getestet und im Anschluss darauf mit dem EPICS Client. Für beide Fälle wird als Testgerät mit CAN Interface ein VME Crate-System genutzt, dessen Beschreibung in Abschnitt 5.2 erklärt wird. Dieses Gerät wird gewählt, da mehrere von diesem im HADES verwendet sind.

Alle empfangenen CAN-Nachrichten des VME Crates werden in den Registern des Mikrocontrollers gespeichert. Die Beschreibung der Ergebnisse beschränkt sich auf die Nutzdaten der Messages CAN-Messages.

Die Struktur für Kommandos und Daten sind im Abschnitt 4.1.5 und deren Bedeutung im Abschnitt 4.1.3 erklärt.

5.1 Aufbau im Labor

Die Abbildung 5:1 zeigt den gesamten Aufbau des Systems im Labor. Der Aufbau besteht aus zwei HADControl-Modulen, einem VME Crate, einer Spannungsversorgung, einem Ethernet Switch, zwei Ethernet-Kabeln für den jeweiligen HADControl und letztendlich zwei CAN-Kabeln. Das erste Kabel verbindet die zwei CAN-Controller auf der jeweiligen Platine. Das zweite Kabel erstellt die Verbindung zwischen den Platinen und dem VME Crate.

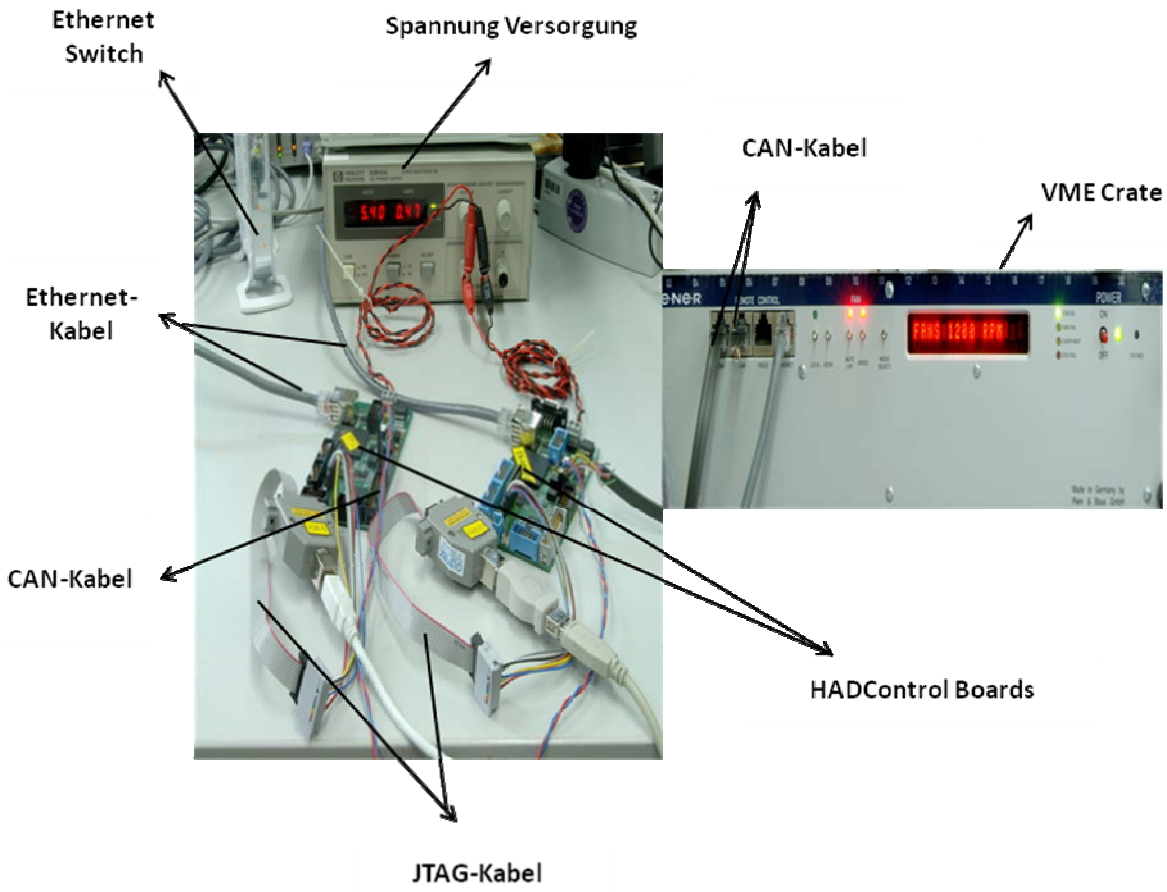

Abbildung 5:1 Der gesamte Aufbau im Labor

5.2 Beschreibung des Testgerätes VME Crate

Versa Modular Eurocard(VME) ist ein Bussystem mit Multiprozessor-Fähigkeiten. Es basiert auf der Architektur von MOTOROLA MC 680x0, das heißt, es arbeitet nach dem Prinzip Master und Slave mit 32Bit Adresse und 32Bit Daten-Bus. Die Kommunikation ist asynchron mit einer Datenübertragung bis zur 80MByte/Sekunde. Das VME stellt heute eines der meist genutzten Industrie-Bus-Standards dar. Für mehr Informationen siehe[14].

Das VME Crate ist ein Gerät mit Hochgeschwindigkeit, an dem bis zum 21 Slot angeschlossen werden können. Das VME Crate besitzt zwei CAN-Bus Remote Control Anschlüsse und ein Lüfter System.

Die Abbildung 5:2 zeigt der Vorderansicht des VME Crates mit einigen Komponenten.

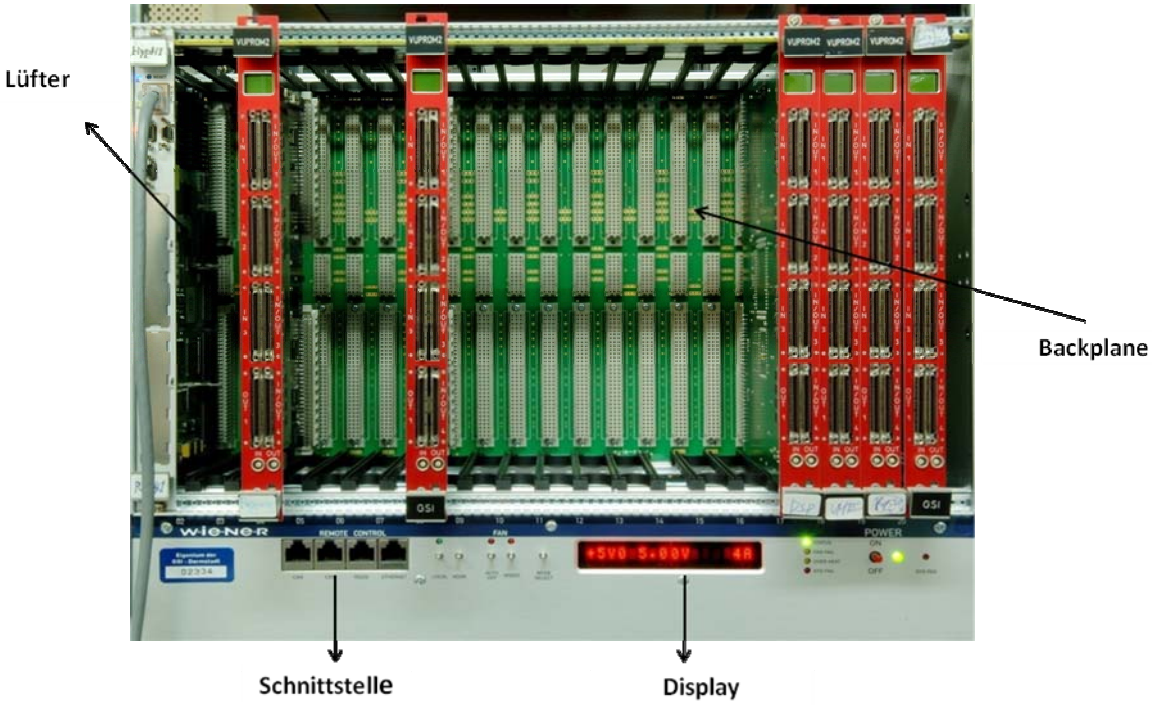

Abbildung 5:2 Vorderansicht des VME Crates

Die Abbildung 5:3 zeigt die verschiedenen Schnittstellen des VME Crates. Es gibt zwei CAN-Schnittstellen, eine davon wird mit dem CAN-Stecker des HADControl Boards verbunden und die zweite mit einem 120Ω Widerstand abgeschlossen werden soll. Dies im Abschnitt 2.1 beschrieben.

Abbildung 5:3 Verschiedene Schnittstellen für VME Crate

Die Abbildung 5:4 zeigt die eingestellte Geschwindigkeit des Lüfters im VME Crate. Auf diesem Display ist es u. A. möglich alle Parameter wie z.B. die Geschwindigkeit, die Temperatur, die Adresse oder die Spannung des Crates anzuzeigen.

Abbildung 5:4 Display für VME Crate

.

5.3 Test mit dem implementierten CAN_API-Master

5.3.1 Testaufbau

Die Abbildung 5:5 zeigt das Blockdiagramm für die Tests mit dem CAN_API-Master.

Der CAN_API-Master sendet die Kommandos über die serielle Schnittstelle zum CAN_API-Slave. Der CAN_API-Slave empfängt diese Kommandos und sendet sie über den CAN-Bus an das VME Crate weiter. Die CAN-Daten aus dem VME Crate werden vom CAN_API-Slaven empfangen und an den CAN_API-Master gesendet. Der CAN_API-Master stellt die Ergebnisse auf der Terminal-Verbindung dar. Der externe Mikrocontroller wird für Testzwecken im Blockdiagramm eingeführt. Dieser externe Mikrocontroller sendet in regelmäßige Abstände verschiedene Nachrichten auf den Bus zu senden. Damit wird die Funktionalität der Filterung in dem Implementierten CAN_API getestet.

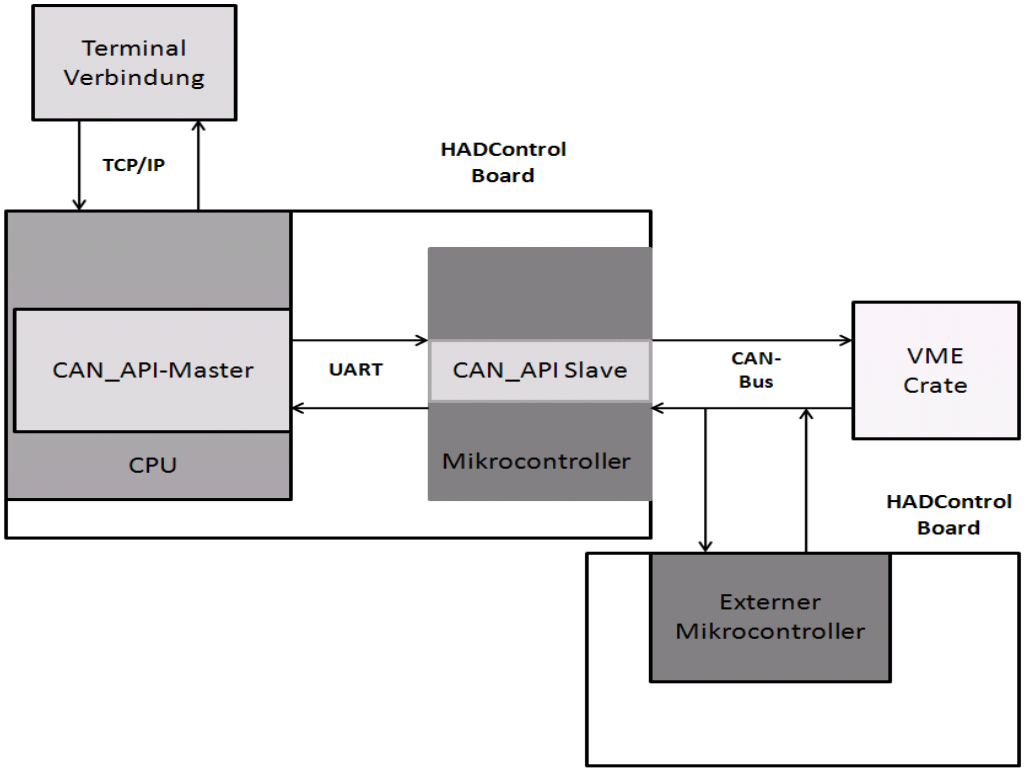

Abbildung 5:5 Das erste Blockdiagramm für den Test des CAN_API-Slaves

5.3.2 Test1: Temperatur des Lüfters auslesen

Die Temperatur des Lüfters im Labor schwankt zwischen 26°C und 35°C. Der Wert -128_D oder 80_H im Register ist nicht für die Temperatur definiert. Der minimale Temperaturwert ist -127 oder 81_H und die maximale 127 oder $7F_H$. Die Tabelle 5:1 zeigt den Aufbau der maximalen 8 Bytes der Temperatur.

Für mehr Information siehe Anhang Seite 9 und 10 ID_Temp-Get Temperatures.

Byte 0							Byte 7
Temp 1	Temp 2	Temp 3	Temp 4	Temp 5	Temp 6	Temp 7	Temp 8

Tabelle 5:1 Beschreibung von 8 Bytes CAN-Daten für Temperatur

Um die Temperatur des Crates auszulesen, wird das Kommando **“SEND 384 0 1 8“** gesendet, als Antwort wird der String **“RECV 0 384 8 18 80 80 80 80 80 80 80 “** empfangen.

Erreichen des Temperaturwertes erreicht:

Temp 1= Temperatur des Lüfters.

Temp 1 = $0x18_H = 24_D$

Die Temp 2 bis Temp 8 sind nicht vorhanden, weil die Werte außer dem erlaubten Bereich liegen.

Die Daten für Temp 2 bis Temp 8 haben den Wert 80_H, da kein weiterer Lüfter angeschlossen ist.

Temperatur des Lüfters = 24°C.

Die Abbildung 5:6 zeigt, wie das Ergebnis auf dem Terminal dargestellt wird.

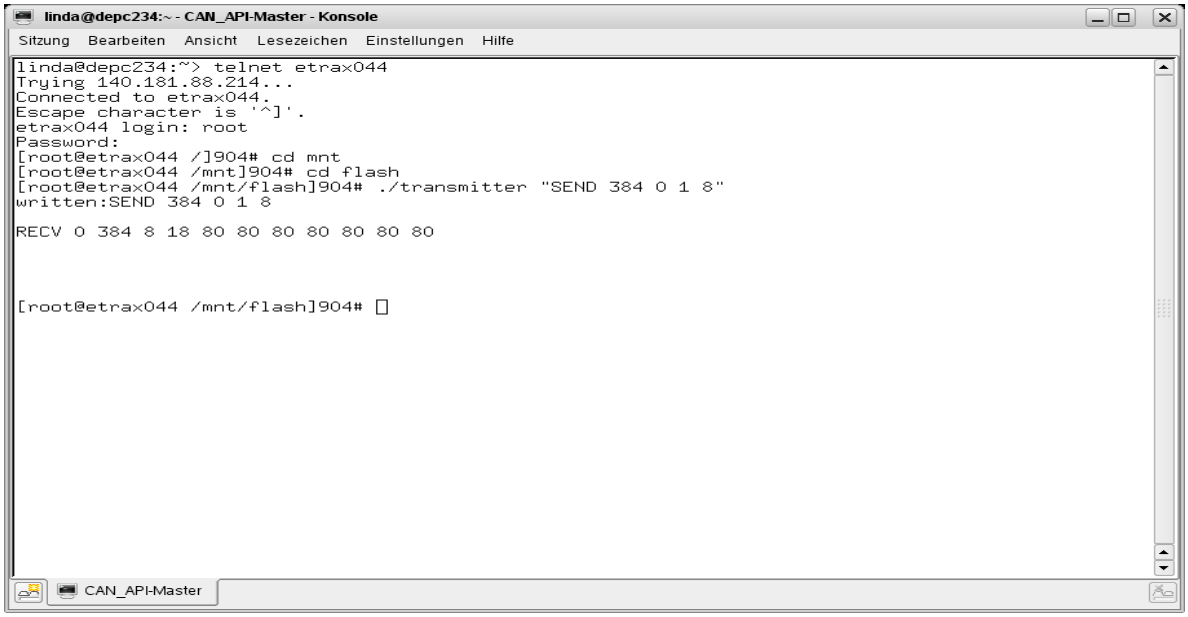

Abbildung 5:6 Darstellung das Ergebnis des Temperatur vom Lüfter

5.3.3 Test2: Speed des Lüfters auslesen

Der Wert 255_D oder FF_H im Register ist nicht für das Speed definiert.

Um das Speed des Lüfters auszulesen, wird das Kommando **“SEND 304 0 1 8 0“** gesendet, als Antwort wird der String **“RECV 0 304 8 8f b0 15 11 14 ff ff ff“** empfangen.

Erreichen des Speedwertes erreicht:

Byte 0							Byte 7
Mittlere fan speed	Nominale fan speed	Fan 1 speed	Fan 2 speed	Fan 3 speed	Fan 4 speed	Fan 5 speed	Fan 6 speed

Tabelle 5:2 Beschreibung von 8 Bytes CAN-Daten für Fan

Byte 0= Mittlere fan speed = (Nominal fan speed)

Byte 1= Nominal fan speed

Byte 2 = fan1 speed

Byte 2 = $0x15_H = 21_D$

Auf dem Display der Abbildung 5:4 ist FANS 1200 RPM abzulesen.

Die Geschwindigkeit des Lüfters hat die Einheit RPM(Revolution per Minute). Um den Speed im RPM umzurechnen, muss der Wert des Byte2 mit 60 multipliziert werden. Der Wert ist die Anzahl der Umdrehungen pro Sekunde.

Speed des Lüfters[RPM] = Byte2*60 = 21_D*60 = 1260 RPM.

Für mehr Information siehe Anhang Seite 9 IDfan-Get Fan Speed.Die Abbildung 5:7 zeigt wie das Ergebnis auf dem Terminal dargestellt wird.

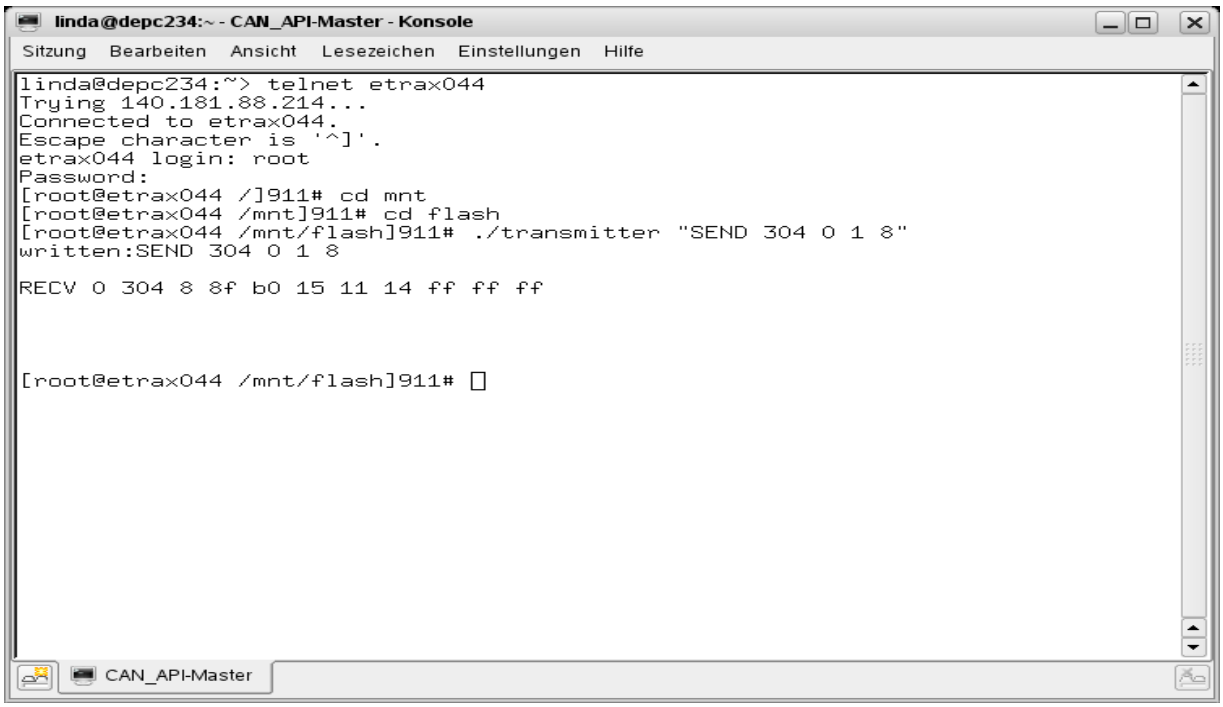

Abbildung 5:7 Darstellung das Ergebnis des Fans Crates

5.3.4 Test2: Speed des Lüfters verändern

Um den Speed des Lüfters zu verändern, wird der String **"SEND 84 0 0 2 80 20"** gesendet, als Antwort wird der String **"RECV Kommando will carried "**empfangen**.**

Diese Antwort bedeutet, dass das Kommando durchgeführt wird. Der Speed wird von 1200 RPM auf 1920 RPM erhöht.

Für mehr Information siehe Anhang Seite 9 IDctrl –Send Control Command.

Die Abbildung 5:8 zeigt, wie das Ergebnis auf dem Terminal dargestellt wird.

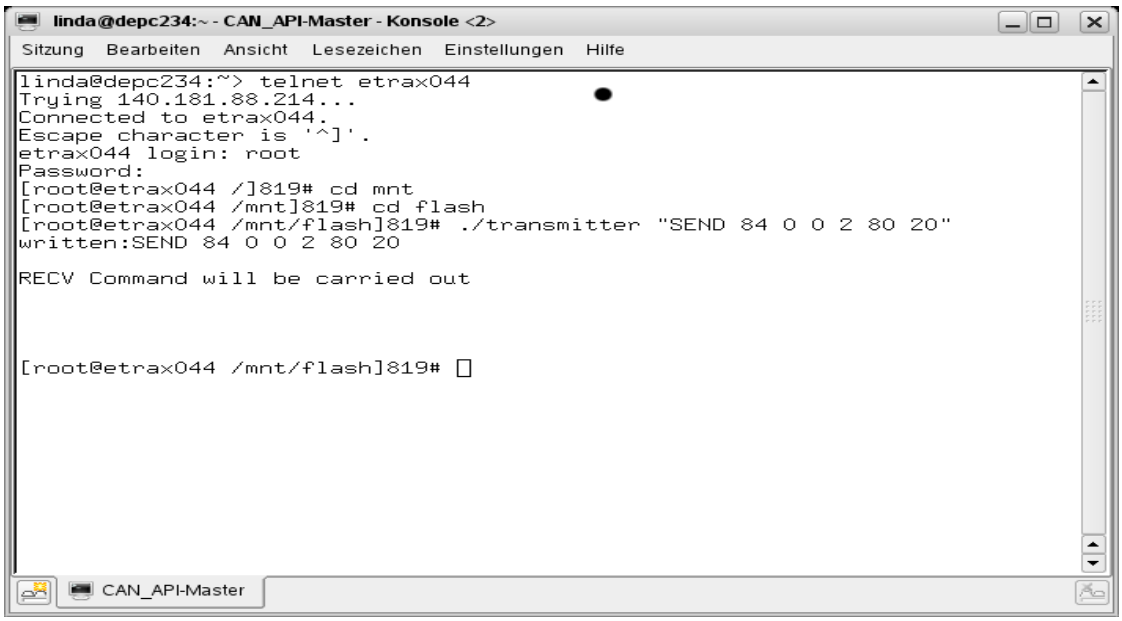

Abbildung 5:8 Darstellung das neue Ergebnis des Fans vom Lüfter

5.3.5 Test 3: Crate ausschalten

Um das Crate auszuschalten wird das Kommando **“SEND 84 0 0 1 1“** gesendet. Als Antwort wird der String **“RECV Kommando will carried “**empfangen**.**

Für mehr Information siehe Anhang Seite 9 IDctrl –Send Control Command.

Die Abbildung 5:9 zeigt, wie das Ergebnis auf dem Terminal dargestellt wird.

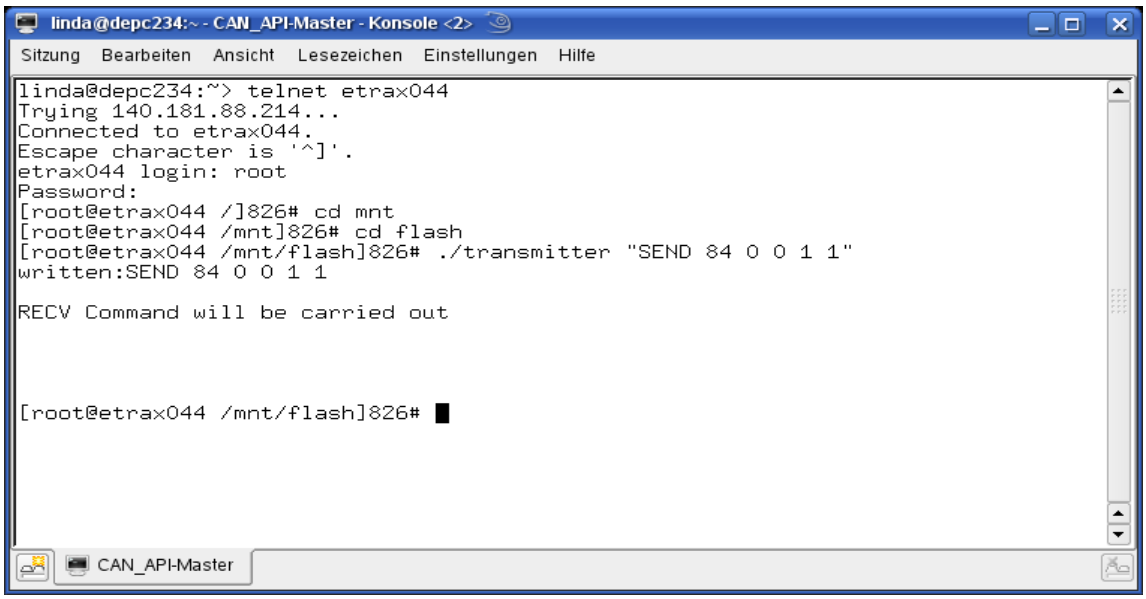

Abbildung 5:9 Darstellung das Ergebnis des Crate -Ausschalten

5.3.6 Test 4: Erkennen eines Eingabefehlers Kommunikation

Wird ein Parameter in der Kommandozeile falsch gesetzt, erkennt der CAN_API-Salve diesen und signalisiert einen Fehler auf dem Terminal.

Beispiel:

Eingabe:**"SANN 304 0 1 8"**

Ausgabe: **"ERRA 0 no valid Command name"**

Die Abbildung 5:10 zeigt, wie das Ergebnis auf dem Terminal dargestellt wird.

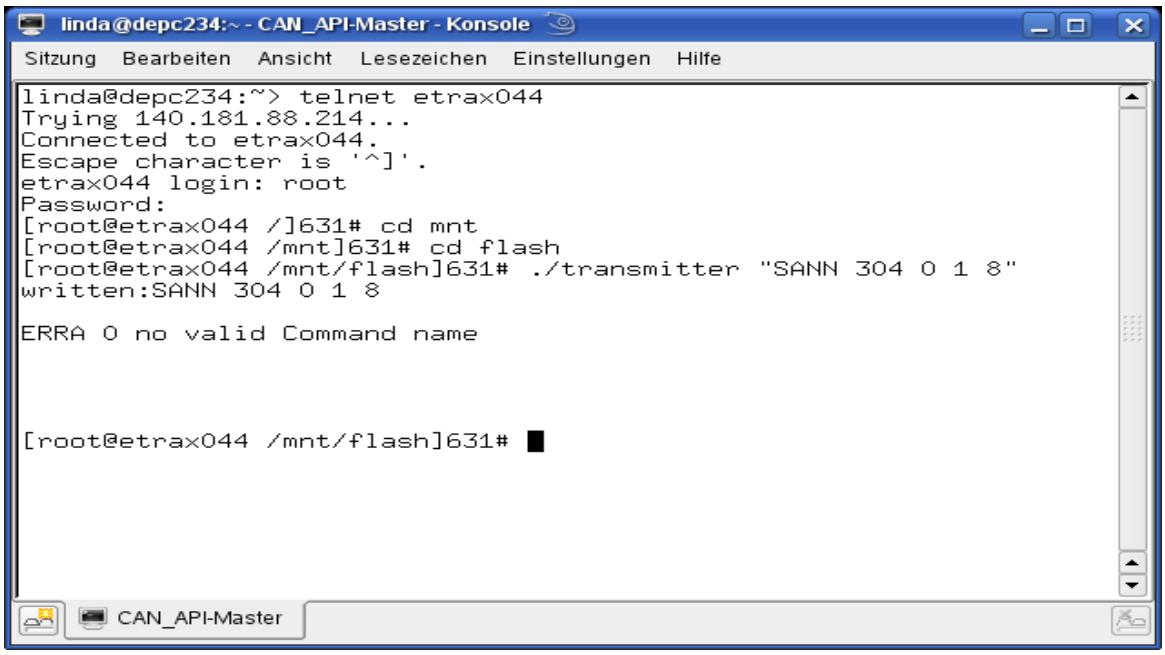

Abbildung 5:10 Darstellung eines Eingabe Fehlers

5.3.7 Test 5: Fehler der CAN-Kommunikation

Während der CAN-Kommunikation können Fehler auftreten, z.B. das CAN-Kabel ist nicht eingesteckt oder die Adresse der Crates ist nicht gültig. Das CAN-Kabel wird absichtlich vom seinem Stecker gezogen, und der String **"SEND 384 0 1 8"** wird gesendet. Als Antwort wird der String **"ERRC 7 Acknowdlegment Error"** empfangen. Diese Antwort signalisiert einen Fehler.

Die Abbildung 5:11 zeigt, wie das Ergebnis auf dem Terminal dargestellt wird.

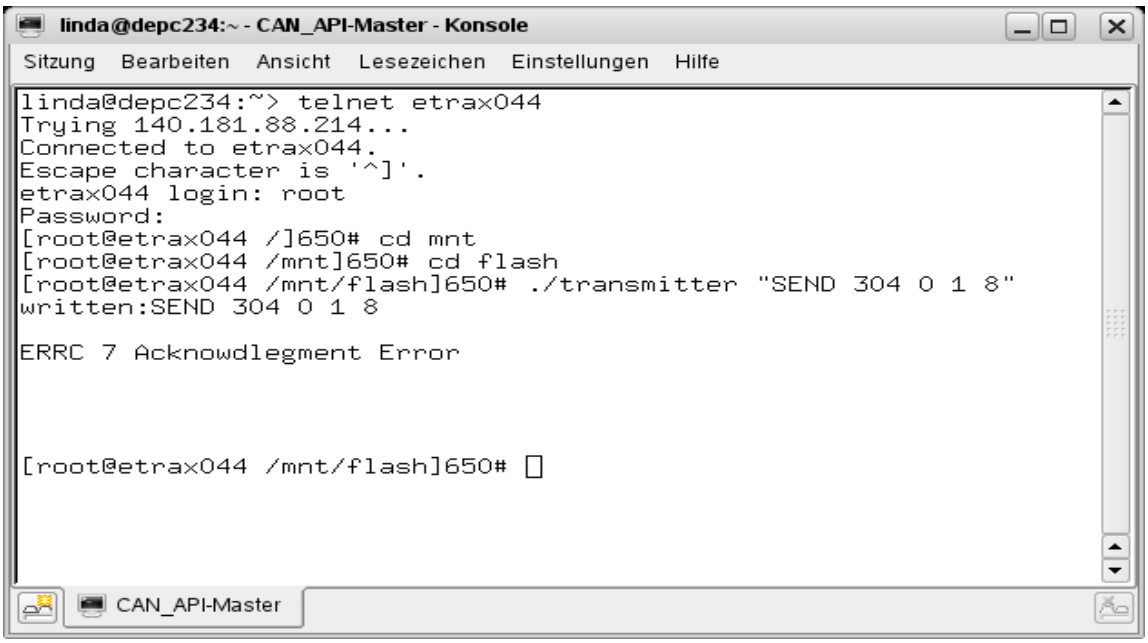

Abbildung 5:11 Darstellung eines Fehlers während der CAN-Kommunikation

5.3.8 Test 6: Empfangen von bestimmten Nachrichten

Um bestimmte Nachrichten aus dem Bus zu empfangen, wird der String **“SUBS 0 700“** gesendet. Der Wert 700_H aus dem String ist die Maske für die CAN-Nachrichten. Mit dieser Maske werden die Nachrichten mit Identifier zwischen 0_H … 80_H empfangen. Die Abbildung 5:12 zeigt, wie das Ergebnis auf dem Terminal dargestellt wird.

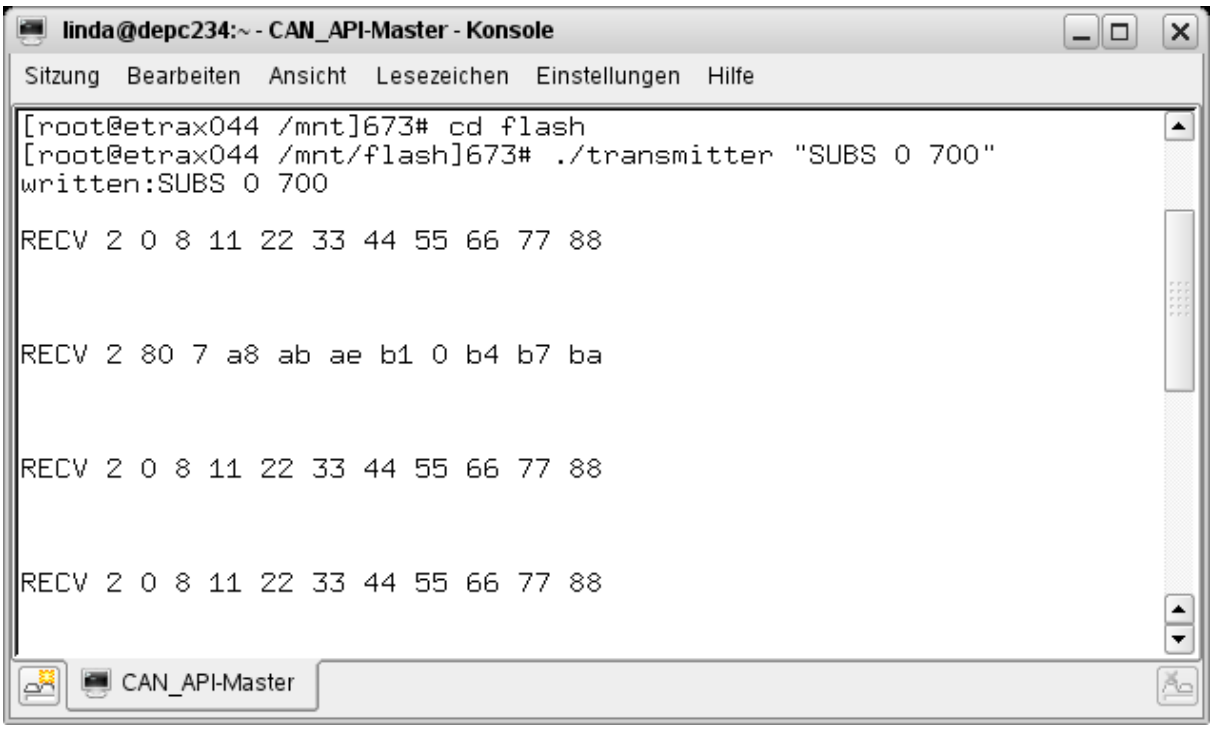

Abbildung 5:12 Darstellung des Ergebnisse für die Anmeldung von bestimmten Nachrichten

5.4 Test mit einem EPICS Client

5.4.1 Testaufbau

Die Abbildung 5:13 zeigt das Blockschaltbild des zweiten Testaufbaus. Die Database des IOC Servers besteht aus verschiedenen Records. Die Process-Variablen werden über diese Records definiert. Der Wert der Process-Variablen wird mit Hilfe des EPICS Device Supports über die serielle Schnittstelle an den CAN_API-Slave gesendet. Der CAN_API Slave sendet den empfangenen Wert über CAN-Bus an das VME Crate weiter. Die empfangenen CAN-Daten aus demVME Crate werden vom CAN_API-Slave an das EPICS Device Support über die serielle Kommunikation gesendet, und die Antwort auf MEDM dargestellt.

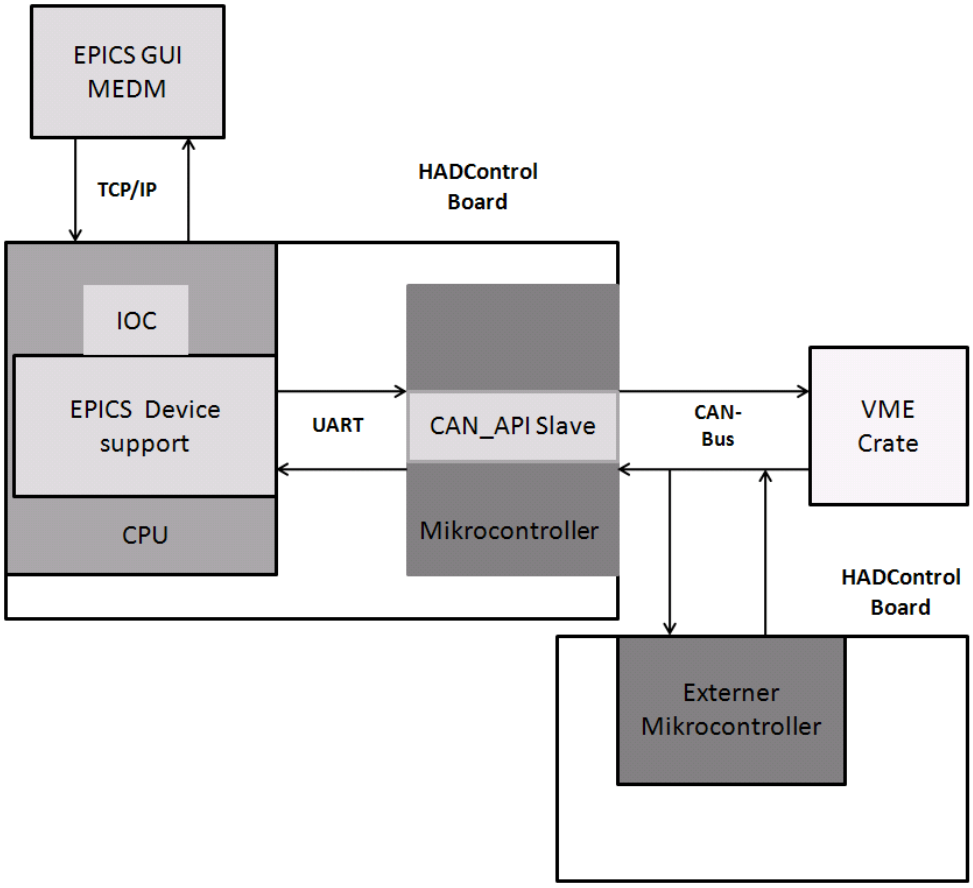

Abbildung 5:13 Das zweite Blockdiagramm für den Test des CAN_API Slaves

Die Abbildung 5:14 zeigt, wie die Sende- und Empfangskommandos auf dem MEDM dargestellt werden. Das obere Feld ist Kommandozeile und das in dem unteren Feld wird die Antwort angezeigt.

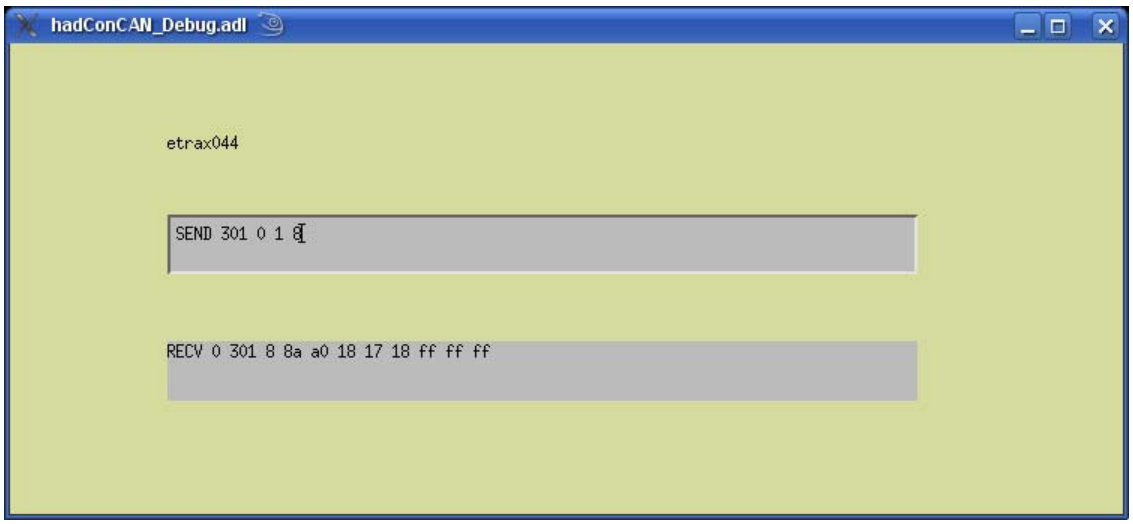

Abbildung 5:14 Zweite Darstellung des Ergebnisses auf EPICS GUI

5.4.2 Interpretation der graphischen Darstellung

Mit Hilfe eines anderen Records im EPICS IOC Server werden die Ergebnisse nicht mehr als String-Format dargestellt, sondern als Graphik. Die Abbildung 5:15 stellt einige Ergebnisse von Temperatur, Speed und Status dar.

In diesem Abschnitt werden kurz die verschiedenen Graphiken beschrieben.

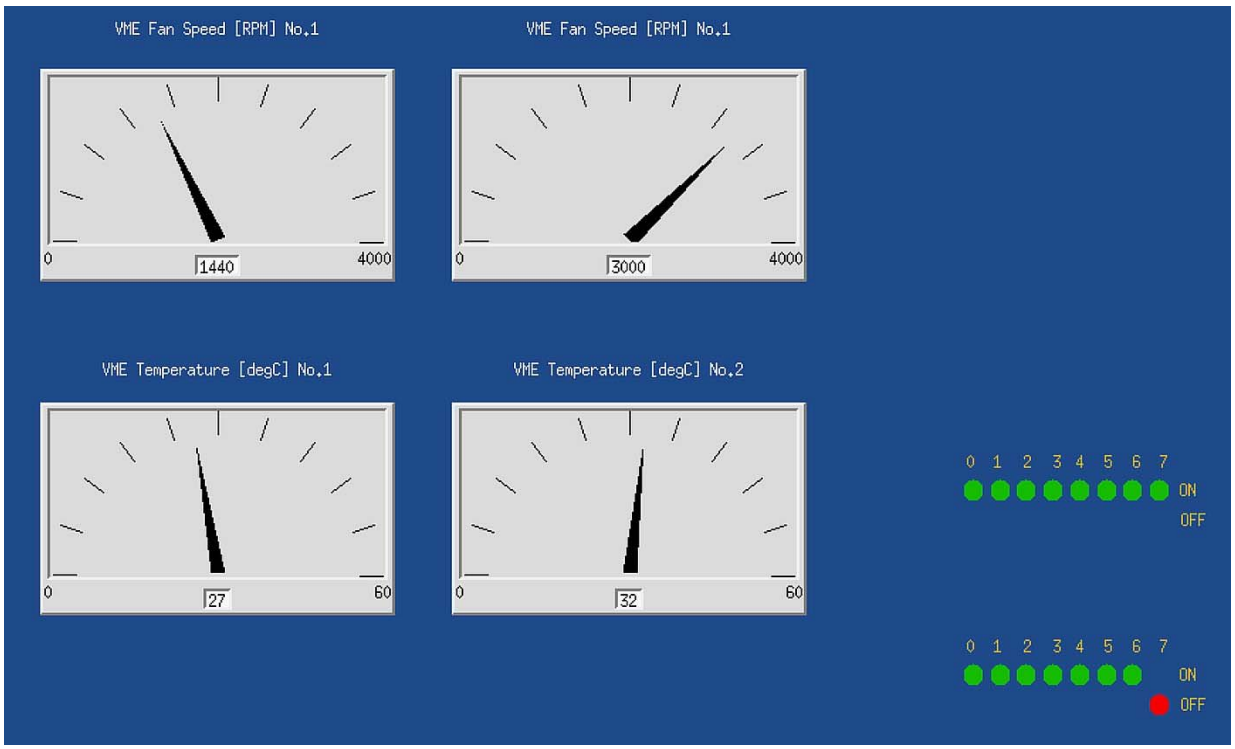

Abbildung 5:15 Zweite Darstellung des Ergebnisses auf EPICS GUI

5.4.2.1 Speed Anzeige

Die Bilder stellen den Speed des Lüfters in Umdrehungen pro Sekunde dar.

Das linke Bild stellt den aktuellen Fan Speed dar. Um das Bild zu bekommen, wird das Kommando **“SEND 304 0 1 8“** gesendet und als Ergebnis wird das Bild dargestellt. Auf dem Bild ist der Wert 1440 abzulesen. Dieser Wert ist der Speed des Lüfters im RPM.

Muss der Speed des Lüfters verändert werden, wird das Kommando **“SEND 84 0 0 2 1 32“** gesendet und als Ergebnis wird das rechte Bild angezeigt. Der Wert 3000 ist abzulesen.

5.4.2.2 Temperatur Anzeige

Die Bilder stellen die Temperatur des Crates dar. Die Temperatur wird regelmäßig ausgelesen. Hierfür wird das Kommando **“SEND 384 0 1 8 0“** gesendet. Die Temperatur wird in Celsius angezeigt.

5.4.2.3 Status Anzeige

Der Crate Status wird durch LED-Anzeige dargestellt.

Um den Status abzufragen, wird das Kommando **“SEND 4 0 1 8“** gesendet und als Ergebnis in der oberen LED-Reihe sind grün alle LED, falls das Crate eingeschaltet ist. Wenn die letzte LED rot leuchtet bedeutet dies, dass das Crate ausgeschaltet ist.

6 Zusammenfassung und Ausblick

Die Aufgabe war die Implementierung eines Application Programming Interfaces im HADControl-Board für die Steuerung von CAN-Geräten mit Hilfe des EPICS Kontrollsystems.

6.1 Erreichte Ergebnisse

Das geforderte System wurde implementiert und erfüllt die Spezifikationen. Es wurde im Labor getestet. Die Tests haben die Stabilität und die geforderte Funktionalität des Systems bewiesen.

Die Abbildung 6:1 stellt das alte System im HADES-Experiment dar. Das Motorola MV162 läuft auf dem VxWorks Operation System. VxWorks ist ein Echtzeitbetriebssystem, das in kleinen Geräten verwendet wird. Maximum 60 Geräte können an dem VME Bussystem angeschlossen werden.

Die Abbildung 6:2 zeigt, wie das neue System im HADES-Experiment in Zukunft eingesetzt wird. Das auf VME basierte Bussystem wird durch das HADControl Board ersetzt, das folgende Vorteile hat: es ist flexibler, mit leicht einsetzbarer Hardware und ist unabhängig von VME. An jedes HADControl Board können bis zu 10 Geräten angeschlossen werden.

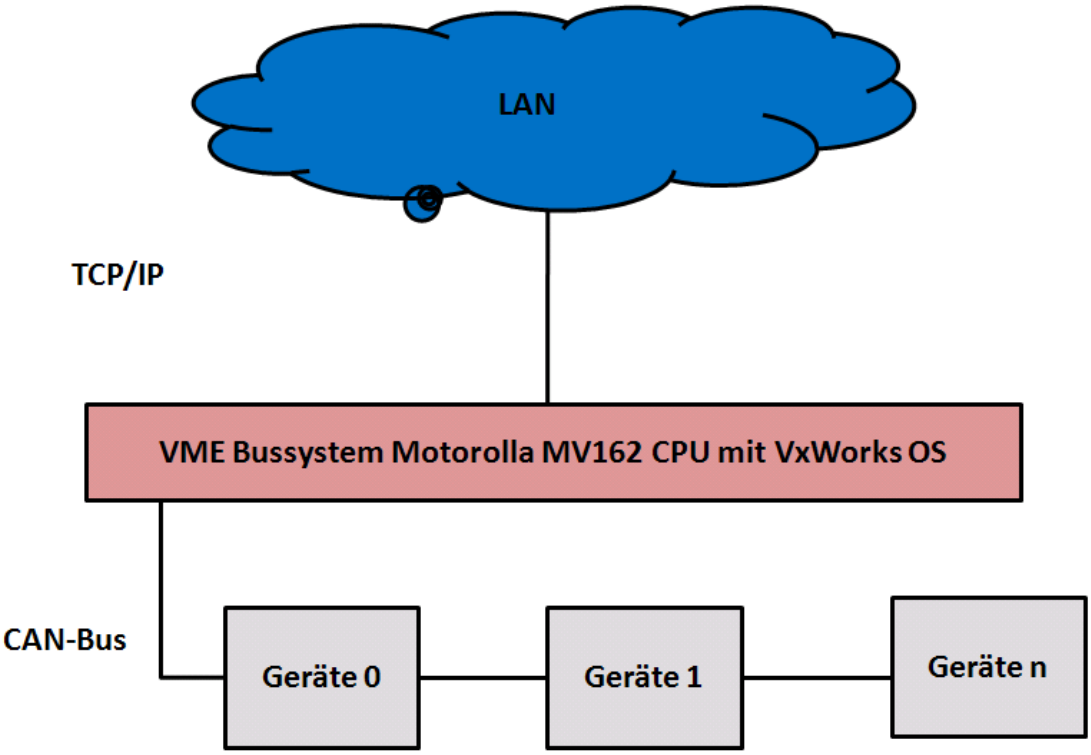

Abbildung 6:1 Aufbau des alten Systems im HADES

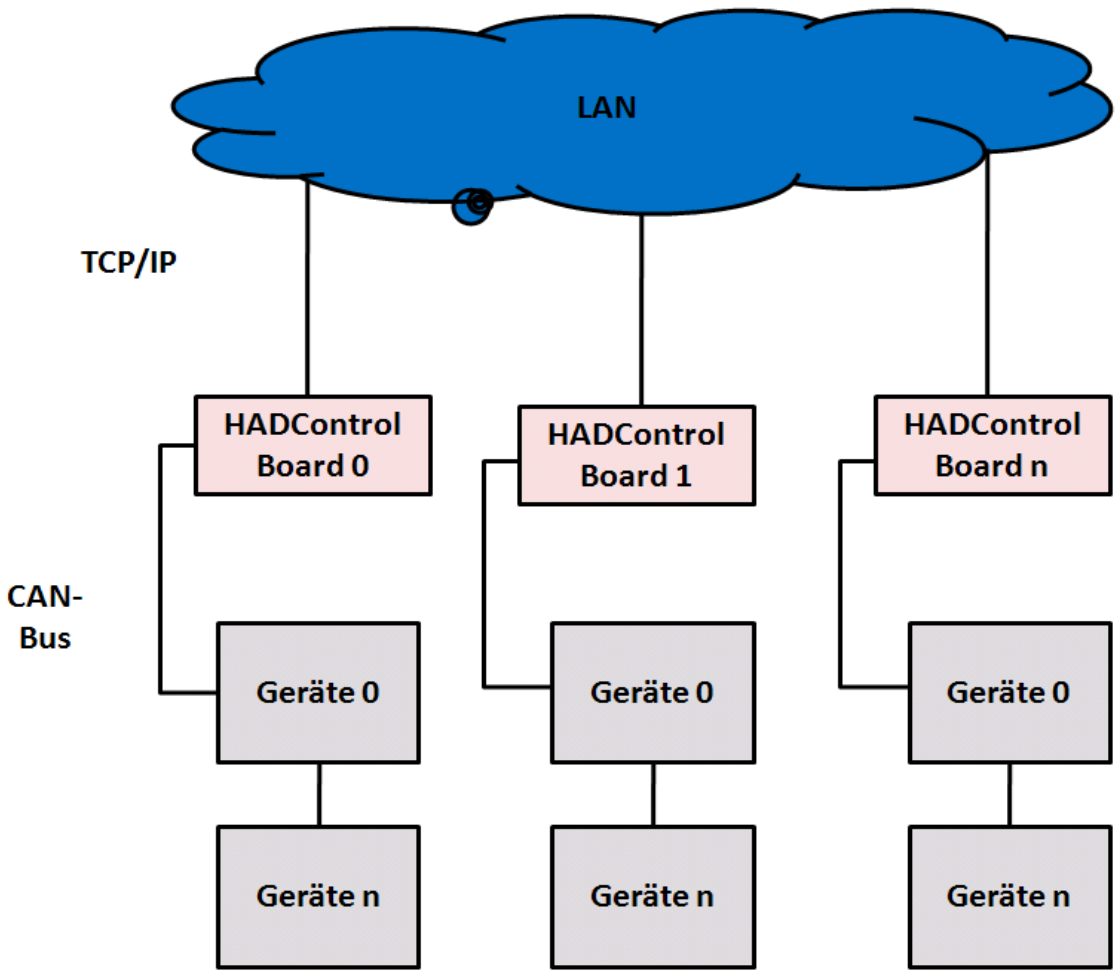

Abbildung 6:2 Aufbau des neuen Systems im HADES

6.2 Ausblick

Das im Rahmen dieser Diplomarbeit entwickelte System ist für weitere, andere Anwendungen einsetzbar.

Mit diesem System können die empfangenen Daten von CAN-Geräten durch eine andere Schnittstelle des Mikrocontrollers, zum Beispiel **I2C,** übertragen und zur Steuerung von weiteren Geräten benutzt werden.

Quellenverzeichnis

[1] http://.www.gsi.de

[2] http://.www.wikipedia.de

[3] http://www.ethernet.de/api/atcan_8c-source.htlm

[4] http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial

[5] http://www.wiener-d.com/support/documentation/CAN-Bus_Protokol.pdf.

[6] http://developper.axis.com/wiki/doku.php?id=axis:compiling_for_cris_howto

[7] http://www.aps.anl.gov/epics/modules/soft/asyn/R4-10/asynDriver.html

[8] http://wiki.gsi.de/cgi-bin/view/Epics/ExampleIOCApplication

[9] http://epics.web.psi.ch/software/streamdevice/

[10] epics.web.psi.ch/training/lectures-series/deutsch_epics_intro_pt2.ppt

[11] http://www.ethernut.de/api/atcan_8c-source.html

[12] Wolfhard Lawrenz: CAN Grundlagen und Praxis. Hüthig Verlag Heidelberg

[13] AT90CAN18 Datenblatt

[14] http://www.wiener-d.com/M/11/8.html

. .

Abbildungsverzeichnis

Seite

Tabellenverzeichnis

Seite

Anhang CAN-Bus_Protocol