DIPLOMARBEIT

P-NET-Slave auf einem 88C166 Mikrocontroller

Implementierung eines P-NET-Knoten

ausgeführt am Institut für Computertechnik
der Technischen Universität Wien

unter Anleitung von o.Univ.-Prof. Dipl.-Ing. Dr. Dietmar Dietrich
und den Betreuern
Dipl.-Ing. Mikhail Gordeev
Univ. Ass. Dipl.-Ing. Thilo Sauter

durch

Markus HAAG
Schloßgegend 71, 3204 Kirchberg
Matr.Nr. 9325058

Wien, Juni 1999 _____

Kurzfassung

Die vor Ihnen liegende Diplomarbeit beschreibt die Implementierung eines P-NET-Slaves auf einem 88C166-Mikrocontroller der 80C166 Familie von Siemens. Dies umfaßte nicht nur die Implementierung des P-NET-Protokolls, sonder auch all jene Arbeiten die rund um das Mikrocontroller nötig waren, um ein vollwertiges P-NET-Modul zu erhalten. Dies beinhaltet die Erstellung von geeigneten Anwendungen, Erstellung einer Interface-Schaltung zum P-NET sowie bis hin zur Spannungsversorgung.
Besonderes Augenmerk wurde auf ein effizientes Programm, auf universelle Einsetzbarkeit und auf eine einfache modulare Struktur gelegt.
Der 88C166-Mikrocontroller besitzt wegen seiner 16 Bit-Architektur, des 32Kbyte Flash EEPROMs, fünf 16-bit Timer/Counter und wegen der Geschwindigkeit von bis zu 40MHz – um nur einige Merkmale zu nennen – als auch wegen der weiten Verbreitung der 80C166-Familie, ideale Voraussetzungen zur Verwendung eines Feldbusknotens.
Der Feldbus P-NET zeichnet sich vor allem durch seine einfache serielle Struktur, der gleichartigen Installation von neuen Knoten und durch ein effizientes Protokoll aus, welches eine Implementierungen von Slaves auf nahezu jeder Mikrocontrollerarchitektur ermöglicht.

Wie in allen Bereichen der prozeßnahen Kommunikation, wird auch im P-Net ein sehr großer Wert auf Echtzeitfähigkeit gelegt – dies wird speziell durch die Master-Slave-Struktur des P-Net erreicht. Und durch die Eigenschaft ein deterministisches System zu sein, kann exakt die Worst Case Situationen berechnet werden.

Abstract

This work, which you see in front you, descripes the implementation of a P-NET-Slave on a 88C166-Microcontroller of the Siemens-product-family.
It deserves mentioning that this process requires ones not only the implementation of the P-NET-Protocol but also all those work which are necessary to receive an adequate P-NET-Modul.
This includes both the making of the equipment for I/O functions, the preparation of an intefacing circuit to the P-NET up to the voltage supply.
Special attention is placed to an efficient program, to the possibility of universial use and to an uncomplicatet modular structure.
The 88C166-Microcontroller has because of its 16Bit architecture, its 32kByte flash EEPROM, five 16bit Timer/Counter an because of its possible speed of 40MHz - just to mention some of its characteristics - and not to forget - because of its great name recognition rating ideal assumptions for its use as an fieldbus node.
The fieldbus P-NET distinguishes itself above all through its simple serial structure and its efficient protocol, which allowes the implementation of slaves on almost every microcontroller-architecture.
As in all sectors of processrelated communication it is set great store by real-time-capability - which is achived through the master-slave-structure of the P-NET.
In addition it is possible through its quality to be a deterministic system to compute the worst-case-situation exactly.

Danksagung

Mein Dank gebührt vor allem Sven Dietz für die Zurverfügungstellung des Mikrocontrollerboards und Dipl.-Ing. Mikhail Gordeev für seine tatkräftige Unterstützung.
Für die kritische Durchsicht des Manuskripts bedanke ich mich bei Martina Streicher.

Inhaltsverzeichnis

Kapitel 1

Einführung

Aufgabe war es, einen P-NET Slave mit einem 88C166-Mikrocontroller-Board zu realisieren. Dies umfaßt ein sehr weites Spektrum, angefangen von der Hardwareimplemtierung von Spannungsversorgung, Kabel, I/O-Beschaltungen bis hin zur eigentlichen Programmierung des P-NET-Protokolls in der Hochsprache C. Also all jene Arbeiten, die notwendig sind, um ein vollwertiges P-NET-Modul zu erstellen.
Aufgrund dieser zahlreichen einzelnen Arbeitsschritte scheint folgende Gliederung der Arbeit als angebracht:
Im ersten Kapitel wird eine kurze Einführung in das Thema Feldbussysteme durchgeführt, sowie ein Vergleich mit anderen Feldbussystemen hergestellt.
Im folgenden Kapitel wird der P-NET-Feldbus vorgestellt, wobei besonders auf die Funktionalitäten von Slaves eingegangen wird.
Der Leser, der an den Eigenschaften von Mastern oder Gateways interessiert ist, sei auf weiterführende Literatur verwiesen [Dat93, Bon95, Die96]. Der P-NET-Standard [Dat93] verwendet die gleiche Gliederung wie das OSI-Schichten-Modell.
Neben dem eigentlichen Standard werden bei wichtigen Themen oder Definitionen zur besseren Verdeutlichung der Kommunikationsabläufe einige diesbezügliche Beispiele angeführt, welche ein auf der Schicht 2 des OSI-Modells, der Sicherungsschicht, übertragenen Datenrahmen darstellen sollen.
Gegen Ende wird die Channelstruktur [Dat92] erklärt. Dabei wird verstärkt auf jenen Channel eingegangen, welche auch im hier besprochenen Knoten vorkommen, den Service-, den Digital-I/O-, und den Analog-In-Channel. Weitere Strukturen bitte ich [Dat92] zu entnehmen.

1.1 Feldbussysteme

In modernen Prozeßleitsystemen wird der *dezentrale Ansatz* [DD98, Rei98] betrieben, also Intelligenz in den Knoten dezentral zur Steuerung, Regelung und Prüfung verwendet. Dies zieht zwangsläufig die Notwendigkeit nach sich, einzelne Komponenten miteinander zu vernetzen bzw. diese über Busse miteinander zu verbinden. Hier kommt der Begriff Feldbus ins Spiel, der einerseits Einsparungen gegenüber dem zentralistischen System bringt und andererseits durch die Verkürzung der Zeitspanne zwischen Messung und Aktorwirkung das Systemverhalten wesentlich verbessert.

1.2 Vergleichende Bewertung des Feldbus P-NET

Kaum eine anderes Gebiet ist so schwer zu vergleichen, wie das der Feldbussysteme. Bedingt durch die unterschiedlichsten Übertragungstechniken, Strukturen und Buszuteilungen ist sogar der Vergleich der Übertragungsraten nicht sehr aussagekräftig, da eine hohe Übertragungsrate bei zu langer Reaktionszeit, welche durch eine zu geringe Nutzdatenrate bewirkt wird, von geringer Bedeutung ist.
Unter Bedachtnahme auf das eben Angeführte, möchte ich die unterschiedlichen Feldbussysteme an Hand von

- Topologie und Buszugriff,
- Übertragungsrate und Leitungslänge,
- Übertragungseffizienz,
- Dezentralisierung,
- und der Adressierungsart

dennoch zu vergleichen versuchen [Die96, MM96, DD98].

1.2.1 Topologie und Buszugriff

Wie in Tabelle 1.2 ersichtlich, verwenden die meisten Bussysteme die *Bus-* oder *Ringstruktur*. An dieser Stelle sei auf die Problematik des Buszugriffs hingewiesen [MM96, Rei98]. Die einfachste Art der Buszugriffssteuerung ist die *Single-Master-Struktur*. Als Beispiele dieser Struktur seien Interbus-S und Profibus-DP angegeben. Diese sind aufgrund der

Tabellarischer Übersicht einiger Bussysteme

	P-NET	*Profibus*	*LON*	*Interbus-S*	*CAN*
Topologie	Bus/phys. Ring	Bus	Bus	Ring	Bus
Medium	RS485	RS485	verschiedene	RS485	frei wählbar
max. Anzahl an Master	32	bis 32	32385	256	30
Teilnehmer	125	bis 32	32385	32	2032
Übertragungsrate	76,8kbit/s	9,6-500kbit/s	10-1000kbit/s	2000kbit/s	bis 1Mbit/s
Segmentlänge	1200m	1,2km bei 93,75kbit/s	1,3km bei 78kbit/s	12800m	40..1000m
Busverwaltung	Virt. Token Passing	Token Passing		festes Zeitfenster	Bitweise Arbitation

Tabelle 1.2: Übersicht einiger Bussysteme

Struktur sehr einfache und schnelle Systeme.
Feldbussysteme mit mehr als einen Master werden unterteilt in die Bussyteme mit CSMA, Token-Passing und Virtual-Token-Passing Technik. LON und CAN verwenden CSMA, Profibus benützt Token-Passing und P-NET gebraucht Virtual-Token-Passing. P-NET und LON besitzen auch noch die Fähigkeit des Routings. Dadurch können Netze gebildet werden, die die Segmentlänge übersteigen.

1.2.2 Übertragungsrate und Leitungslänge

Wie in obiger Tabelle schon ersichtlich, hängen Übertragungsrate und maximale Leitungslänge (ohne Repeater) unweigerlich zusammen. Dies liegt daran, daß die Dämpfung einer Leitung mit der Leitungslänge und auch mit der Frequenz steigt.
Durch die umlaufenden Summenrahmen und der Repeaterfunktion jedes Teilnehmers besitzt Interbus-S Vorteile in der Leitungslänge und der Übertragungsgeschwindigkeit. P-NET und Profibus umgehen, wie schon erwähnt, die Ausbreitungsbeschränkung durch ihre Routing-Fähigkeit.

1.2.3 Übertragungseffizienz

Die Übertragungseffizienz [Rei98, Nie97] gilt als Verhältnis von Nutzdaten zu den insgesamt übertragenen Daten. Welche besonders bei Interbus-S durch den speziellen Telegrammaufbau ins Gespräch kommt.
Die maximale Effizienz folgt aus einer Transaktion mit maximalen Anzahl von Datenbits. Die minimale Effizienz ergibt sich bei der Übertragung nur eines Datenbits. Tabelle 1.4 [1] gibt den Vergleich einiger Feldbussysteme hinsichtlich der Übertragungseffizienz in Prozent an.

Bussysteme	*Max. [%]*	*Min. [%]*
Profibus	70	0.6
P-NET	67	0.2
CAN	59	0.7
LON	97	0.6
Interbus-S	120	0.01

Tabelle 1.4: Grenzwerte der Übertragungseffizienz [Rei98]

1.2.4 Dezentralisierung

Dezentralisierung ist einer der größten Vorteile von Feldbusystemen gegenüber normalen, zentralisierten Systemen.
CAN und LON sind voll dezentralisiert, d.h. der Feldbus macht keine Unterscheidungen zwischen Master und Slaves. Danach folgt P-NET, der seine Dezentralisierung durch die Programmierbarkeit der Slaves bezieht. Bei Profibus-FMS sind ebenfalls Slaves programmierbar.

1.2.5 Adressierungsart

Die oben vorgestellten Bussysteme verwenden zum Teil sehr unterschiedliche Methoden der Adressierung.
P-NET verwendet ein Byte zur Adressierung, von welchem wiederum nur die ersten 7-Bit die Adresse bilden. Daraus wiederum folgt eine maximale Anzahl an 128 Adressen von denen effektiv nur 125 zu verwenden sind. Mehr dazu findet der Leser im folgenden Kapitel.

[1] 120% bei Interbus entstehen dadurch, daß der Summenrahmen gleichzeitig für Ein- als auch Ausgabe genutzt werden kann.

Im Profibus-FMS wird ebenfalls ein Byte zur Adressierung verwendet.
Im Interbus-S hat jeder Knoten eine eigene Identifikationsnummer, die sich aus der physikalischen Position im Ring ergibt.
CAN benützt eine Objektorientierte Adressierung, die maximal 2032 Objekte unterscheidet.

Kapitel 2

P-NET

Im P-NET-Standard [Dat93] sind folgend drei Basis-Typen erklärt:

- Slaves
- Master
- und Gateways

Slaves

Die hauptsächliche Funktionen eines Salves besteht darin, Anfragen eines Masters zu beantworten. Da hierbei eine maximale Verzögerungszeit von 390μs einzuhalten ist, bedeutet dies, daß ein Slave nicht auf mehrer Masteranfragen antwortet. Dies wiederum bewirkt, daß Befehls-Warteschlangen unnötig werden. Jedoch ist zu beachten, daß ein Slave nicht von sich selbst aus, sondern nur auf die Anfrage eines Masters senden darf. Anfragen beziehen sich immer auf die sogenannten Softwire-Liste, welche eine Liste aus globalen Variablen im Speicher des Slaves ist.

Master

Master haben hingegen als einzige Teilnehmer im P-NET das Recht, von sich aus auf den Bus zuzugreifen. Der Buszugriff im Multimastersystem P-NET wird durch das virtual Token-Passing geregelt, worauf später noch näher eingegangen werden wird.
Da ein Master meist rechenintensiven Aufgaben wie Steuer- und Regelungstechnikaufgaben löst, haben diese aus hochwertiger Rechnerarchitektur zu bestehen. Um dies zu vereinfachen, wird im P-NET insofern die Idee der Aufgabenteilung betrieben, als der Master “schwierigere“ Aufgaben löst und der Slave für “leichtere“ zuständig ist.

Gateways oder Multiport-Master

Die Funktion des Gateways besteht einerseits darin, zwei P-NET-Busse miteinander zu verbinden, wodurch es möglich wird ein komplettes P-NET-System in einzelne Abschnitte zu unterteilen, sowie andererseits durch Redundanz die Fehlertoleranz zu erhöhen.

2.1 Der P-NET-Standard nach dem OSI-Modell

2.1.1 Das OSI-Referenzmodell

Folgende Aufteilung in einzelne Schichten wird wie im OSI-Modell (Open System Interconnection, [Die96, Rei98]) durchgeführt. Im P-NET-Standard [Dat93] sind nur die Schichten eins bis vier und die Schicht sieben vorhanden.
Zu beachten ist, daß die Transportschicht auch manchmal Service-Schicht genannt wird, da sich in dieser Schicht das P-NET Service und das Programm-Service befinden (vergleiche Abbildung 2.1).

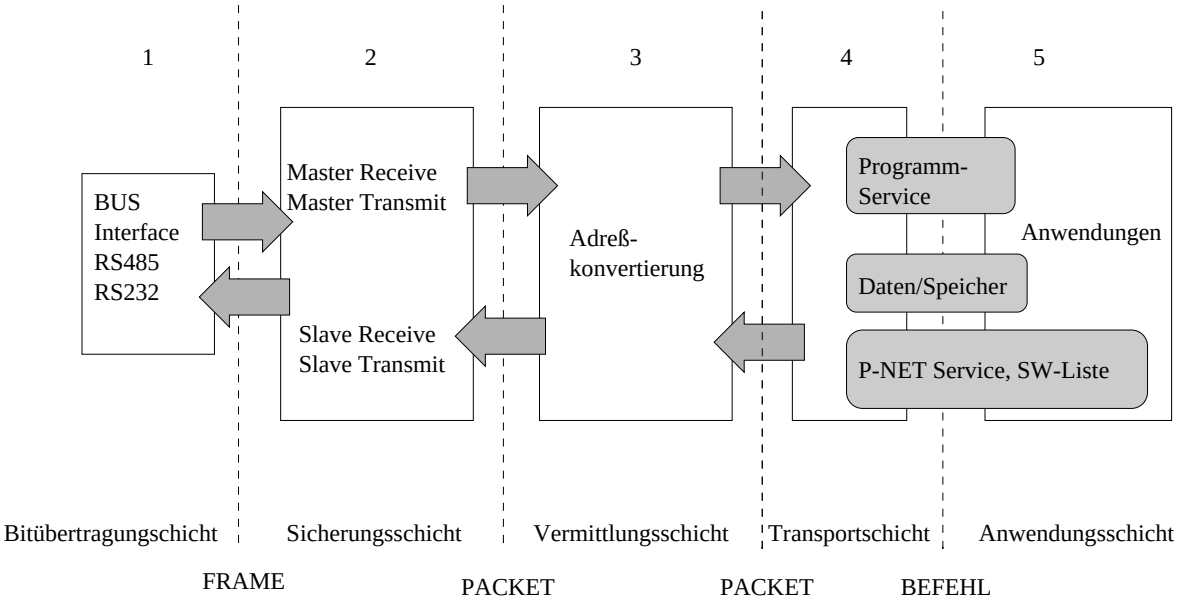


Abbildung 2.1: P-NET Architektur

2.1.2 Schicht 1: Bitübertragungsschicht

P-NET verwendet den EIA RS485 Standard, wobei der Bus jedoch nicht mit zwei Abschlußwiderständen abgeschlossen wird, sondern die Enden zu einem physikalischen Ring verbunden werden. Durch die Verwendung von Adreßbytes, bei denen nur sieben Bits zur Adreßdekodierung benützt werden, wird eine max. Knotenanzahl von 125 erreicht. Die Übertragungsgeschwindigkeit beträgt hierbei 76800 Bit/s, mit einer Toleranz von +/-0,2%. Die Übertragung wird in NRZ-Kodierung (Non Return to Zero) durchgeführt. Wie im RS485-Standard [Sch96] festgelegt, verwendet auch das P-NET Differenzsignale, wodurch auch eine die maximale Ausdehnung von 1200m möglich wird.
Folgende Kodierung wird verwendet:

- binäre 1: A-Signal ist negativ gegenüber B-Signals
- binäre 0: A-Signal ist positiv gegenüber B-Signals

Eine binäre 0 liegt in einem Spannungsbereich von 1.5V $\leq U_{AB} \leq$ 5V am Sender und eine binäre 1 liegt im Bereich von -5V $\leq U_{AB} \leq$ -1.5V.

2.1.3 Schicht 2: Sicherungsschicht

Die Sicherungsschicht hat folgende Aufgaben:

- *Buszugriffskontrolle*, die auch die Regelung bezüglich des Multi-Master-Zugriffes beinhaltet (siehe auch [Dat93, Bon95])
- das *Erzeugen und Erkennen von Rahmen und der Knotenadresse*
- *Fehlerkontrolle*

2.1.3.1 Buszugriff von Slaves

Slaves ist es nur erlaubt, zwischen der 11ten und 30ten Bitperioden nach einer Anfrage eines Masters auf den Bus zuzugreifen (siehe Abb. 2.2). Das bedeutet, daß ein Slave mit möglichst geringer Verzögerung – im Zeitfenster von 143μs bis 390μs – mit der Antwort zu beginnen hat.

Daraus folgt wiederum, daß ein Slave so programmiert werden muß, daß es ihm möglich ist, innerhalb dieser minimaler Verzögerungszeit zu antworten.

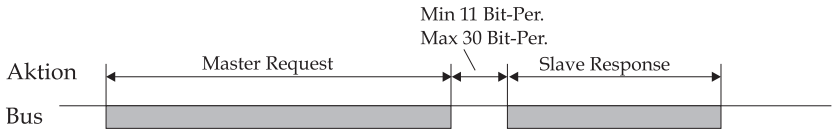


Abbildung 2.2: Slave-Buszugriff

2.1.3.2 Erzeugen und Erkennen von Rahmen

Jeglicher Austausch von Information zwischen den einzlnen Teilnehmern, sprich Kommmunikation zwischen den Knoten, wird über ein einheitliches Rahmenformat durchgeführt (Abb. 2.3).

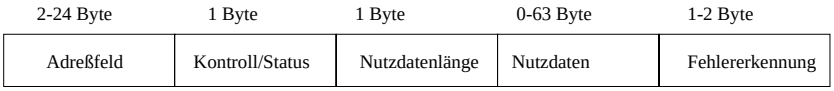


Abbildung 2.3: Diese Abbildung zeigt das Format eines Rahmen.

Die Bytes des Rahmen werden im asynchronen Mode mit einen Startbit, 8 Datenbits, einem Adresse bzw. Datenbit und einem Stopbit (siehe Abb 2.4) übertragen. Während der Übermittlung eines Rahmens dürfen keine Bitpausen auftreten.

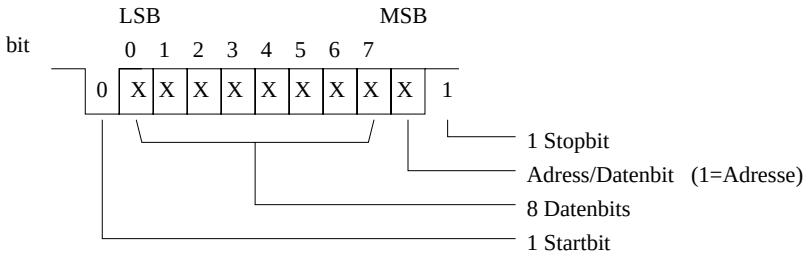


Abbildung 2.4: Byte in einem Frame

Das Adreßfeld

Jeder Frame beinhaltet das Adreßfeld, welches ein Länge von 2 bis 24 Adreßbytes haben kann. Das erste Adreßbyte beinhaltet immer die Zieladresse im lokalen Bus. Die Länge des Adreßfeldes ist direkt in das Adreßfeld selbst kodiert, wodurch ein zusätzliches Byte für die Adreßfeldlänge eingespart wird. Grundsätzlich wird immer das Bit-7 eines Adreßbytes verwendet, um zu unterscheiden, um welchen Adreßfeldtypen es sich handelt oder ob ein Request oder ein Response vorliegt. Die restlichen Bits 0 bis 6 beinhalten die eigentliche Adresse im P-NET. Daraus folgt nun auch die höchstmögliche Anzahl an

Knotenadressen: $2^7 = 127$ Adressen.
Die Länge des Andreßfeldes hängt von der Länge des Weges durch den P-NET ab, denn es wird die komplette Route durch das P-NET im Adreßfeld gespeichert. Auf dem Weg durch den Feldbus wird das Adreßfeld nun so modifiziert, daß die jeweils erreichte Zieladresse in eine Qelladresse umgewandelt wird. Dadurch kehrt sich das gesamte Adreßfeld um, wodurch sich der gesamte Weg bis zur Quelle zurückverfolgen läßt.
Einfacher Adreßtyp:

Der einfache Adreßtyp enthält eine Zieladresse und eine Quelladresse. Er wird für Anforderungen (Requests) an einen Slave verwendet. Wie in Abbildung 2.5 zu erkennen, wird der einfache Adreßtyp mit einer null-eins Kombination des siebenten Bits eingeleitet.

0	Zieladresse
1	Quelladresse

Abbildung 2.5: Einfacher Adreßtyp

Komplexer Adreßtyp:

Der komplexe Adreßtyp wird für zwei oder mehr Quell- und Zieladressen verwendet, wobei im dritten Adreßfeld, dem Extraadreßfeld, die Anzahl der noch folgenden Adreßfelder angegeben wird. Auch der komplexe Adreßtyp wird nur in Request-Frames von einem Master zum Slave verwendet.

0	Zieladresse
0	Zieladresse
0	Extraadresse
0	Zieladresse
1	Quelladresse

1	Quelladresse

Abbildung 2.6: Komplexer Adreßtyp

Erweiterter Adreßtyp:

Das erweiterte Adreßfeld beinhaltet zwei Ziel- und zwei Quelladressen und stellt somit eigentlich nur einen Spezialfall des komplexen Adresstyps dar.
Das erweiterte Adreßfeld wird ebenfalls – wie die obigen Adreßfeldtypen – nur für Anforderungen (Requests) verwendet.

0	Zieladresse
0	Zieladresse
1	Quelladresse
1	Quelladresse

Abbildung 2.7: Erweiterter Adreßtyp

Antwort-Adreßtyp:

Ein Antwort-Frame wird dadurch gekennzeichnet, daß im Adreßfeld das Bit 7 eine Eins-Null-Kombination (vrgl. Abbildung 2.8) enthält. Es ist jedoch zu beachten, daß der Antwort-Adreßtyp insgesamt nie mehr als eine Zieladresse und eine Quelladresse enthalten kann.

1	Zieladresse
0	Quelladresse

Abbildung 2.8: Response-Adreßtyp

2.1.3.3 Fehlererkennung

Es werden zwei unterschiedliche Methoden der Fehlererkennung verwendet — die *normale* und die *reduzierte* Fehlererkennung.

Normale Fehlererkennung

Generierung:

Die normale Fehlererkennug verwendet zwei Bytes zur Erkennung von Fehlern. Jedes zu sendende Byte wird mit zwei Register exklusive ODER verknüpft, zusätzlich wird das zweite Register noch um ein Bit nach links rotiert. Nach dem Info-Feld wird der Inhalt des ersten Registers dem folgend der des rotierten Registers geschickt.

Fehlererkennung:

Die empfangenen Bytes werden wiederum mit zwei Register exklusive ODER verknüpft, das zweite wird auch wieder um ein Bit nach links rotiert. Nach dem Empfang vom ersten Kontroll-Byte muß das erste Register Null sein – nach Erhalt des zweiten Kontroll-Bytes hat auch das zweite Register Null zu enthalten, da ansonsten die Antwort "Error detect failure" zurückgeschickt wird.

Reduziert Fehlererkennung

Generierung:

Bei der reduzierten Fehlererkennung wird jedes Byte zu einem Wert addiert (ohne Übertrag), welcher dann als Zweierkomplement nach dem Info-Feld übertragen wird.

Fehlererkennung:

Das empfangene Error-Byte wird wiederum mit der, aus den übertragenen Bytes gebildeten, Summe addiert. Falls diese nicht Null ergibt wird, auch hier mit obiger Antwort der Frame quittiert.

2.1.4 Schicht 3: Vermittlungsschicht

Wie der Name schon schließen läßt, dient diese Schicht zur Vermittlung und zum Aufbau einer Übertragung über den gesamten P-NET-Bus.
Die Vermittlungsschicht hat in Slaves im Gegensatz zu Dualport- oder Multiport-Master, bei denen sie die Aufgabe des Routings erledigt, eine sehr geringe Bedeutung. Da Routing bei Slaves nicht nötig ist, dient diese Schicht nur der Weiterleitung der Datenpakete zwischen der Sicherungsschicht und der Transportschicht sowie der Erstellung der Antwort "Answer Comes Later".

2.1.5 Schicht 4: Transportschicht oder Service-Schicht

Die Transportschicht dient zur Verwaltung der *Softwirelist*. Diese Tabelle verweist von globalen Variablen wie Knotenadresse, Anzahl der höchsten Softwirenummer, Meßwert, Fehlercodes usw. auf die physikalische Adresse im eigenen Knoten oder aber auf einen externen Knoten. Wenn somit die Anwendungsschicht auf eine Softwirenummer zugreift, wird in dieser Schicht entweder die interne physikalische Adresse ermittelt oder auf einen

externen Knoten verwiesen, d.h. daß bei einer externen Variable die Softwirelist lediglich die betreffende Knotenadresse beinhalten.

2.1.6 Das Paket-Format

Die physikalischen Schicht stellt der Sicherungsschicht die Daten in Form von einzelnen Bytes zur Verfügung, welche von der Sicherungsschicht (Data-Link-Layer) in Rahmen strukturiert werden. Diese Rahmen werden in Pakete konvertiert, welche wiederum in den Schichten Vermittlung und Transport verwendet werden.
Das Format des Paket-Format unterscheidet sich nur marginal vom Rahmenformat: lediglich das Adreßfeld ist unterschiedlich und die Fehlererkennung wird weggelassen. Dem Adreßfeld und dem Info-Feld sind fixe Längen zugewiesen und dem Paket erhält außerdem ein Retry-Timer.

- 25 Bytes - Adreßfeld
- 1 Byte - Kontrol/Status
- 1 Byte - Info-Länge
- 63 Bytes - Info-Feld
- 4 Bytes - Retry-Timer

2.1.6.1 Adreßfeld

Das erste Byte des Adreßfeldes beinhaltet die Länge des Feldes. Unmittelbar danach folgen die Zieladressen und die Quelladressen. Dies unterscheidet sich insofern gegenüber dem Rahmenformat, welches in der Sicherungsschicht verwendet wird, als dort keine Längenangabe verwendet wird, aber mittles Bit 7 zwischen den einzelnen Adreßtypen unterschieden wird.

2.1.6.2 Kontroll/Status-Feld

Dieses Feld ist sowohl im Rahmen- als auch im Paket-Format gleich und hat beim Senden eines Datenpakets vom Master zum Slave die Bedeutung eines Befehls (Control). Bei der anschließenden Antwort des Slaves beinhaltet dieses Feld Statusinformationen, welche entweder einen Fehlercode angeben oder die Anfrage bestätigen.
Abbildung 2.9 und 2.10 zeigen die unterschiedlichen Bedeutungen des Kontroll/Status-Feldes.

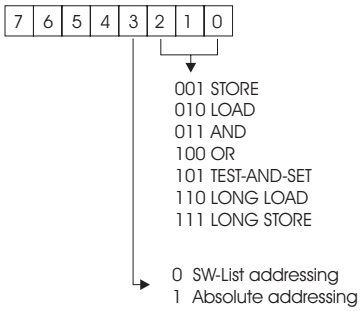


Abbildung 2.9: Befehle

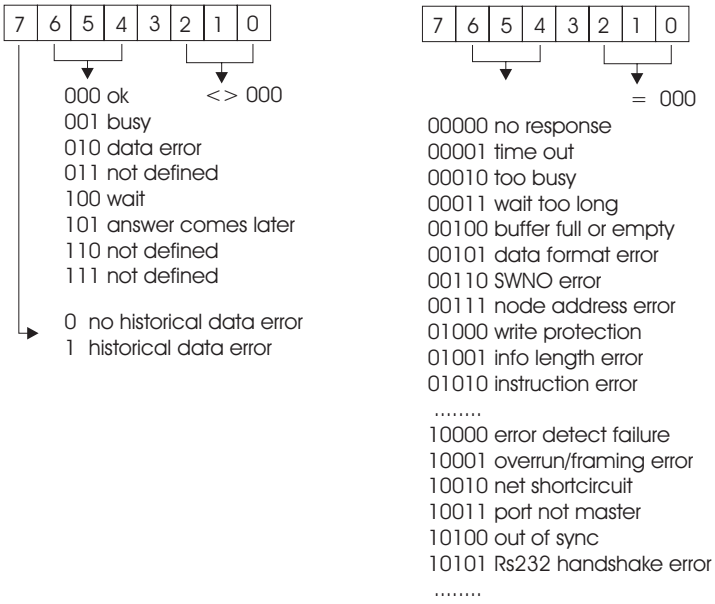


Abbildung 2.10: Kodierung im Kontroll/Status-Feld

Befehle

Wie in Abbildung 2.9 gezeigt, werden die letzten drei Bits zur Befehlskodierung verwendet. Beim Response von Slaves werden wie in Abbildung 2.10 alle acht Bits zur Statusmeldung benutzt.
Im P-NET-Standard werden fünf *Befehle* definiert, zwei Basisdienste LOAD und STORE sowie fünf Zusatzdienste AND, OR, TEST-AND-SET, LONG-LOAD und LONG-STORE.

Im P-NET wird also ein sogenanntes *Reduced Instruction Set* verwendet, wodurch die Effizienz bzw. die Abarbeitung von Befehlen in Slaves erhöht wird.

STORE:

Der STORE-Befehl dient zum Datentransfer vom Master zum Slave, wobei sich die Daten unmittelbar hinter den Softwirenummern im Infofeld befinden. Im Slave werden die Daten direkt in die betreffende Variable geschrieben und ein Responsefeld ohne Daten bestätigt den empfangenen STORE-Befehl.

LOAD:

Mit dem LOAD-Befehl wird der Datentransfer vom Slave zum Master realisiert. Der Master schickt mit ein "Load-Befehl-Paket" die gewünschte Softwire-Nummer, welche im Datenfeld unmittelbar vor der gewünschten Anzahl der Bytes angegeben wird, zum Slave. Dieser schickt dann mit einem Response-Paket, das nur die Daten enthält (keine Softwire-Nummer) zurück.

AND:

Der AND-Befehl führt, wie der Name schon sagt, eine UND-Verknüpfung der gewünschten Softwirenummer durch. Dies kommt also einem Datentransport vom Master zum Slave gleich.

OR:

Die Abhandlung des OR-Befehles ist äquivalent dem AND-Befehl bis auf die ODER-Verknüpfung mit dem gewünschten Wert.

TEST-AND-SET:

Bei TEST-AND-SET wird ein Transport der Daten vom Master zum Slave und zurück durchgeführt.

LONG-LOAD und LONG-STORE:

Diese beiden Befehle werden dann verwendet, wenn die Länge der Daten die maximale Länge der in einem Paket zu übertragenen Bytes, nämlich 54, übersteigt.

2.1.6.3 Info-Länge

Das Format der Info-Länge ist im Rahmen- und Paket-Format gleich, d.h. in der Sicherungsschicht und in der Transportschicht. Es wird in Abbildung 2.11 dargestellt.

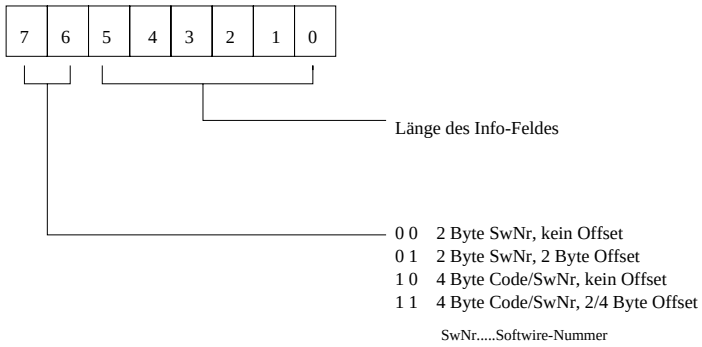


Abbildung 2.11: Info-Länge

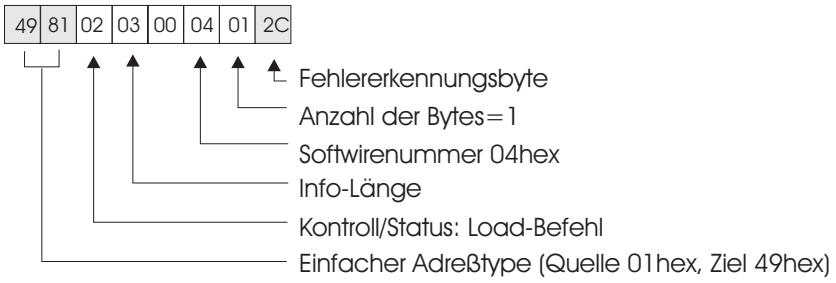
2.1.6.4 Info-Feld

Das Info-Feld kann folgende Information beinhalten:

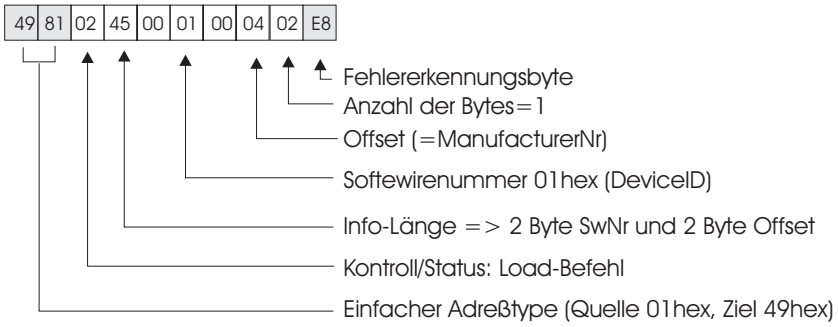
- Softwire-Nummern
- Offset
- Anzahl der zu ladenden Bytes
- Daten
- LONG-Code (bei "Long"-Befehlen)

2.1.7 Beispiele von Rahmen

Um die Bedeutung der einzelnen Felder des Paket- und des Rahmen-Formats besser zeigen zu können, werden einige Beispiele von auf Schicht-2-Rahmen analysiert. (Alle angegebenen Werte sind im hexadezimalen Format angegeben.)



Beispiel 2.1: Load-Befehl



Beispiel 2.2: Load-Befehl mit Offset

2.1.8 Schicht 7: Anwendungsschicht

Die Anwendungsschicht greift auf die Daten der Transportschicht, strukturiert in der Softwirelist, zu. Werden externe Daten, benötigt so formt die Anwendungsschicht ein Kommandopaket mit den nötigen Informationen (Routing-Tabelle), um den Weg zum jeweiligen Knoten zu finden. Softwirenummern werden auch weiters noch in sogenannte *Channels* unterteilt, wie z.B. Service-, Digitale I/O-, Analog-, Drucker- und Weightchannels. Jeder Slave im P-NET muß den Service-Channel beinhalten, der die nötigen Informationen über den jeweiligen Slave enthält. Doch mehr dazu im nächsten Kapitel.

2.2 Datentypen

Durch die Verwendung von fix vorgegebenen Datenstrukturen wird einerseits der Datenaustausch genormt, und andererseits hat eine fixe Struktur den Vorteil unnötige Datenkonvertierungen zu vermeiden. Bei den folgenden Erklärungen wird im Speziellen auf den Datenaustausch – wie und in welcher Reihenfolge die einzelnen Bytes übertragen werden – eingegangen. Dies ist, was wie wir später in der Protokollimplementierung noch sehen werden, von essentieller Bedeutung.

2.2.1 Einfache Variablen

Einfache Variablen sind die Datentypen *Boolean, Byte, Char, Word, Integer, Longinteger, Real* und *Longreal.*

2.2.1.1 Boolean

Eine Booleanvariable wird wie ein Byte übertragen, jedoch ist nur das Bit 0 ausschlaggebend (vgl. Abbildung 2.12)

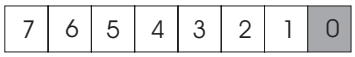


Abbildung 2.12: Boolean-Variable

2.2.1.2 Byte

Ein Byte wird als solches übertragen und beinhaltet einen Wert zwischen 0 und 255.

2.2.1.3 Char

Eine char-Variable wird wie eine Byte-Variable übertragen, jedoch wird der Inhalt als ein ASCII-Zeichen (ISO 8-bit Code) interpretiert.

2.2.1.4 Word

Eine Word-Variable besteht aus zwei Bytes, welche wie in Abbbildung 2.13 angegeben übertragen wird. Eine Word-Variable hat also einen Wertebereich von 0 bis 65535.

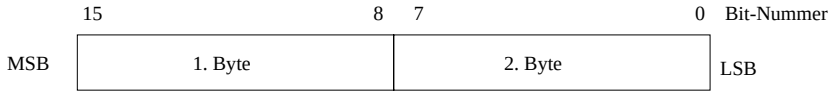


Abbildung 2.13: Word

2.2.1.5 Integer

Die Integer-Variable wird im Zweierkomplement abgespeichert, wodurch sich ein Wertebereich von -32768 bis +32767 ergibt. Die Übertragung einer Variable des Integertyps erfolgt auf gleiche Art wie die der Word-Variable.

2.2.1.6 Longinteger

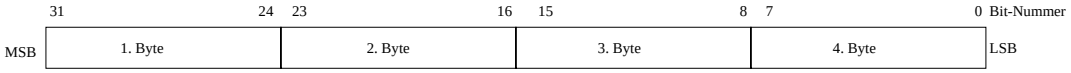


Abbildung 2.14: Longinteger

2.2.1.7 Real

Ein Real-Wert wird ebenso wie ein Longinteger in vier Bytes übertragen, jedoch werden hier die unteren 23 Bit für die Mantisse, die darauffolgenden 8 Bit sind der Exponent und das letzte Bit wird als Vorzeichen interpretiert. Eine Real-Variable hat einen Wertebe-

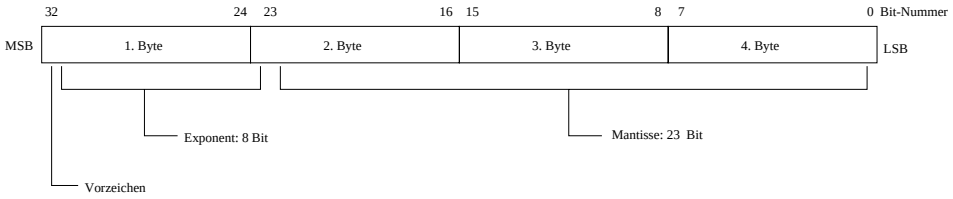


Abbildung 2.15: Real

reich von $-3.4 * 10^{-}38$ bis $-3.4 * 10^{+}38$ für negative Zahlen und den selben Wertebereich für positive Zahlen.

2.2.1.8 Longreal

Die Longreal-Variable belegt bei der Übertragung acht Bytes. Die Aufteilung ist ähnlich der Real-Variable, jedoch werden hier die ersten 52 Bit für die Mantisse, danach 11 Bits für den Exponenten und das MSB wird wieder als Vorzeichen gewertet.

2.2.2 Komplexe Variablen

Als Komplexe Variablen werden *String, Array, Record* und *Buffer* bezeichnet.

2.3 Channelstruktur

In diesem Kapitel wird auf die Channelstruktur [Dat92] im P-NET eingegangen, wobei die generelle Struktur der Channels, der Servicechannel und zur besseren Veranschaulichung der digitalen IO-Channel genauer beschreiben werden. Die restlichen Channelstrukturen sind in [Dat92] nachzulesen.

Ein Channel ist eine Zusammenfassung von 16 Variablen, welche je nach Channeltype gewisse Anforderungen entsprechen müssen. Somit liegen verschiedene Typen oder ein Record aus verschiedenen Typen vor.

2.3.1 Speichertypen

Die P-NET-Speichertypen stehen sehr mit der jeweilig verwendeten Speicherart, flüchtiger oder nichtflüchtiger Speicher (RAM, ROM bzw. EEPROM) in Beziehung. Welcher Speichertyp verwendet wird, hängt natürlich primär von der jeweiligen Aufgabe der Variable ab, z.B. ist die DeviceID read-only und liegt im ROM im Gegensatz zu einem Mailbox-Buffer, der Read-Write ist und im RAM liegt.

Die in der Channelstruktur verwendeten Speichertype sind wie folgt definiert:

- Read-Only
  - PROM Read-Only
    Der PROM-Speicher ist immer schreibgeschützt und kann somit auch nie verändert werden.
  - RAM Read-Only
    Eine Variable im RAM ist aber dennoch nur lesbar.
- Read-Protected-Write
  - RAM RPW
    Die Variable ist immer geschützt, außer nach dem Setzen eines entsprechenden Flags.
  - EEPROM RPW
    Wird immer geschützt, bis *WriteEnable* auf TRUE gesetzt wird. Danach behält die Variable den Wert auch nach einem RESET.
- Read Write
  Diese Variable kann immer verändert werden und ist nach dem RESET gleich NULL.
- Read Write, Protected Backup Write
  Die Variable ist im EEPROM und im RAM vorhanden, wird aber nach dem RESET vom EEPROM in den RAM kopiert. Bei einer Änderung der Variable wird nur der RAM verändert, außer es ist die *WriteEnable* auf TRUE gesetzt, dann wird auch der EEPROM beschrieben.

2.3.2 Generelle Channelstruktur – Channel N

Jeder Channel muß der einer generellen Struktur gehorchen, sogenannte fixe Variablen in einem Channel, d.h. ein fixes Set von Softwirenummern ist für jeden Channel gleich. Tabelle 2.2 gibt die allgemeneine Struktur des Channel-N wieder.

SWNr. n0:
Wie zu erkennen ist, enthält Softwirenummer-0 immer den primären Wert – in einem Analog-IN-Channel daher der Analog-IN-Wert.

SWNr. n9
ChConfig enthält channelspezifische Konfigurationen.

SWNr. nD
Diese Nummer beinhaltet das Datum und Kategorie der letzten Wartung.

SWNr. nE
ChType beschreibt welche Softwirenummern überhaupt existieren und um welchen Channeltype (eindeutige Nummer z.B. Digital_IO: ChannelType=2) es sich handelt.

SWNr. nF
ChError enthält die Fehlercodes.

Softwire-Nummer	Bezeichnung	Speichertyp	Format	Speicherart
n0	primärer Wert	——	——	——
n1				
n2				
..	..	..	..	..
n7				
n8				
n9	ChConfig	EEPROM RPW	——	Record
nA				
nB				
nC				
nD	Maintenace	EEPROM RPW	——	Record
nE	ChType	PROM Read Only	——	Record
nF	ChError	RAM Read Only	——	Record

Tabelle 2.2: Channel N

2.3.3 Service-Channel – Channel 0

Der Service-Channel muß in jedem Knoten vorkommen, da der Service-Channel wichtige Information bezüglich der Ansteuerung des Knotens enthält. Dieser Channel liegt immer an den ersten 16 Softwirenummern.

SWNr. 00
NumberOfSWNo enthält die Anzahl der im Knoten vorkommenden Softwirenummern.

SWNr. 01
DeviceID ist ein RECORD und enthält herstellerspezifische Daten.

SWNr. 04
PNETSerialNo beinhaltet die P-NET-Adresse und eine Seriennummer.

SWNr. 0D
WriteEnable ist jene Variable die es ermöglicht Write-Protected-Variablen zu beschreiben (WriteEnable=TRUE)

SWNr. 0E
ChType beschreibt auch hier welche Softwirenummern überhaupt existieren und um welchen Channeltype (Service-Channel: Channeltype = 1) es sich handelt.

SWNr. 0F
CommonError beinhaltet im Fehlerfall die nötige Information (für alle Channnels im Modul).

Die Softewire-Nummer 04 wird neben den normalen Befehlen auch noch von einer "Special function" bedient, der "Set P-NET Address"-Funktion. Diese Funktion dient zum Setzen von Knoten-Adressen bei neu im Feldbus zu installierenden Knoten. Doch mehr dazu in Kapitel 3.

Softwire-Nummer	Bezeichnung	Speichertyp	Format	Speicherart
00	NumberOfSWno	PROM Read only		Integer
01	DeviceID.	PROM Read only	——	Record
02				
03				
04	PNETSerialNo	Special function	——	Record
05				
06				
07				
08				
09				
0A				
0B				
0C				
0D	WriteEnable	RAM Read Write	——	Boolean
0E	ChType	PROM Read Only	——	Record
0F	CommonError	RAM Read Only	——	Record

Tabelle 2.4: Service Channel

Kapitel 3

Implementation

Zur besseren Veranschaulichung möchte ich die Erklärung der Implementierung des Protokollstacks in einzelne Schritte unterteilen:
Hiezu möchte ich wieder in einzelnen OSI-Schichten gliedern, wodurch es einfacher wird, die jeweiligen Funktionen den Schichten zuzuordnen. Folge dessen ist es auch leichter möglich, den Standard mit der Implementierung zu vergleichen.
Die Abbildung 3.1 gibt einen Überblick der Schichtung samt den zugehörigen Funktionen. Eine Beschreibung der Funktionen soll dieses Kapitel wiedergeben. Die wichtigsten Funktionen werden zusätzlich noch mit SDL-Diagramme beschrieben, die restlichen Diagramme sind im Anhang abgebildet.

3.1 Bitübertragungsschicht

3.1.1 P-NET-Interface

Das P-NET-Interface ist das Bindeglied zwischen der seriellen Schnittstelle (RS232) des 88C166-Microcontrollerboards und des RS-485-Bus des P-NETs. Es wird eine Pegelwandlung, beziehungsweise eine Umwandlung in das Differenzsignals durchgeführt. Wie schon in der Einführung beschrieben, wird eine NRZ-Kodierung verwendet, d.h. daß keine Pegeländernug innerhalb eines Bits erfolgt. Die Schaltung und die verwendete Hardware werden im Kapitel 5 ausführlich beschrieben.

3.1.2 PEC-Funktion

Die PEC-Funktion (Peripheral Event Controller) [Joh93, KHM95] ist eine ideale Funktion zur automatischen Steuerung des seriellen Empfangs. Bei steigenden Anforderungen

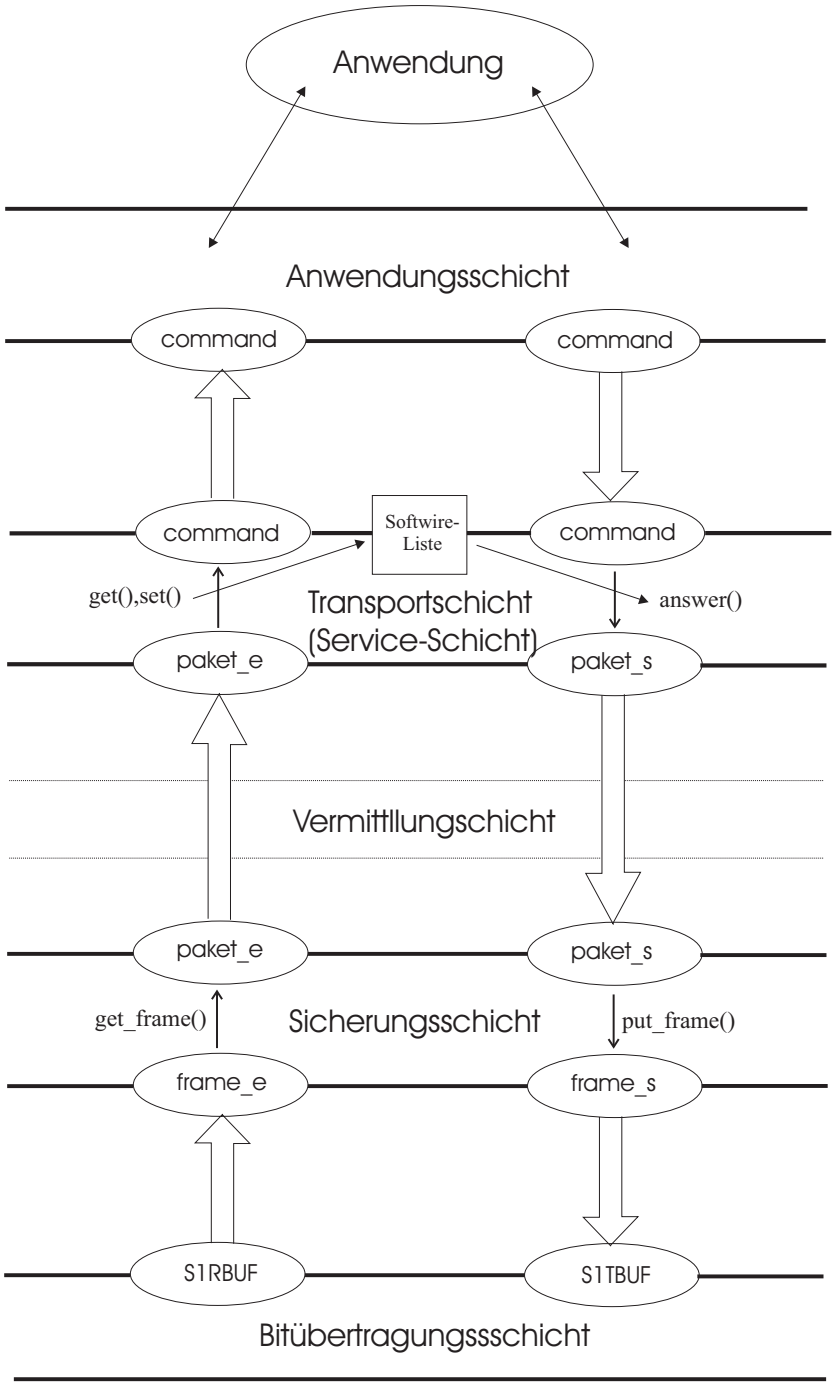


Abbildung 3.1: Übersicht der Implementierung

einer Applikation ist es von Vorteil, den PEC-Transfer zu verwenden.
Der PEC-Transfer zeichnet sich dadurch aus, daß er den normalen Programmablauf für nur lediglich einen Maschinenzyklus unterbricht. Dieser Vorgang wird als injezierte Instruktion bezeichnet. Es wird somit in die Befehlspipline des Mikrocontrollers nur ein MOV-Befehl injiziert, wodurch die Abarbeitung eines Programms nur um einen Maschinenzyklus verzögert wird (ca.100ns bei 20MHz CPU-Takt). Als Quelle zur Injizierung dient ein Interrupt-Request, z.B. S1-Receive Interrupt.

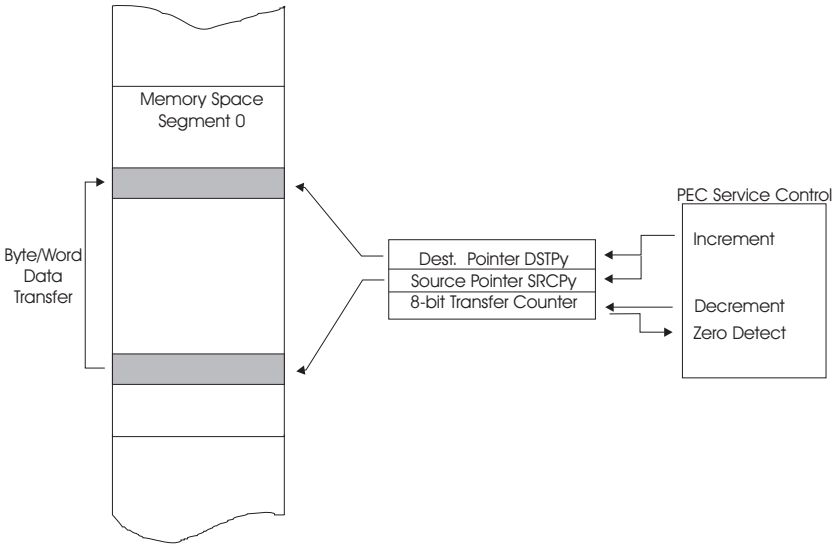


Abbildung 3.2: PEC-Transfer

Da häufig beim Empfang eines Datenblocks über die serielle Schnittstelle die Anzahl der zu empfangenen Bytes nicht bekannt ist, bereitet es Schwierigkeiten, den PEC-Transfer zu verwenden, als keine Informationen über das Ende einer Übertragung vorhanden sind. Abhilfe schafft die "IDLE LINE DETECTION" (beschrieben in [KHM95], welche genau definierten Bitpausen – beim P-NET sind es 11 Bit – zwischen einzelne Datenblöcke erkennt, und somit den PEC-Transfer beendet. Die Erkennung erfolgt mittels einem Timeout, welches durch ein externes Signal (hier serielle Schnittstelle 1) bei Aktivität immer wieder erhöht wird.

Da der PEC-Transfer einige Probleme bereitete im Zusammenhang mit der vorher implementierten Wake-Up-Funktion (siehe 3.2.1.1), ist er in der derzeit vorhandenen Implementation nicht vorhanden. Da dieser Transfer jedoch einen Geschwindigkeitsvorteil, besonders in Hinblick auf die Implementation eines Masters mit dem 88C166-Mikrocontroller mit sich bringt, sollte dieses Kapitel nicht fehlen.

3.2 Sicherungsschicht

Wie in Kapitel 2.1.3 erklärt, wurde in dieser Schicht Rahmenerkennung/Adreßerkennung, Buszugriff und Fehlerkontrolle implementiert.

3.2.1 Slave-Empfang

Zu Beginn erlaube ich mir, auf die wichtigen Einstellungen des Mikrocontrollers bezüglich der seriellen Schnittstelle einzugehen. Da wie bereits beschrieben ein Adreßbit für die Unterscheidung von Adressen und Daten verwendet wird, muß auch die serielle Schnittstelle des Mikrocontrollers auf neuen Datenbits eingestellt werden. Die Konfiguration samt den Werten für die Steuerregister des Mikrocontrollers, der seriellen Schnittstelle (vgl. [Joh93]) sieht folgend aus:

Asynchroner Modus	9-bit-Daten
Stoppbit	1
Parity-Überprüfung	deaktiv
Overflow-Überprüfung	aktiv
Framing-Überprüfung	aktiv
Steuerregisterwert	S1CON=80D4hex

Nur das jeweils erste Adreßbyte eines Rahmens wird – wie in Abbildung 2.4 gezeigt – durch das neunte Bit gekennzeichnet, wodurch dir Möglichkeit der Implementierung einer Wake-Up-Funktion besteht.

3.2.1.1 Wake-Up-Funktion

Dieser Modus (beschrieben in [Joh93]) erlaubt es, das Empfangsregister der jeweiligen seriellen Schnittstelle so zu steuern, daß nur bestimmte Daten empfangen werden. Dadurch werden im Wake-Up-Modus (S1M=101B) nur jene Bytes empfangen, in denen das Wake-Up-Bit=1 ist – es erfolgt somit keine Auslösung des Empfangs-Interrupts und auch keine des Fehlerinterrupts. Daraus folgt, daß nur jeweils das erste P-NET-Adreßbyte des Adreßfeldes empfangen wird.
Durch diesen Vorgang ist es ideal möglich, den P-NET-Ring nach der eigenen Adresse "abzuhorchen", jedoch muß dabei darauf geachtet werden, rechtzeitig vom Wake-Up-Modus in den normalen 9-bit-Daten-Modus(S1M=100B) umzuschalten, um die eigentlichen Daten empfangen zu können. Die einzig direkt von der Hardware unterstützte Funktion ist nämlich die Unterdrückung des Empfangs in Abhängigkeit vom Wake-Up-Bit.
Im Wake-Up-Modus muß der Registerwert 80D5hex in das S1CON geschrieben werden.

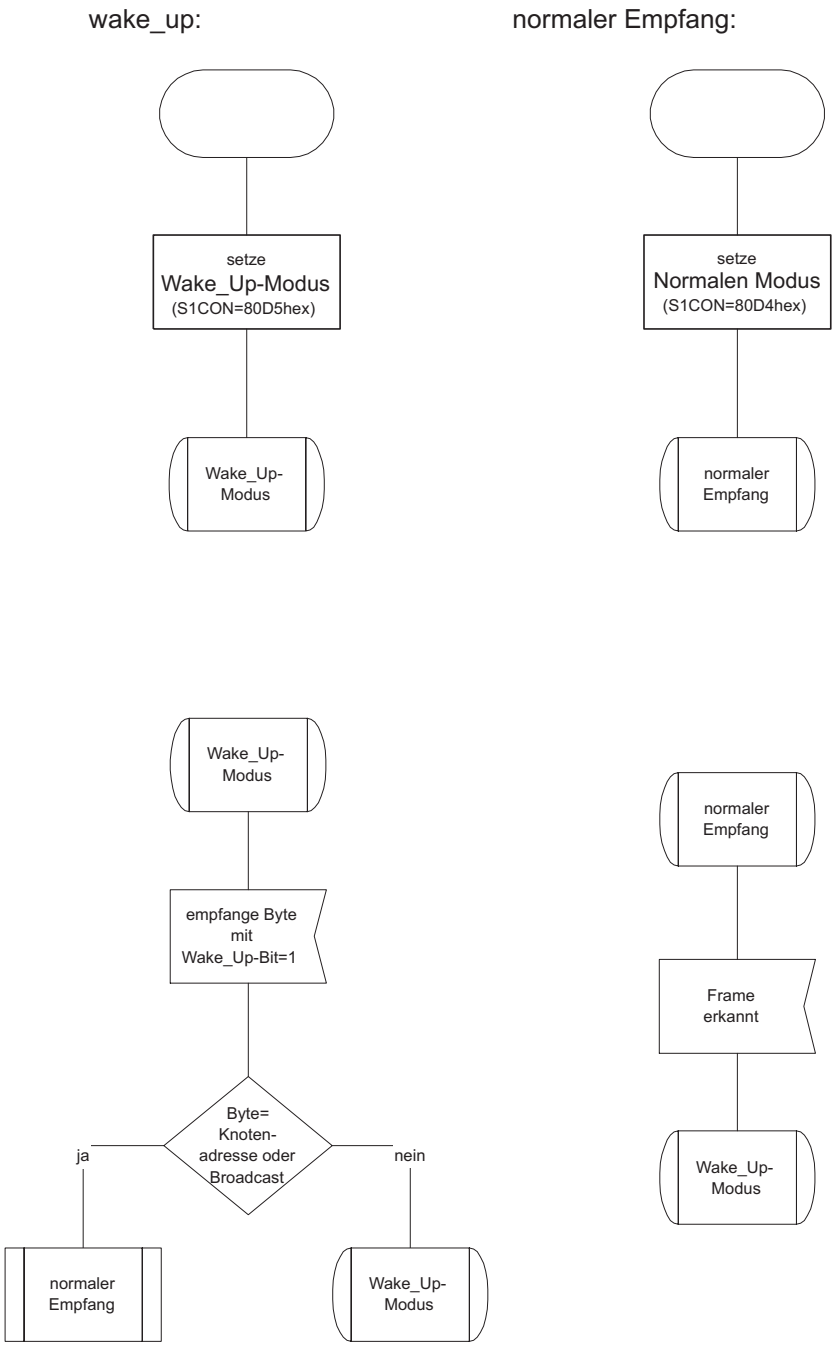


Abbildung 3.3: Wake-Up-Funktion

3.2.1.2 Fehlererkennung

Wie bei den meisten asynchronen Kommunikation wird jedes einzelne Byte auf Rahmen- und Überlauffehler geprüft. Dieser Fehler resultiert in einem Antwortrahmen mit dem Fehlercode für Overrun/Framing-Error. Die Funktion (siehe 3.4.2.4) `Answer(code)` erfüllt die Aufgabe des Sendens von Antwortrahmen, welche entweder einen Fehlercode oder eine erfolgreiche Bestätigung von Anfragen eines Masters enthalten.
Je nach der Anzahl der empfangenen Byte, wird die normale Fehlererkennung oder die reduzierte Fehlererkennung durchgeführt.

3.2.2 Rahmen-/Adreßerkennung – Funktion `get_frame()`

Der Slave empfängt nur all jene Rahmen, bei denen das siebente Bit der ersten Adresse gleich 0 ist. Der Slave filtert also die "Request" von den "Response"-Rahmen heraus. Nach diesem ersten Schritt wird nun überprüft, ob es sich um einen einfachen Adreßtyp handelt, d.h. um eine null-eins Kombination des siebenten Bits der ersten beiden Abdreßfelder (vergleiche Kapitel2.1.3.2).
In den Abbildungen 3.4 und 3.5 wird der Ablauf der Rahmenerkennung bzw. der Erkennung des Rahmens, bei dem die Zieladresse gleich der eigenen Adresse des Slaves entspricht, in SDL-Diagrammen dargestellt.
Es werden die einzelne Zustände durchlaufen, bis aus dem empfangenen Rahmen die entsprechende Information, wie in Kapitel 2.1.3.2 Abbildung 2.3 beschrieben, herausgefiltert wird. Zu beachten ist die Sonderbehandlung der Service-Funktion beim Setzen der P-NET-Adresse. Diese Funktion wird vor allem bei der Inbetriebnahme von fabriksneuen Modulen verwendet, wobei ein Broadcast mit der dem Modul enthaltenen Seriennummer und der gewünschten P-NET-Adresse ausgesendet wird.
Diese Daten werden in eine Datenstruktur, die dem Paketformat wie in 2.1.6 beschrieben entspricht, transformiert. Auf diesem Paketformat wird in den höheren Schichten in Folge aufgebaut. Da jedoch in Slaves die Datenstruktur

3.2.3 Senden von Rahmen – Funktion `put_frame()`

Das Senden von Rahmen übernimmt die Funktion `put_frame()`, welche auch gleichzeitig die Bytes zur Fehlererkennung berechnet und übermittelt. Die jeweilig zu sendenden Daten entnimmt die Funktion der Datenstruktur `paket_s()`, welche wie im Kapitel 2.1.6 beschrieben, in der Transportschicht des OSI-Modells verwendet wird.
Hierbei ist wiederum zu beachten, daß im erste Byte des Adreßfeldes das 9.Bit, also das Adreß/Datenbit, gesetzt ist. Danach werden die einzelnen Adreßfelder aus der `paket_s`-Struktur entnommen und übermittelt.

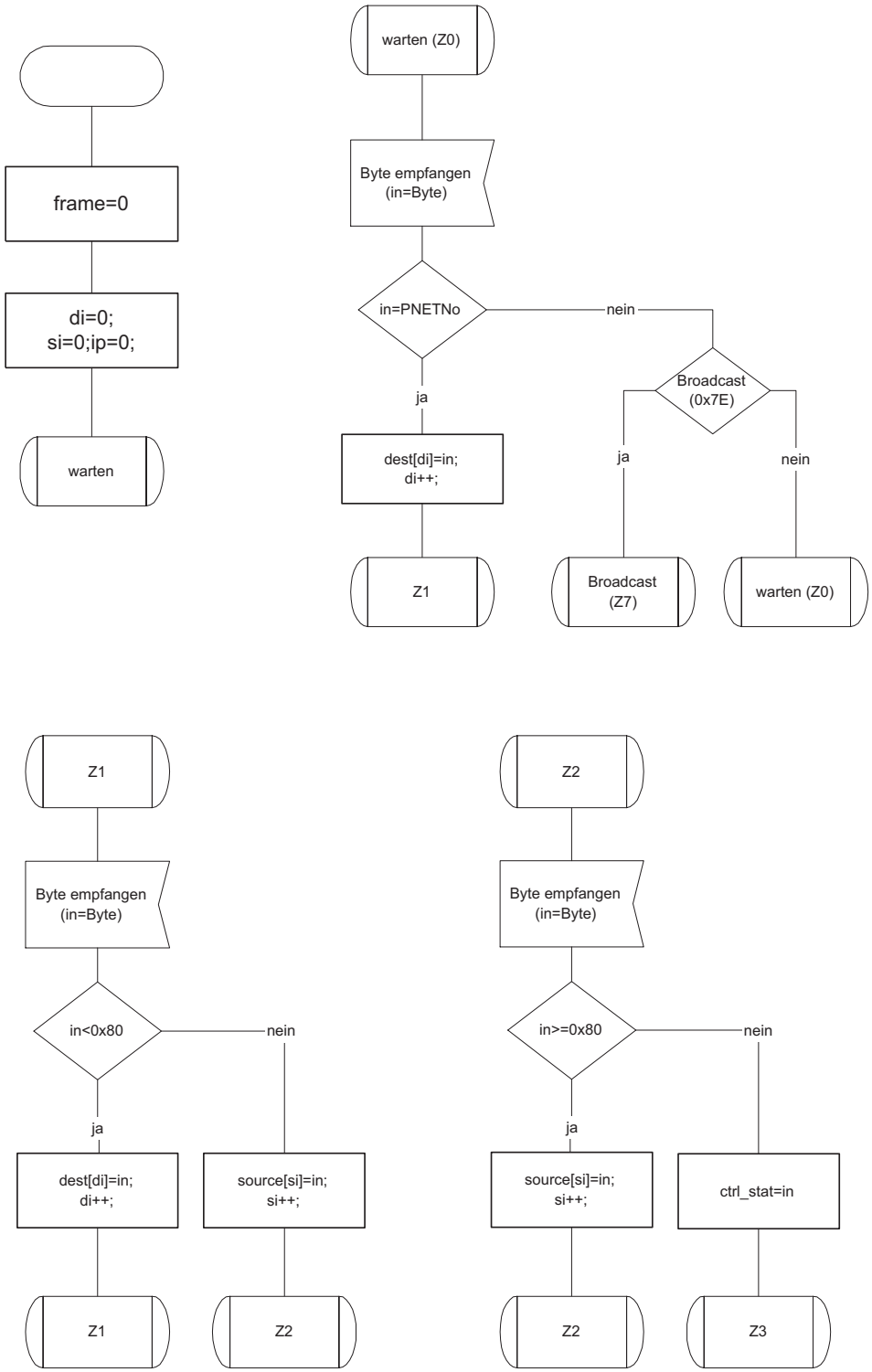


Abbildung 3.4: Rahmen-Erkennung (Teil 1)

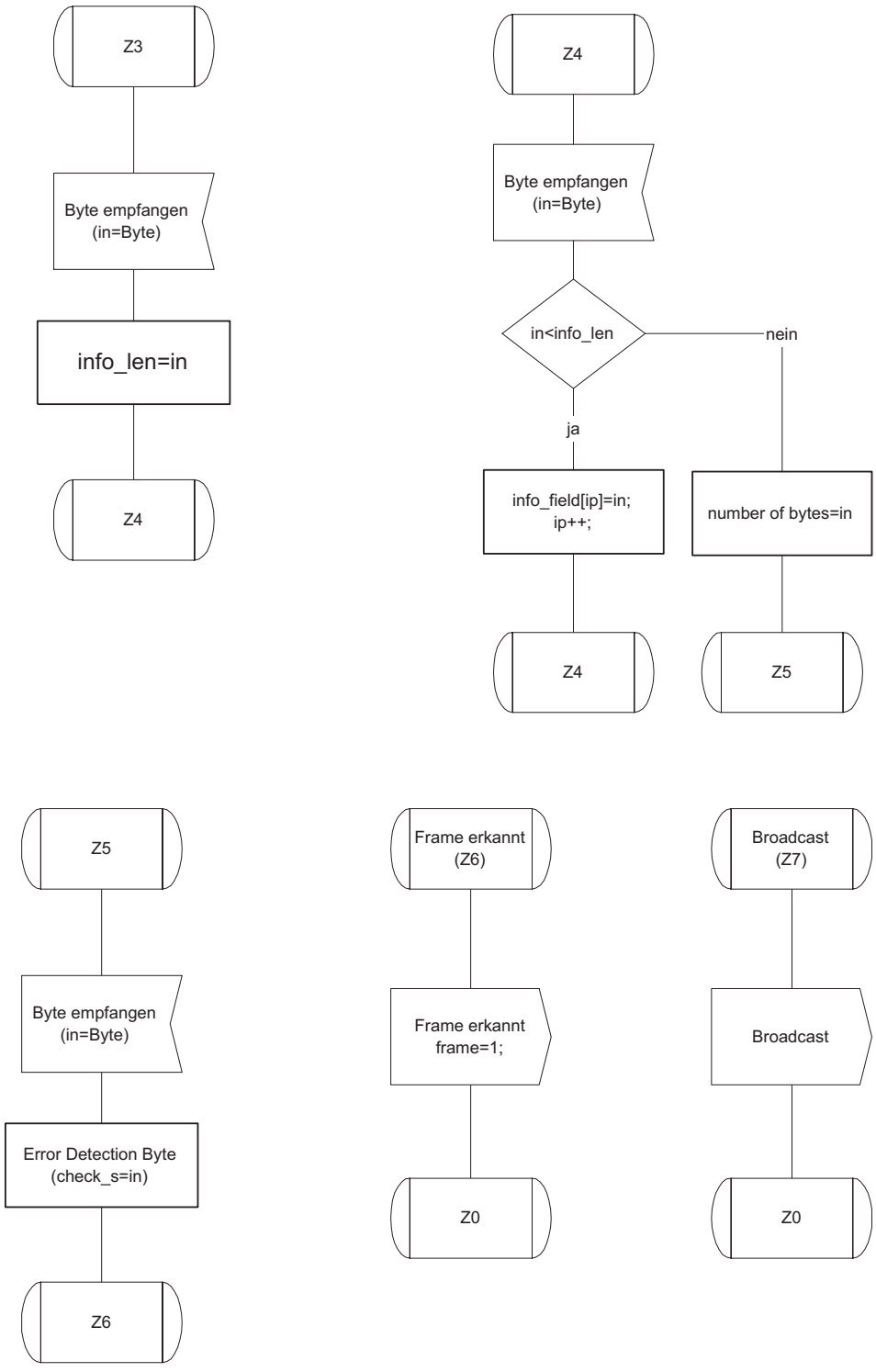


Abbildung 3.5: Rahmen-Erkennung (Teil 2)

Besondere Beachtung sollte der Übermittlung der Error-Detection-Bytes am Ende eines Rahmen geschenkt werden. Entgegen der im Standard [Dat93] beschriebenen Fehlererkennungsroutine zeigte sich, daß bei Verwendung der reduzierte Fehlererkennung, ebenfalls 2 Bytes zu senden sind! Diese beinhalten aber den gleichen Wert.

3.2.4 Library: `layer2.h`

Die Library `layer2.h` beinhaltet alle auf der Sicherungsschicht vorkommenden Funktionen wie

- `get_frame()`
- `put_frame()`
- `ini_fkt()` – dient zur Initialisierung der seriellen Schnittstellen
- `clean()` – löscht den letzten Inhalt der frame_e-Struktur

3.3 Vermittlungschicht

Die oben in Sicherungsschicht betriebene Trennung von Rahmen-Struktur und Paket-Struktur, oder besser, die Einführung einer Vermittlungsschicht, welche eigentlich nur als "Durchkontaktierung“ zwischen der Sicherungsschicht und der Transportschicht dient, soll nur die Übersichtlichkeit steigern.

3.4 Transportschicht

In der Transportschicht wird die komplette Softwireliste verwaltet. Da P-NET vorgegebene Module mit ebenso fixerten Channels verwendet, muß auch der P-NET-Slave einer Moduldefinition genügen. Um eine möglichst großes Anwendungsspektrum zu zeigen, wurde das Universelle Prozeß Interface PD3221 (UPI) gewählt [PD95].
Das UPI enthält eine wie in untenstehender Tabelle , fix vorgegebene Channelstruktur. Die Aufteilung der Leds und der Schalter ist ebenfalls in der Tabelle angegeben. Eine genauere Beschreibung des eigentlichen Zugriffs auf den Knoten erfolgt in Kapitel 4.

Channel	Channel-Nummer	Belegung
Service	0	—
Digital-I/O	1	LED 1
Digital-I/O	2	LED 2
Digital-I/O	3	LED 3
Digital-I/O	4	LED 4
Digital-IN	5	Schalter 1
Digital-IN	6	Schalter 2
Common-I/O	7	—
Analog-IN	8	Analog 1
Analog-IN	9	Analog 2
Current-OUT	A	nicht benützt
PID-Regler	B	n.b.
Calculator	C	n.b.
Pulse-Prozessor	D	n.b.

Tabelle 3.3: UPI-Channelbelegung

3.4.1 Softwirelist

Alle Softwirevariablen wurden in der Datei *channels.h* definiert. Die Softwireliste besteht aus einem Feld von Elemente, jedes Element ist vom Type *SWList*. Der Type SWList definiert somit jeweils eine Softwirenummer. Der Type SWList ist folgend definiert:

```
typedef struct SWList {
char type;
char status;
void *point;
int NumbOfBytes;
} SWL;
```

Die Variable `type` speichert die Information über den eigentlichen Datentype der SW-Variable und unterscheidet zwischen den Typen (eingeklammerter Wert = hexadezimaler Code):

- Boolean (00h)
- Byte (01h)
- Word (02h)
- Integer (03h)

- Longinteger (04h)
- Real (05h)
- Loangreal (06h)
- Complex (80h)

Der zweite Eintrag `status` enthält die Lese- bzw Schreibrechte für die SW-Variable:

- No Protection (00h)
- Read Only (10h)
- Write Protection (20h)

Der Zeiger `point` weist auf die eigentliche globale Variable, das ist jene Variable, die den Wert der SW-Nummer enthält.

```
const struct SWList SW[xx] = {
/*SW00*/ Integer, Read_Only, &NumberOfSWNo, 2,
/*SW01*/ Complex, Read_Only, &DeviceID, 32,
/*SW02*/ NotUsed,
/*SW03*/ NotUsed,
/*SW04*/ Complex, No_Protection, &PNETSerialNo, 21,
/*SW05*/ NotUsed,
/*SW06*/ NotUsed,
/*SW07*/ NotUsed,
/*SW08*/ NotUsed,
/*SW09*/ NotUsed,
/*SW0A*/ NotUsed,
/*SW0B*/ NotUsed,
/*SW0C*/ NotUsed,
/*SW0D*/ Boolean, No_Protection, &WriteEnable, 1,
/*SW0E*/ Complex, Read_Only, &ChType, 6,
/*SW0F*/ Complex, Read_Only, &CommonError, 4,
/* ------------------ IO Channel fuer LED1 --------------- */
/*SW10*/ Byte, No_Protection, &IO_1, 1,
/*SW11*/ NotUsed,
/*SW12*/ NotUsed,
/*SW13*/ NotUsed,
/*SW14*/ NotUsed,
/*SW15*/ NotUsed,
```

```
/*SW16*/ NotUsed,
/*SW17*/ NotUsed,
/*SW18*/ NotUsed,
/*SW19*/ NotUsed,
/*SW1A*/ NotUsed,
/*SW1B*/ NotUsed,
/*SW1C*/ NotUsed,
/*SW1D*/ NotUsed,
/*SW1E*/ Complex, Read_Only, &ChTypeIO1, 8,
/*SW1F*/ Complex, Read_Only, &CommonErrorIO, 4,
/* ------------------- IO Channel fuer LED2 ---------------- */
.......
...
};
```

3.4.1.1 Regeln für komplexe Variablen

Beim Ablegen von komplexen Datentypen (Records) ist darauf zu achten, daß folgende Regeln befolgt werden:

- Variablen mit einem Inhalt von *mehr* als einem Byte müssen einen *geraden* Offset haben.
- Variablen mit einem Inhalt von nur *einem* Byte können an einem beliebigen Offset stehen.
- alle komplexen Subtypen, welche mehr als ein Byte belegen, benötigen also immer nur eine gerade Anzahl an Bytes

Um diese Regeln einhalten zu können, müssen Dummybytes eingefügt werden. Als Beispiel sei die Variable PNETSerialNo aus dem Service-Channel angegeben:

```
typedef struct PNETSer {
char PNETNo; /* Knoten Adresse */
char dummybyte;
char Serie[20]; /* Seriennummer vom Hersteller */
} PNETSerD;
```

3.4.1.2 Hinweis zur Speicherorganisation des Mikrocontrollers

Da der Mikrocontroller den Speicher byteweise organisiert, ist besonders zu achten (vgl. Abbildung 3.6), daß zuerst das Low-Byte und erst dann das High-Byte abspeichert wird. Dies führt bei der Behandlung von Variablen mit mehr als ein Byte zu einem Mehraufwand.
Diese bedeutet ebenfalls, daß 16 bit breite Worte immer auf geraden Adressen beginnen.

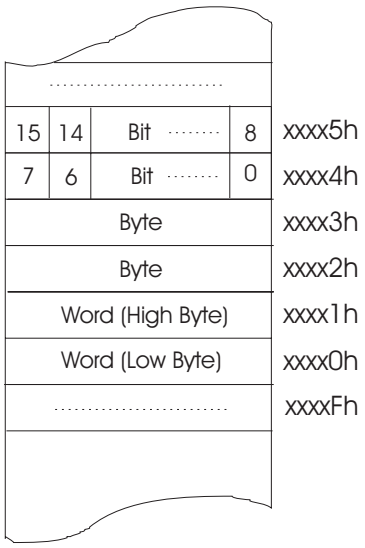


Abbildung 3.6: Anordnung von Bytes, Worten und Einzelbits im Speicher

3.4.2 NET-Service

Das P-NET-Service oder "Net Service" empfängt Pakete von der Vermittlungsschicht, verarbeitet das Kontroll/Status-Feld und führt dann den entsprechenden Befehle aus. Die Befehlsstruktur wurde in die zwei Basis-Befehle *Load* und *Store* unterteilt. Der Befehl Store wurde im Programm durch die Funktion `set(SwNr,Mode)` realisiert. Diese Funktion unterscheidet noch durch den Parameter `Mode` unter den zusätzlichen Befehle *AND, OR und TEST-AND-SET.*
Der *Load-Befehl* wurde durch die Funktion `get(SwNr)` verwirklicht.

Wenn ein Rahmen auf der Sicherungsschicht empfangen wird, so wird je nachdem, ob es sich um einen Broadcast oder einen normalen Befehl handelt, entweder die Funktion `set_pnetnr` oder die Funktionen `get` bzw. `set` aufgerufen. Vergleiche dazu die SDL-Diagramme in Abbildung 7.1. Der Anhang enthält die restlichen Diagramme.

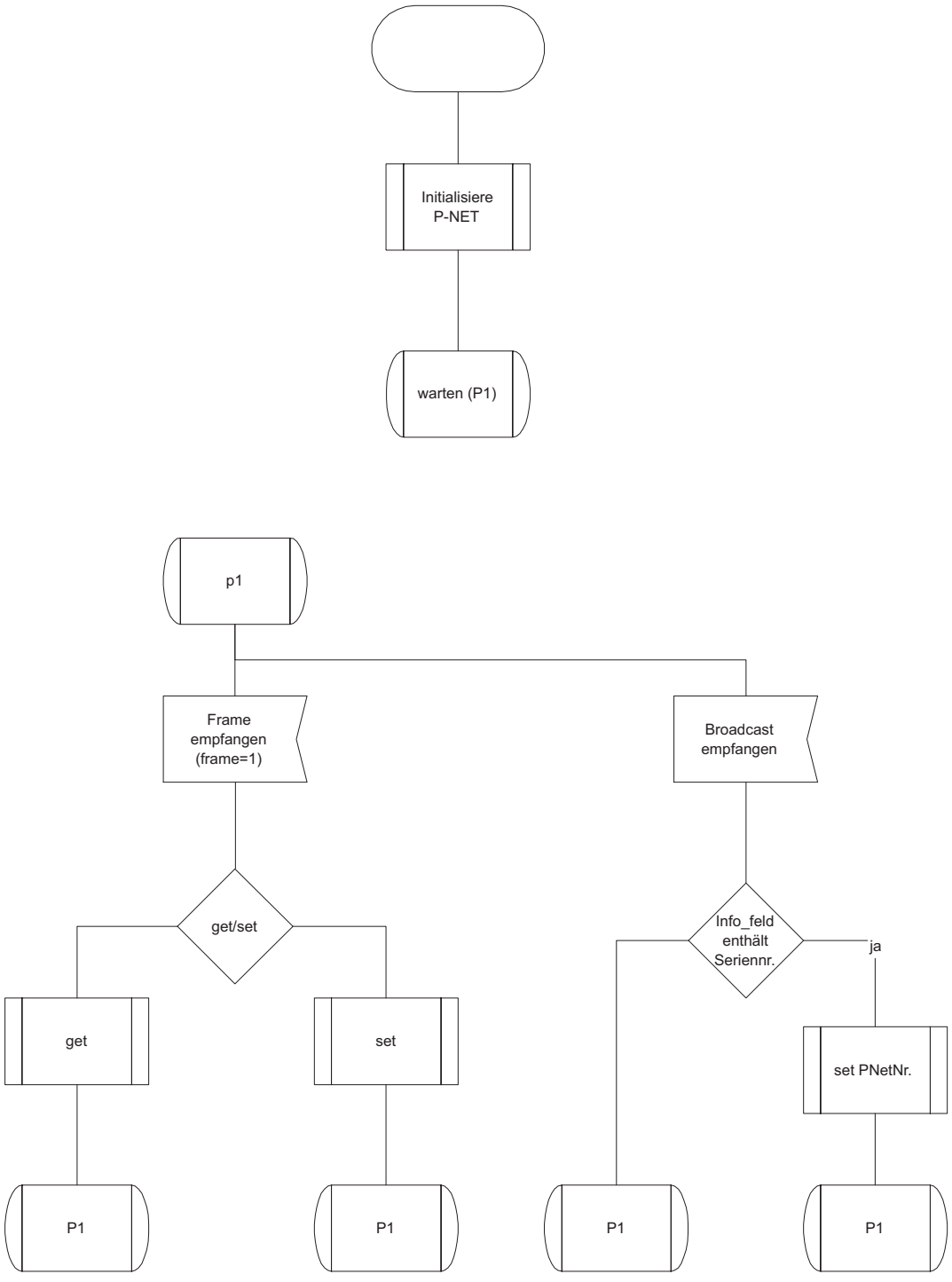


Abbildung 3.7: P-NET-Service

Fehlerbehandlung

Folgende Fehler werden in P-NET Service behandelt:

- *Instruction error*: wenn ein Befehl nicht implementiert ist (z.B.: Long-Befehle in einem Slave)
- *SWNo error*: es wird auf eine falsche Softwirenummer zugegriffen
- *Data format error*: Tritt auf, wenn die Datenlänge mit der in der SWNr nicht übereinstimmt, in komplexen Variablen jedoch wird nur überprüft ob die maximale Anzahl an Bytes im Record überschritten wird
- *Write protection*: ein schreibender Zugriff auf eine schreibgeschützte Variable erzeugt diesen Fehler

Die Antworten auf die oben angegebenen Fehlerfälle wurden durch die Funktion `answer(code)` realisiert.

3.4.2.1 Die Funktion `get(SWNR)`

Diese Funktion realisiert einen Load-Befehl, d.h. wenn ein Load-Befehl von einem Master auf eine Softwire-Nummer des Slaves erfolgt. Die Funktion überprüft an Hand des in der Softwire-Nummer enthaltenen Datentypen, um welche Variable es sich handelt (Boolean, Byte, Word, Integer usw) oder ob es sich um eine komplexe Variable handelt.
Danach wird der Offset berechnet, welcher natürlich nur bei komplexen Variablen Sinn macht.

3.4.2.2 Die Funktion `set(SWNR,Mode)`

Die `set(SWNR,Mode)`-Funktion führt je nach Parameter `Mode` einen STORE-, AND-, OR- oder TEST-AND-SET-Befehl durch. Zu Beginn wird aus der Softwireliste der Status der zu behandelnden Softwirenummer (`SWNR`) ausgelesen. Dabei wird darauf geachtet, ob es sich um den Status `No_Protection, Read_Only` oder `Write_Protection` handelt (Beschreibung der Softwireliste siehe 3.4.1). Falls sich ergibt, daß es sich um einen Read_only- oder Write_Protection-Status ohne WriteEnableFlag=TRUE handelt, wird eine Fehlermeldung mit der Funktion `answer(code)` an den entsprechenden Master geschickt.
Wurden keine Schreibrechte verletzt, wird der dem Parameter `Mode` entsprechende Befehl durchgeführt. Natürlich wird noch zwischen den einzelnen Variablentypen unterschieden. Der `Mode`-Parameter wird wie folgt zugeordnet:

- Mode=1: STORE auf entsprechende SWNR
- Mode=3: AND
- Mode=4: OR
- Mode=5: TEST-AND-SET

Die Bedeutung der einzelnen Befehle ist bitte dem Kapitel 2.1.6.2 zu entnehmen.

3.4.2.3 Die Funktion `set_pnetnr()`

Wird ein Knoten in einen bestehenden Feldbus integriert, so entsteht ein Problem bein Setzen der P-NET-Adresse. Dies wird durch die spezielle Funktion `set_pnetnr()` gelöst. Durch das Aussenden eines Broadcast samt der Seriennummer und der gewünschten Adresse erhält der Knoten seine neue Adresse. Aufgrund des Broadcast erhalten zwar alle Knoten im Bus den Rahmen, doch durch die eindeutige Seriennummer kann jeder Knoten kontrollieren, ob er die Funktion zum Setzen der Adresse aufrufen soll oder nicht.
Hierzu liest die Funktion die eigene Seriennummer aus der Softwirenummer 4 (vgl. Tabelle 2.4) und vergleicht sie mit der erhalten Seriennummer. Stimmen diese überein, so wird der Variable `PNETNo` die neue Adresse zugewiesen.

3.4.2.4 Die Funktion *answer()*

Die Funktion `answer(code)` hat die Aufgabe, je nach dem Parameter `code`, ein Antwortpaket zu erzeugen und mittels eine Anforderung an die Funktion `put_frame()` der Sicherrungsschicht die Antwort auf ein Masterrequest einzuleiten. Diese Antwort beeinflußt in den meisten Fällen nur das Kontroll/Status-Feld (vgl. 2.1.6.2) und das Feld *Info-Länge*. Das Kontroll/Status-Feld weist dann einen Fehlerkode oder die OK-Meldung auf und das Feld Info-Länge beinhaltet den Wert 0. Der Parameter `code` ist wie in Tabelle zu verwenden.

Wert für `code`	Bedeutung	Kontroll/Status-Feld
0x00	ok	0x07
0x01	Busy	0x17
0x02	Actual Data Error	0x27
0x03	Wait	0x47
0x05	Histoical Data Error	8.Bit eins
0x0A	Data Format Error	0x28
0x0C	SWNo Error	0x30
0x0D	Node Address Error	0x38

0x0E	Write Protection	0x40
0x10	Instruction Error	0x50

Tabelle 3.4: Parameter der Funktion `answer()`

Die Werte für den Parameter `code` wurde so ausgewählt, daß jene Fehlermeldungen, die den Master betreffen, in der Liste ergänzt werden können [Dat93].

3.4.3 Programm-Service

Im P-NET-Standard [Dat93] wird festgelegt, daß das Programm-Service in Slaves entfällt. Der Vollständigkeit halber soll jedoch kurz gezeigt werden, weshalb es nötig ist, daß in P-NET-Service und Programm-Service unterschieden wird.
Das Programm-Service stellt den Dienst für die Anwendungsschicht dar, d.h. es empfängt und verarbeitet Befehls-Blöcke der Anwendungsschicht. Die in Befehlsblöcke (Commandblock) enthalten Befehle dürfen nicht mit den oben genannten Befehlen (RIS) verwechseln werden. Das Programm-Service hat außerdem die Aufgabe, *Long*-Befehle durchzuführen. Es unterteilt die Datenströme in einzelne 54-Byte lange Felder und führt danach einen *Long*-Befehl durch.

3.4.4 Library: `layer4.h`

Alle Funktionen der Transportschicht wurden in einer Library zusammengefaßt (layer4.h). Dies umfaßt folgende Funktionen und sollen der Übersichtlichkeit halber, nochmal zusammengefaßt werden:

- `get(SWNR)`
- `set(SWNR, Mode)`
- `answer(Code)`
- `set_pnetnr()`

3.5 Anwendungsschicht

Als Anwendungen wurden acht LEDs, vier Schalter, ein LC-Display und vier analoge Eingänge an den Mikrocontroller geschalten. Von denen jedoch, bedingt durch die

Struktur des UPIs, nur vier LEDs, zwei Schalter und zwei Analog-Eingänge als P-NET-Channels realisiert wurden. Die restlichen LEDs, Schalter und das LC-Display dienen als zusätzlich Anwendungen.
Die Ansteuerung der Ausgänge, beziehungsweise das Einlesen von den Eingängen, wurde in einer eigenen Interruptroutine untergebracht. Diese Routine wird durch den Basis-Timer-T3 alle 3,2μs aufgerufen, welche dann die entsprechenden Softwirenummern aktualisiert.

Kapitel 4

Zugriff auf den Knoten

Dieses Kapitel soll die notwendigen Konfigurationen wiedergeben, die nötig sind, um mit dem Knoten tatsächlich kommunizieren zu können.
Im ersten Abschnitt wird gezeigt, wie mit dem Feldbus-Management-System VIGO [PD96] von einem PC aus auf den Knoten zugegriffen werden kann. Dies soll anhand eines Beispiels in Delphi erläutert werden. Der zweite Teil beinhaltet die Information, die nötig ist, mit anderen P-NET-Knoten in Verbindung treten zu können. Als Beispiel sei die Ansteuerung des Knotens mit dem P-NET-Controller PD4000 angeführt.

4.1 Ansteuerung mit VIGO

Durch das OLE2-Interface von VIGO erreicht die Palette der Anwendungen, mit denen auf den Knoten zugegriffen werden kann, einen sehr großen Umfang. Als Beispiele hierzu seien angeführt: Delphi, Excel, Visual C++ usw. (somit jede Anwendung, die auf OLE-Objekte zugreifen kann)

4.1.1 Definition in der MIB

Um mit Vigo auf den Knoten zugreifen zu können, ist zuerst ein UPI-Modul dem aktuelle Projekt hinzuzufügen. Nach diesem Vorgang muß dem Knoten mit der Funktion "Set P-NET Node Address" eine P-NET-Adresse zugewiesen werden. Eine weitere Möglichkeit dies zu erreichen, besteht auch darin, die Adresse gleich direkt in der Library `channels.h` zu ändern, bevor in VIGO selbst jene Adresse eingegeben wird.
Durch diese Aktionen werden die Daten des Slaves, wie Channel-Struktur und Knotenadresse, in der MIB (Manager Information Base) hinzugefügt. Dies bewirkt, daß jede Applikation auf den Slave zugreifen kann.

Durch die flexible Implementierung der Softwire-Liste kann auch jedes andere Modul in VIGO verwendet werden. Dazu muß nur die Softwire-Liste verändert werden.

4.1.2 Zugriff auf ein physikalisches Objekt

VIGO stellt die Eigenschaften und Methoden eines physikalischen Objekts der Applikation in Form eines virtuellen Objekts zur Verfügung. Dadurch kann jede Anwendung direkt durch das Schreiben und Lesen des virtuellen Objekt auf das eigentlich physikalische Objekt zugreifen. (vgl. Abbildung 4.1)
Beispiel 4.1 zeigt den Umgang mit virtuellen Objekten. Zuerst muß das virtuelle Objekt

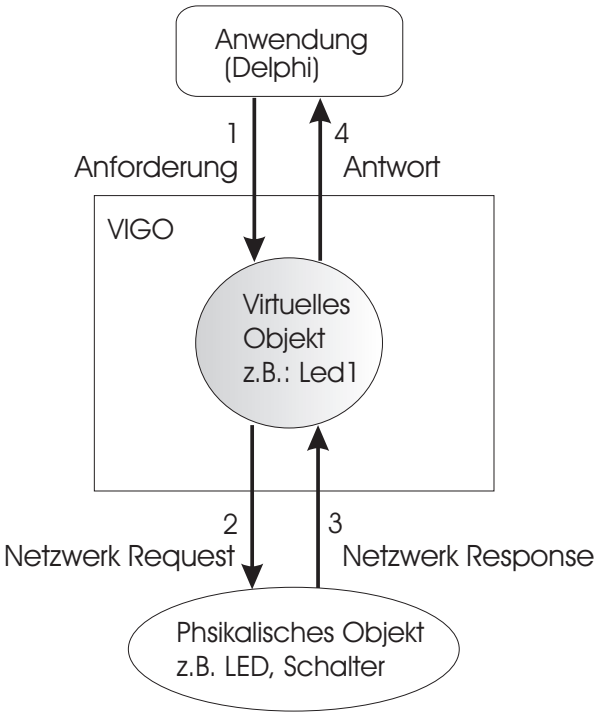


Abbildung 4.1: Zugriff auf physikalische Objekte

erzeugt werden, um dann einer gewissen Variable zugeordnet werden zu können.
Sobald das virtuelle Objekt zur Verfügung steht, kann durch Schreiben oder Lesen in Selbiges der Wert des physikalischen Objekts verändert bzw. ausgelesen werden.

```
Led1: Variant;
switch1: Variant;
Service_NumberOfSwNo: Variant;

Service_NumberOfSwNo :=CreateOleObject ('VIGO.Std');
Led1 :=CreateOleObject ('VIGO.Std');
switch1 :=CreateOleObject ('VIGO.Std');

Service_NumberOfSwNo.PhysID := 'MC88C166.SERVICE.NUMBEROFSWNO';
Led1.PhysID := 'MC88C166.DIGITAL_IO_1.FLAGREG[7]';
switch1.PhysID := 'MC88C166.Digital_IO_5.FLAGREG[7]';

if switch1.Value = 0 then
begin
Led1.Value := 0;
end
else
begin
Led1.Value := 1;
end;
```

Beispiel 4.1: Zugriff auf physikalische Objekte in Delphi

4.2 Zugriff mit P-NET-Modulen

Der Zugriff auf den Slave mit P-NET-Modulen soll mit dem PD-4000-Controller gezeigt werden.
Der erste Teil zeigt wieder die nötigen Definitionen um auf den Slave überhaupt zugreifen zu können. Im zweiten Teil wird dies mit einem kurzen Process-Pascal-Programm verdeutlicht.

4.2.1 Moduldefinition in PROCESS-PASCAL

Um nun aber auf den 88C166-Slave zugreifen zu können, muß dieser in der Datei `PDModule.def`, im Unterverzeichnis `/inc` des Process-Pascal Verzeichnis, definiert werden.
Dadurch wird bei jedem Kompilieren eines Process-Pascal-Programms der Salve als Modul mit eingebunden. Beispiel 4.2 zeigt die notwendigen Zeilen die hinzugefügt werden müssen.

```
 MC88C166 = INTERFACE [ DeviceType: 3221; ObjectType = 1000;
Capabilities = NoBitAddress, NoOffsetInLong ] Service : ServiceCh;
Digital_IO_1 : DigitalCh;
Digital_IO_2 : DigitalCh;
Digital_IO_3 : DigitalCh;
Digital_IO_4 : DigitalCh;
Digital_IO_5 : DigitalCh;
Digital_IO_6 : DigitalCh;
CommonIO : CommonIO8Ch;
Analog_In_1 : AnalogInCh;
Analog_In_2 : AnalogInCh;
Current_Out : CurrentOutCh;
PID : PIDCh;
Calculator : CalculatorCh;
PulseProcessor: PulseProcCh;
END;
```

Beispiel 4.2: Moduldefinition in Process-Pascal

4.2.2 Beispiel in Process-Pascal

Der PD-4000-Controller ist ein P-NET-Master und ist ausgestattet mit einer Tastatur und einem Display. Der PD-4000 besitzt einen 256kB großen FLASH-EPROM für das Programm und ein 128kB großer RAM als Datenspeicher.
Programmiert wird der Controller in Process-Pascal [PD91]. Es gibt zwei Möglichkeiten des Downloads in den PD-4000: in der FLASH und in den RAM. Wird ein Download in den FLASH durchgeführt, so wird ebenfalls das Betriebsystem des Controllers ebenfalls neu installiert.
Mit dem Beispielprogramm kann das LED 1 gesetzt werden und der Status der Schalter abgefragt werden.

Die Datei DEMO.PP enthält das Hauptprogramm, in welchem folgende Zeilen hinzugefügt werden müssen:

```
MC88C166 : 88C166 AT NET: (1,5,2,73) NAME: 'MC88C166';
Light -> MC88C166.Digital_IO_1.FlagReg;
```

Diese Zeilen sind in FKEY4000.inc verhanden:

```
PROCEDURE FuncKeys(NewKey: BYTE);

BEGIN
CASE NewKey of
01: Light := true;
02: Light := false;
03: ;
08: ;
09: ;
10: ;
15: ;
16: ;
17: ;
22: ;
23: ;
24: ;
END; (* Case *)
END; (* FuncKeys *)
```

Kapitel 5

P-NET-Knoten - Hardware

Ziel dieses Kapitel ist es, die verwendete Hardware, wie Mikrocontroller-Board, P-NET-Interface und den angeschlossenen Hardwareapplikationen zu beschreiben. Jene drei Elemente wurden modular voneinander getrennt aufgebaut (vgl. 5.1) und samt Spannungsversorgungen in ein passendes PVC-Gehäuse integriert.

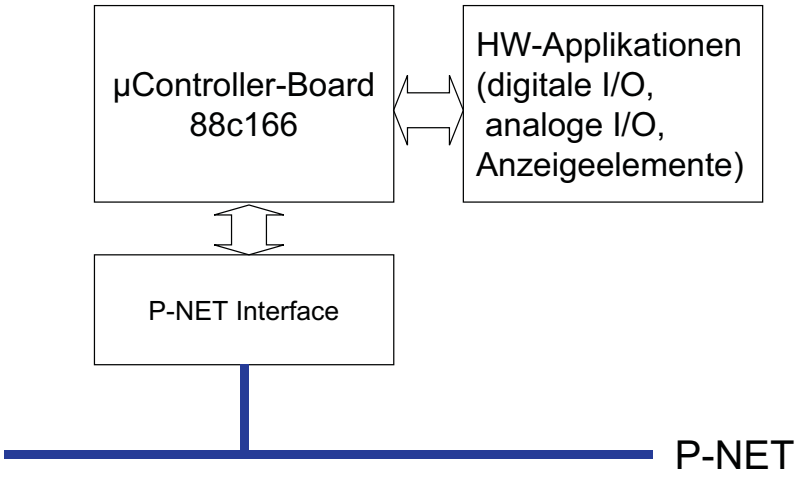


Abbildung 5.1: Hardware-Überblick

5.1 Das Mikrocontroller-Board

5.1.1 Technische Daten

Das Herz des Mikrocontroller-Boards ist ein 16-bit Mikrocontroller von SIEMENS – die 88C166-CPU aus der 80C166 Familie. Abbildung 5.2 zeigt eine Überblick der Funktionsblöcke. Folgend werden nochmals die Eckdaten der Mikrocontroller-Platine zusammengefaßt, genauere Information bitte ich in [KHM95, S.D95, Joh93] nachzulesen.

- 16-bit CPU mit einer 4-stufigen Pipeline
- 32KBytes Flash-Speicher
- On-Chip Bootstrap-Loader
- 100ns Befehlszyklus
- 7 Timer
- zwei serielle Schnittstellen
- 256K x 16bit-SRAM
- 10 Analog/digital-Wandler
- Stromsparfunktion

In der Entwicklungsphase hat sich der On-Chip-Bootstrap-Loader als sehr nützlich erwiesen, da innerhalb von wenigen Sekunden die Anwendung in den Speicher des Boards geladen werden kann. Der fertige Protokollstack wurde dann samt den PROM-, EEPROM-RPW- und Backup-Variablen (Beschreibung Speichertypen siehe Kapitel 2.3.1) in den FLASH geschrieben.

5.1.2 Portbelegung

Wie in den Abbildungen 5.2 und 5.3 gezeigt, besitzt der 88C166-Mikrocontroller sechs Ports. Von denen allerdings schon Port P0,P1 und P4 zur Ansteuerung des 256k-RAM belegt werden. Dennoch stehen mit P3, P2 und P5 noch zwei 16-Bit und ein 10-Bit breiter Port zur Verfügung. Vom 16-Bit breiten P3 werden jedoch 4 Pins für die beiden seriellen Schnittstellen S0, S1 und 2 Pins zur Ansteuerung des Speichers benötigt.
Daraus folgt doch noch eine beträchtliche Anzahl von 26 digitalen Ein-/Ausgänge.

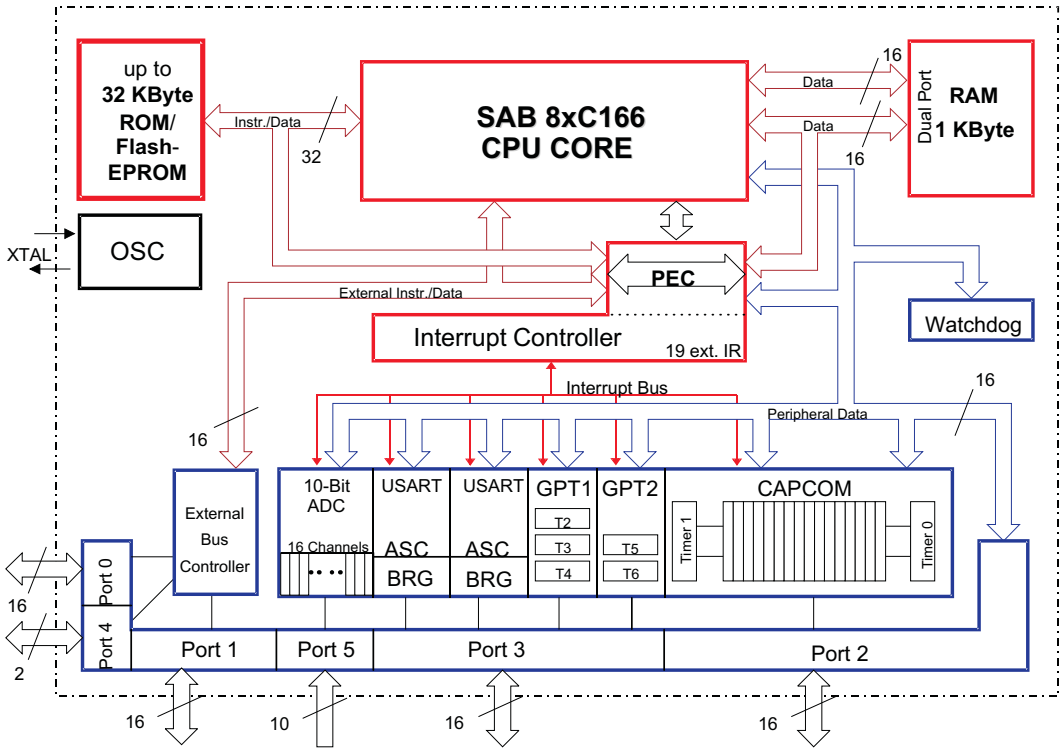


Abbildung 5.2: CPU-Übersicht

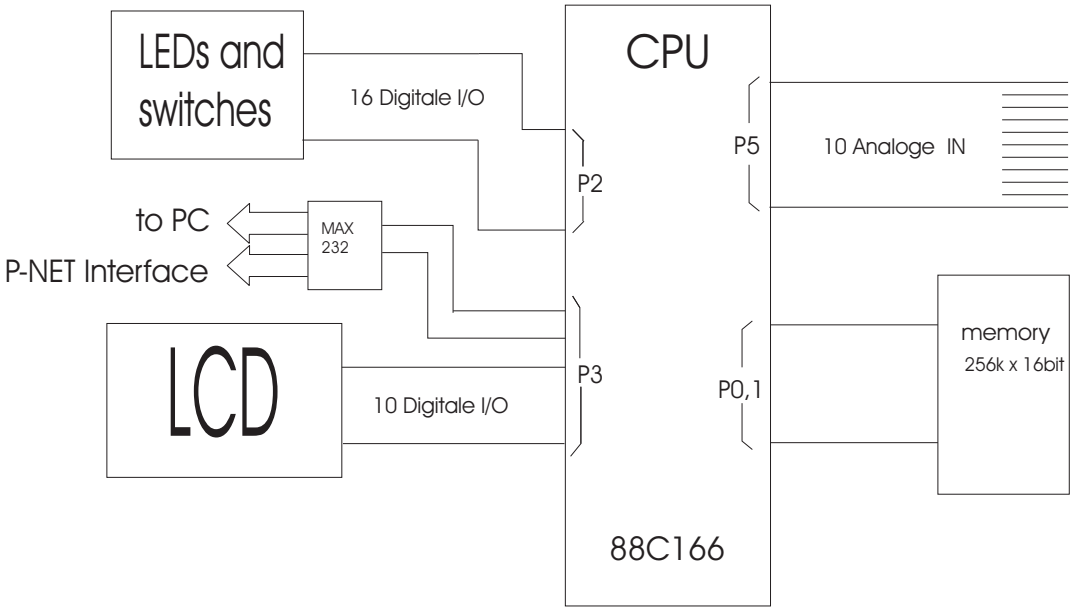


Abbildung 5.3: μC-Board

5.1.3 Takt

Bestimmt durch die etwas exotische Baudrate von 76800 Baud müßte ein Oszillator von 19,660800MHz verwendet werden ([Joh93] berechnet aus der unten stehenden Gleichung 5.1). Der 88C166-Mikrocontroller besitzt zwar einen 13bit-breiten-Reloadwert, dieser ist jedoch bei hohen Baudraten zu grob aufgelöst. Daraus folgt auch der unüblichen Oszillator-Wert.

$$B_{async0} = \frac{fosz}{64 * (SxREL + 1)} \tag{5.1}$$

B.....Baudrate
f_{OSZ}.....Frequenz des Quarzes
$SxREL$.....Reload-Wert
$mitSxBPS = 1$.....Steuerbit

Da das Mikrocontroller-Board aber einen Oszillator in SMD-Ausführung benötigt, welcher in 19,6608MHz nicht zu erhalten war, wurde ein 19,666660MHz-Oszillator verwendet. Bei Verwendung dieses Oszillators stellt sich eine Baudrate von 76823bit/s ein, die sich aber in den Toleranzgrenzen (+/-0,2% dies entspricht +/-154bit/s) befindet.

5.2 Das P-NET-Interface

Das P-NET-Interface dient zur Wandlung des RS232-Ausgangs des Mikrokontroller auf die RS485-Norm des P-NETs, wobei eine vollkommene galvanische Trennung zwischen der seriellen Schnittstelle des Mikrokontrollers (S1) und des P-NET-RS485 durchgeführt wurde (vgl. Abbildung Y.Z). Das Herzstück des Interfaces ist der Maxim-RS487 Baustein, welcher die Pegelanpassung durchführt. Um eine vollkommen galvanische Trennung durchführen zu können, mußte eine eigene Spannungsquelle den Teil der Schaltung versorgen, der nach den Optokoppler liegt.

5.3 Die Hardwareapplikationen

Aus Demonstrationszwecken wurden an das Mikrokontrollerboard verschiedene Ausgabe- und Eingabeeinheiten angeschlossen. Die Ausgabeeinheiten bestehen aus 8 LEDs und einem zweizeiligen LC-Display mit 16 Zeichen pro Zeile. Als Eingabeeinheiten wurden

vier Schalter implementiert. Die Ports wurden wie in Tabelle 5.1 und in Blockschaltbild 5.3 beschrieben belegt.

Bezeichnung	Portbelegung	Channel
LED 1	P2.0	Digital_IO_1
LED 2	P2.1	Digital_IO_1
LED 3	P2.2	Digital_IO_1
LED 4	P2.3	Digital_IO_1
LED 5	P2.4	—
LED 6	P2.5	—
LED 7	P2.6	—
LED 8	P2.7	—
Schalter 1	P2.8	Digital-IN_1
Schalter 2	P2.9	Digital-IN_2
Schalter 3	P2.10	—
Schalter 4	P2.11	—

Tabelle 5.1: Portbelegung

Die Portbelegung samt Anschlußbelegung an das μC-Board ist der Tabelle 7.1 im Anhang zu entnehmen.

Kapitel 6

Zusammenfassung

Der größte Aufwand war es, ein P-NET Protokoll-Analyse durchzuführen. Dabei half zunächst die Entwicklung mit einem P-NET-Monitorprogramm, welches auf einem normalen PC lief und über die RS232-Schnittstelle mit dem Mikrocontroller verbunden war. Doch zeigte sich, daß zwischen RS232 und RS485 einige Unterschiede vorhanden sind. Dies äußerte sich darin, daß beispielsweise das Adreßfeld im RS232 immer vom erweiterter Adreßtyp war, wobei jeweils die zweite Adresse keine Bedeutung hatte. Dies mag eventuell darauf zurückzuführen sein, daß das Monitor-Programm nur in einer Experimental-Version vorhanden war. Trotzdem konnten die Befehle Load und Store mit selbigen simuliert werden.
Beim Übergang von RS232 auf das P-NET stellte die größte Hürde die exotische Baudrate von 76800Baud dar. Dieser Umstand sollte allerdings durch den 13-Bit breiten Reloadwert keine Probleme aufwerfen, erwies sich jedoch anders. Durch die grobe Gliederung der Baudratenerzeugung im Bereich hoher Baudraten konnte auf Anhieb der Wert nicht erreicht werden. Erst durch das Tauschen des CPU-Oszillators konnte der Wert erlangt werden.

Zu erwähnen ist auch, daß die Protokoll-Implementierung weniger als 10kByte Speicherplatz im Mikrocontroller benötigt. Dies ist deshalb von so großer Bedeutung, als diese Größe nicht einmal 5% des zur Verfügung stehenden Speicherplatz und weniger als ein Drittel des 32kByte großen Flashspeichers einnimmt. Je nachdem ob der Protokollstack im FLASH oder im RAM abgelegt wird, stehen noch ausreichend Ressourcen zur Programmierung von eigenen Anwednungen zur Verfügung.

Abschließend sei noch ein Ausblick getätigt.
Folgende zusätzliche Implementierungen stellen sich aufgrund des bisher erwähnten als sinnvoll dar:

- watchdog-Funktion
- Implementierung eines eigenen Betriebssystems
- Erweiterung zum Master
- Entwicklung eines Download-Programms, welches die Programmierung des FLASH steuert.

Kapitel 7

Anhang

7.1 Diagramme

Folgende Diagramme werden im Gegensatz zu den vorigen Kapiteln nicht als SDL-Diagramme ausgeführt. Der Grund hierfür liegt darin, daß die einzelnen Funktionen mit Flußdiagrammen genauer erklärt werden können.
Die ersten beiden Abbildungen zeigen Flußdiagramme der Fehlerbehandlung im P-NET-Services. Darauf folgen die Diagramme für die Funktionen `get` und `set`.

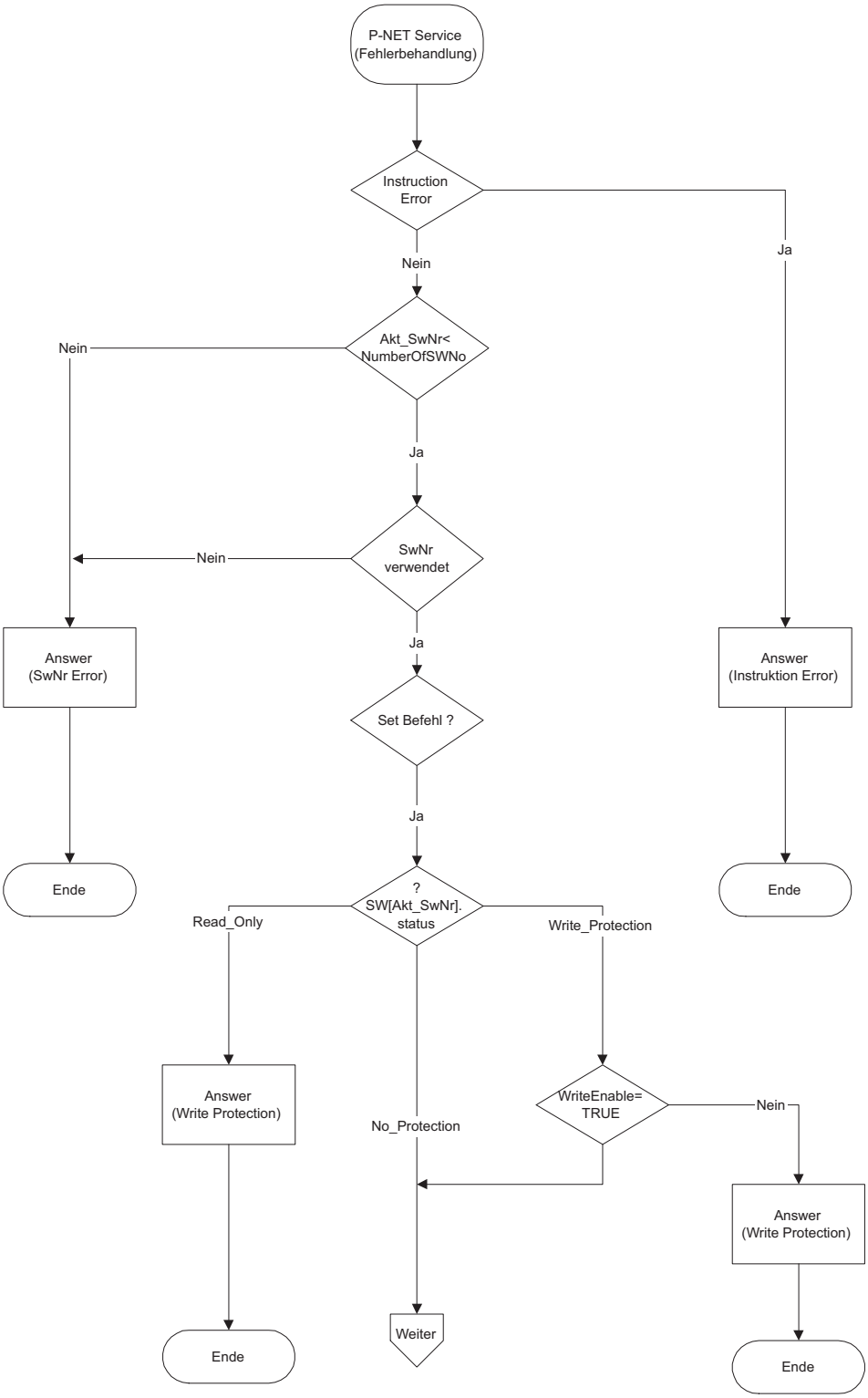


Abbildung 7.1: Flußdiagramm P-NET-Service (Teil 1)

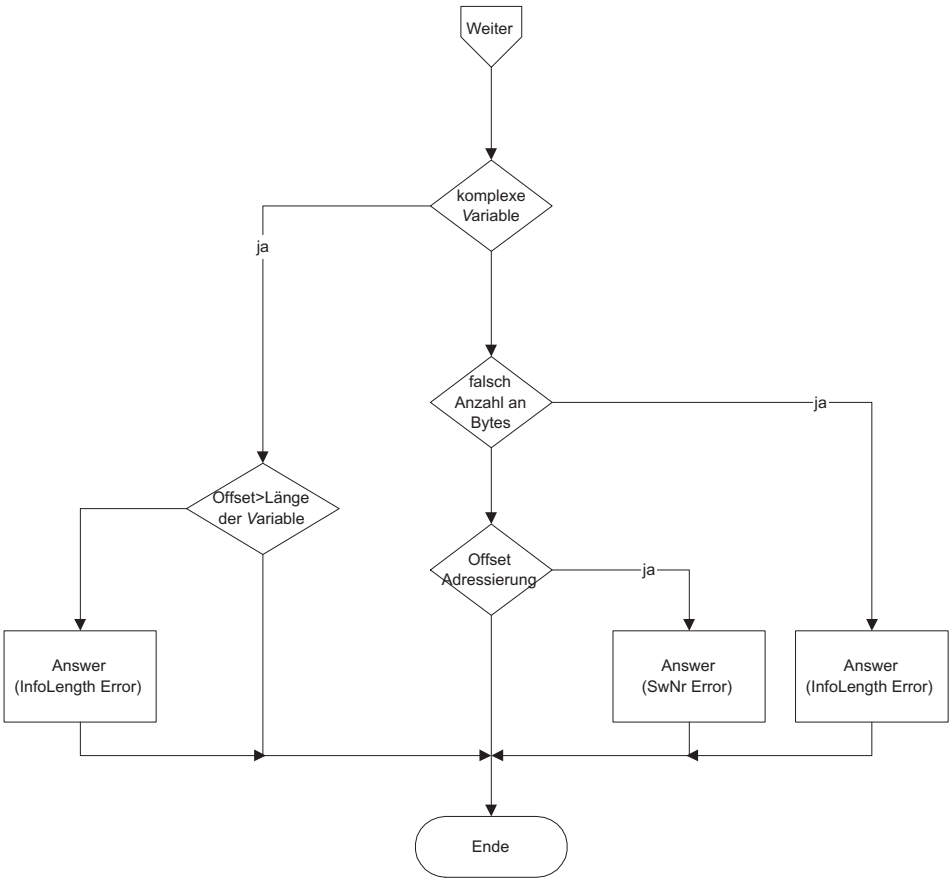


Abbildung 7.2: Flußdiagramm P-NET-Service (Teil 2)

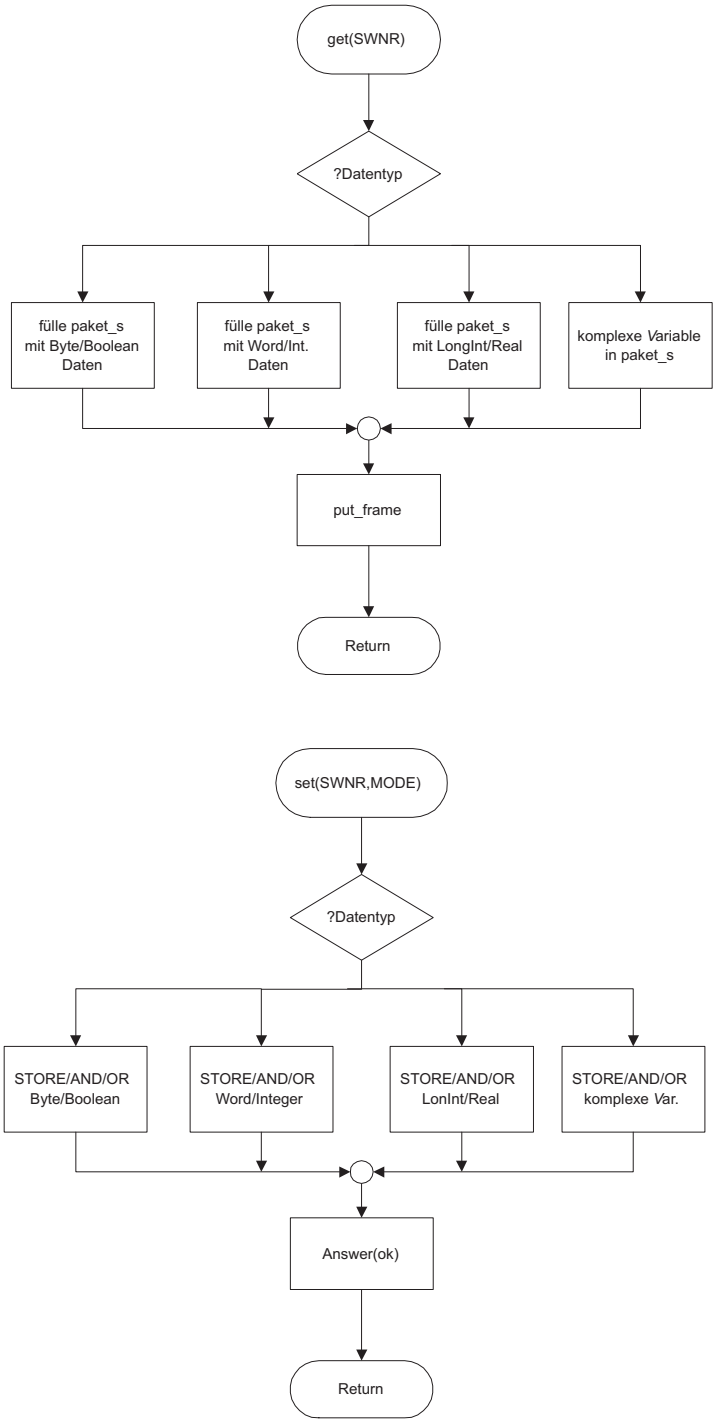


Abbildung 7.3: Flußdiagramme der Funktionen `get` und `set`

7.2 Wichtiger Quellcode

7.2.1 Sicherungsschicht

```
/*********************************************************/
/* Funktion get_frame:  liest einen Frame der fuer den   */
/*                      Knoten bestimmt ist ein und      */
/*                      uebergibt in die Struktur die    */
/*                      Destination und die Source Adr.  */
/*                      Ctrl/Status Bit und das Datenfeld */
/*********************************************************/
void get_frame(void)
{
frame=0;
state=0;
di=si=0;
  while (frame==0)
  {
    in=bt_get_word();
    in=in&0xFF;
    switch(state) {
      case 0: ip=0;
              if (in==ADR)    /* warte solange bis eigene Adresse im PNET */
              {               /* die Destination Adresse ist immer mit 8.BIT=0 gekennzeichnet*/
               S1CON=INIT_S1CON; /* umschalten von Wake_up in den normalen Empfang */
               frame_e.dest[di++]=in;
               sum_check(check_e,in);
               state=1;
              }
              else
              {
              if (in==0x7E)   /* Broadcast */
              {
               S1CON=INIT_S1CON; /* umschalten von Wake_up in den normalen Empfang */
               frame_e.dest[di++]=in;
               sum_check(check_e,in);
               state=1;
              }
              }
              break;
      case 1: /* bei RS232 wird der Extended Adress Type verwendet*/
              if (in<0x80)     /* wobei die 2. und 3. Adresse immer 70 und 80 sind */
              {
               frame_e.dest[di++]=in;
               sum_check(check_e,in);
              }
              else
              {
               frame_e.source[si++]=in;
               sum_check(check_e,in);
               state=2;
              }
              break;
      case 2:  /* die Source Adresse wird am 8. BIT =1 erkannt !!! */
              if (in>=0x80)
              {
               frame_e.source[si++]=in;
               sum_check(check_e,in);
              }
              else
```

```
            {
               frame_e.ctrl_stat=in;
               sum_check(check_e,in);
               state=3;
            }
            break;
      case 3:  /* lese Info Length aus */
            frame_e.info_len=in;
            sum_check(check_e,in);
            state=4;
            break;
      case 4:  /* uebernehme das Infofeld ip=info pointer        */
            if (ip<(frame_e.info_len&0x3F)-1) /* ACHTUNG: Das Feld Info Length */
                                              /* nur die unteren 6 Bit enthalten Laenge*/
            {                                 /*  wird mitgezaehlt und die unters
                                                  6 Bits = Anz.Bytes*/
             frame_e.info_field[ip++]=in;
             sum_check(check_e,in);
            }
            else                              /* Anzahl Bytes */
            {
             ip--;
             frame_e.num=in;
             sum_check(check_e,in);
             state=5;
            }
            break;
      case 5: /* lese Checksumme aus (die Summer der oben ermittelten*/
            frame_e.check=in; /* Summe (check_e) muss mit dieser CS 0 sein */
            state=0;
            frame=1;          /* Frame wurde erfolgreich gefunden */
            di=0;
            si=0;
            break;
    } /* switch */
  } /* while bis frame ok */
} /* get_frame*/

/*******************************************************/
/* Funktion put_frame:  speist einen Frame in das PNet  */
/*                      ein                             */
/*******************************************************/
void put_frame()
{
check_s=0;
/*   ---- Antwortframe  ---- */
frame_s.dest[0]=frame_e.source[0];
frame_s.source[0]=ADR;

/*   ---- Adressfeld ----- */

_putbit( 1, P2, 12 );   /* aktiviere PNET */

frame_s.dest[0]=frame_s.dest[0]|0x0100; /* sende Adreßbit */
putch_S1(frame_s.dest[0]);
    check_s=check_s+frame_s.dest[0];
frame_s.source[0]=frame_s.source[0];
putch_S1(frame_s.source[0]);
    check_s=check_s+frame_s.source[0];
/*   ---- Control/Status ---- */
putch_S1(frame_s.ctrl_stat);
    check_s=check_s+frame_s.ctrl_stat;
```

```
/*   ---- Infolaenge ----  */
putch_S1(frame_s.info_len);
    check_s=check_s+frame_s.info_len;
/*   ---- Infofeld ---- */
a=0;
while (a<frame_s.info_len)
{
putch_S1(frame_s.info_field[a]);
    check_s=check_s+frame_s.info_field[a];
a++;
}
/* ---- error detection ---- */
summe=check_s;
check_s=0-summe;    /* bilde Zweierkomplement */
check_s=check_s&0x00ff;
putch_S1(check_s); /* reduzierte Fehlererkennung */

_putbit( 0, P2, 12 );   /* deaktiviere PNET */

}
```

7.2.2 Transportsschicht

```
/*******************************************************/
/* Funktion Get: Realisiert den Load-Befehl            */
/*                                                     */
/* SWNR...auszulesende Softwirenummer                  */
/*                                                     */
/*******************************************************/

void Load(byte_t SWNR)
{
 int i;

offset=0;
if (frame_e.info_len > 0x3F )
{
 offset=frame_e.info_field[3];
}
Inhalt=(char*)SW[SWNR].point;
 switch (SW[SWNR].type)
 {
    case Boolean:
    case Byte:
            paket_s.info_field[0]=*Inhalt;
            paket_s.info_len=1;
            paket_s.ctrl_stat=frame_e.ctrl_stat;
            break;
    case Complex:
            /*------ String ------*/
            if (frame_e.num==0x15)
            { /* Stringbehandlung d.h. 1.Byte=Anzahl */
                paket_s.info_field[0]=SW[SWNR].NumbOfBytes-offset;
             for (i=0; i<frame_e.num; i++)
              {
              paket_s.info_field[i+1]=*(Inhalt+i+offset);
              }
            }
            else
```

```
            { /* Normale Behandlung der Variablen */
            switch (frame_e.num)
             {
              case 1: /* char,byte, boolean */
                paket_s.info_field[0]=*Inhalt;
                paket_s.info_len=1;
                break;
             case 2: /* Word bzw Integer */
                paket_s.info_field[0]=*(Inhalt+1+offset); /* Mikrocontroller speichert zuerst das LOW Byte*/
                paket_s.info_field[1]=*(Inhalt+offset);
                paket_s.info_len=2;
                break;
             case 4: /* Long Integer */
                paket_s.info_field[0]=*(Inhalt+1+offset);
                paket_s.info_field[1]=*(Inhalt+0+offset);
                paket_s.info_field[2]=*(Inhalt+3+offset);
                paket_s.info_field[3]=*(Inhalt+2+offset);
                paket_s.info_len=4;
                break;
             } /* switch */
            } /* if */
            paket_s.info_len=frame_e.num;
            paket_s.ctrl_stat=frame_e.ctrl_stat;
            break;
    case Word:
    case Integer:
            paket_s.info_field[0]=*(Inhalt+1+offset); /* Mikrocontroller speichert zuerst das LOW Byte*/
            paket_s.info_field[1]=*(Inhalt+offset);
            paket_s.info_len=2;
            paket_s.ctrl_stat=frame_e.ctrl_stat;
            break;
    case LongInteger:
                paket_s.info_field[0]=*(Inhalt+1+offset);
                paket_s.info_field[1]=*(Inhalt+0+offset);
                paket_s.info_field[2]=*(Inhalt+3+offset);
                paket_s.info_field[3]=*(Inhalt+2+offset);
                paket_s.info_len=4;
                paket_s.ctrl_stat=frame_e.ctrl_stat;
            break;
    case Real: break;
    case LongReal: break;
    case Timer: break;
    case 0:/* Not Data */
             paket_s.ctrl_stat=0x27;  /* Actual Data Error */
             paket_s.info_len=0;
           break;
    default: break;
  }
}

/*********************************************************/
/* Funktion Answer:      sendet einen der Fehlerliste     */
/*                      entsprechenden Antwortframe       */
/*********************************************************/

void Answer(byte_t code)
{

 switch (code)
 {
  case 0x00: /* ok */
         paket_s.ctrl_stat=frame_e.ctrl_stat;
```

```
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x01: /* Busy */
        paket_s.ctrl_stat=0x17;
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x02: /* Actual Data Error */
        paket_s.ctrl_stat=0x27;
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x03: /* Wait */
        paket_s.ctrl_stat=0x47;
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x0A: /* Data format error */
        paket_s.ctrl_stat=0x28;
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x0C: /* SwNr Error, SwNo existiert nicht im Channel */
        paket_s.ctrl_stat=0x30;
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x0E: /* Write Protection */
        paket_s.ctrl_stat=0x40;
        paket_s.info_len=0;
        put_frame();
        break;
  case 0x10: /* Instruction Error */
        paket_s.ctrl_stat=0x50;
        paket_s.info_len=0;
        put_frame();
        break;
  default:
        break;
 }
}
```

7.2.3 Service-Channel

```
/*****************************************************************************/
/* Variablen                                                                 */
/*****************************************************************************/

/* Service Channel */

    char NodeAdr;         /* Knotenadresse */
    char WriteEnable;     /* Wenn WriteEn. = TRUE(1) koennen Write protected Var.
                             beschrieben werden */
/* modified IO Channel */

    char IO_L1;           /* modified IO-Channel-1 Wert - LED1     */
    char IO_L2;           /* modified IO-Channel-2 Wert - LED2     */
    char IO_L3;           /* modified IO-Channel-3 Wert - LED3     */
```

```
    char IO_L4;           /* modified IO-Channel-4 Wert - LED4     */
    char IO_S1;           /* modified IO-Channel-1 Wert - Schalter */
    char IO_S2;           /* modified IO-Channel-2 Wert - Schalter */

/*****************************************************************************/
/* Strukturen                                                                */
/*****************************************************************************/
typedef struct DevID {
    unsigned int DeviceNumber;   /* */
    unsigned int ProgramVersion;
    unsigned int ManufacturerNo;
    char Manufacturer[20];
} DeviceIDD;


typedef struct PNETSer {
    char PNETNo;    /* Knoten Adresse */
    char dummybyte;
    char Serie[20]; /* Seriennummer vom Hersteller */
 } PNETSerD;

typedef struct ChTyp {
    unsigned int ChannelType;
    unsigned int Exist;
    unsigned int Functions;
} ChTypD;

typedef struct ComErr {
    char His;
    char dummybyte1;
    char Act;
    char dummybyte2;
    char ComHis;
    char dummybyte3;
    char ComAct;
    char dummybyte4;
} ComErrD;


/*****************************************************************************/
/* Deklaration der Konstanten und Strukturen                                 */
/*****************************************************************************/
const int NumberOfSWNo = 0x00CF;    /* Anzahl der Softwirenummern */

const struct DevID DeviceID = {                 /* SWNR 1 */
    0x0166, 0x0101, 0x1709, "Markus Haag e9325058" };

const struct PNETSer InitPNETSerNo = {          /* SWNR 4 */
    73, 00 ,"9325058" };

const struct ChTyp ChType = {
    1, 0xE013, 0 };

const struct ChTyp ChType_IO = {      /* ChannelType fuer IO Channel */
    2, 0xE205, 2 };                   /* Type=2; Bitfeld der belegten SwNr; Function=2 (=Output) */

/* --- Strukturen deklarieren --- */

struct PNETSer PNETSerialNo;

struct ComErr CommonError;    /* Service Channel */
```

```
struct ComErr CommonError_S1; /* Schalter 1*/

struct ComErr CommonError_S2; /* Schalter 2*/

struct ComErr CommonError_L1; /* LED 1 */

struct ComErr CommonError_L2; /* LED 2 */

struct ComErr CommonError_L3; /* LED 3 */

struct ComErr CommonError_L4; /* LED 4 */

/*****************************************************************************/
/* Struktur der  Softwire-Liste                                              */
/*****************************************************************************/

typedef struct SWList{
  char type;             /* Type der Variable:  0x00 Boolean; 0x01 Byte; 0x02 Word; 0x03 Integer; 0x04 Longint
                                                 0x05 Real; 0x06 LongReal; 0x07 Timer; 0x80 Complex */
  char status;           /* Status: 0x00 No protection; 0x10 Read Only; 0x20 Write protection*/
  void *point;           /* Zeiger auf die Variable, beinhaltet die physikalische Adresse */
  int  NumbOfBytes;      /* Anzahl der Bytes */
} SWL;

const struct SWList SW[208] = {
/*SW00*/  Integer, Read_Only, &NumberOfSWNo, 2,
/*SW01*/  Complex, Read_Only, &DeviceID, 32,
/*SW02*/  NotUsed,
/*SW03*/  NotUsed,
/*SW04*/  Complex, No_Protection, &PNETSerialNo, 21,
/*SW05*/  NotUsed,
/*SW06*/  NotUsed,
/*SW07*/  NotUsed,
/*SW08*/  NotUsed,
/*SW09*/  NotUsed,
/*SW0A*/  NotUsed,
/*SW0B*/  NotUsed,
/*SW0C*/  NotUsed,
/*SW0D*/  Boolean, No_Protection, &WriteEnable, 1,
/*SW0E*/  Complex, Read_Only, &ChType, 6,
/*SW0F*/  Complex, Read_Only, &CommonError, 4,
/* Begin der flexible Softwirelist z.B.: Digital I/O Channel */
........
```

7.3 Schaltpläne und Belegung der Anschlüsse

7.3.1 Mikrocontroller-Board

Abbildung 7.6 zeigt den Schaltplan des Mikrocontrollerboards. Die Jumper J2, J3 und J4 dienen zur Einstellung der Busmodi. Der Mikrocontroller wird im 16-Bit-Demultiplex-Modus betrieben. Dies bedeutet, daß der Speicher in Wort-Breite organisiert wird.

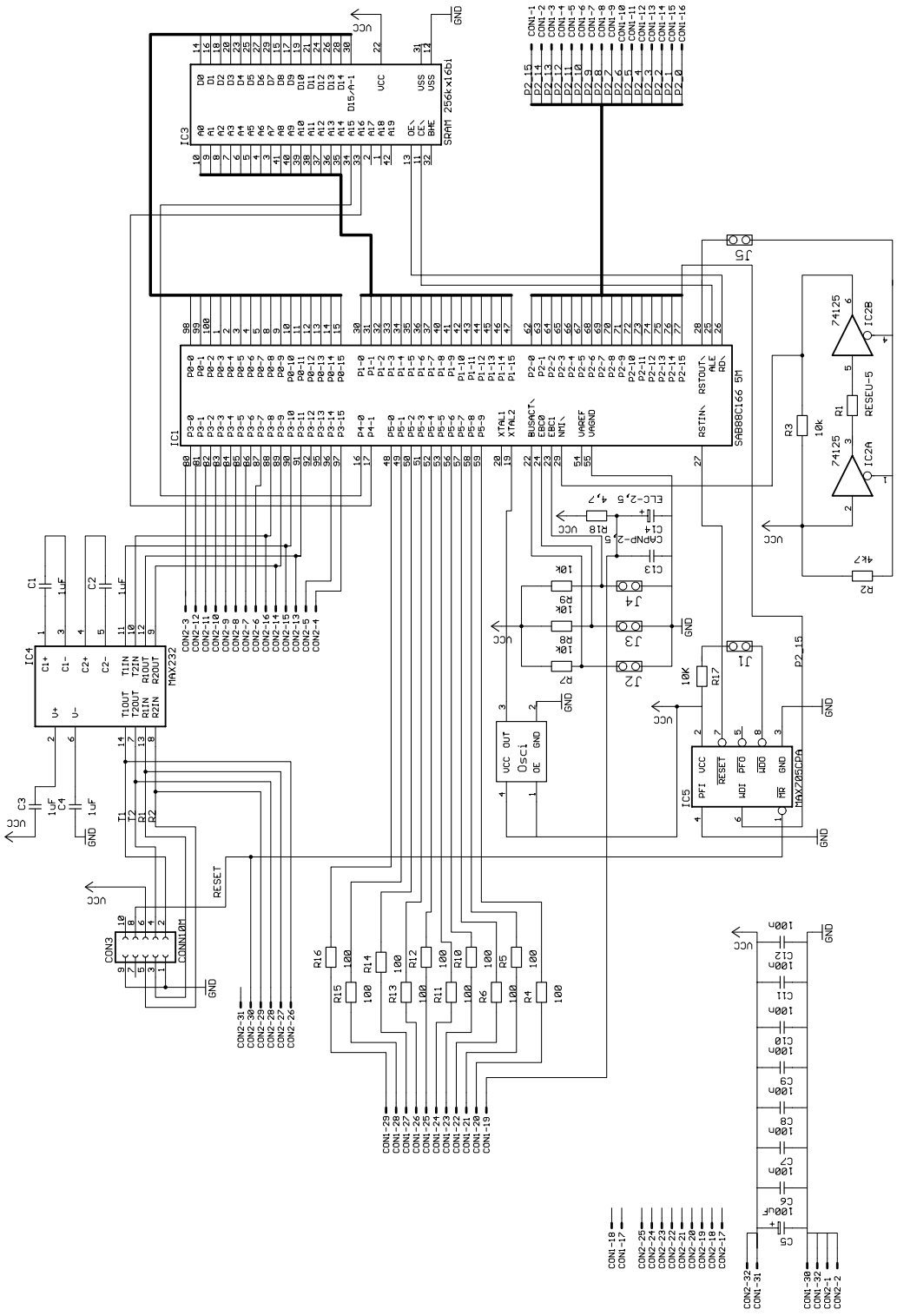


Abbildung 7.4: Schaltung Mikrocontroller-Board

7.3.2 Anschlußbelegung μC-Board

Tabelle 7.1 gibt die Belegung der Anschlüsse – die in Abbildung 7.6 dargestellt wurden – wieder.

Bezeichnung	Portbelegung	Anschluß
LED 1	P2.0	CON1-16
LED 2	P2.1	CON1-15
LED 3	P2.2	CON1-14
LED 4	P2.3	CON1-13
LED 5	P2.4	CON1-12
LED 6	P2.5	CON1-11
LED 7	P2.6	CON1-10
LED 8	P2.7	CON1-9
Schalter 1	P2.8	CON1-8
Schalter 2	P2.9	CON1-7
Schalter 3	P2.10	CON1-6
Schalter 4	P2.11	CON1-5
LCD		
D0	P3.0	CON2-13
D1	P3.1	CON2-12
D2	P3.2	CON2-11
D3	P3.3	CON2-10
D4	P3.4	CON2-9
D5	P3.5	CON2-8
D6	P3.6	CON2-7
D7	P3.7	CON2-6
R/W	P3.15	CON2-4
RS	P3.14	CON2-5
E	P2.13	CON1-3
P-NET-Interface		
A	weiß	Sub D Pin 9
GND	schwarz	Sub D Pin 5
B	grau	Sub D Pin 4

Tabelle 7.1: Anschlußbelegung

7.3.3 P-NET-Interface

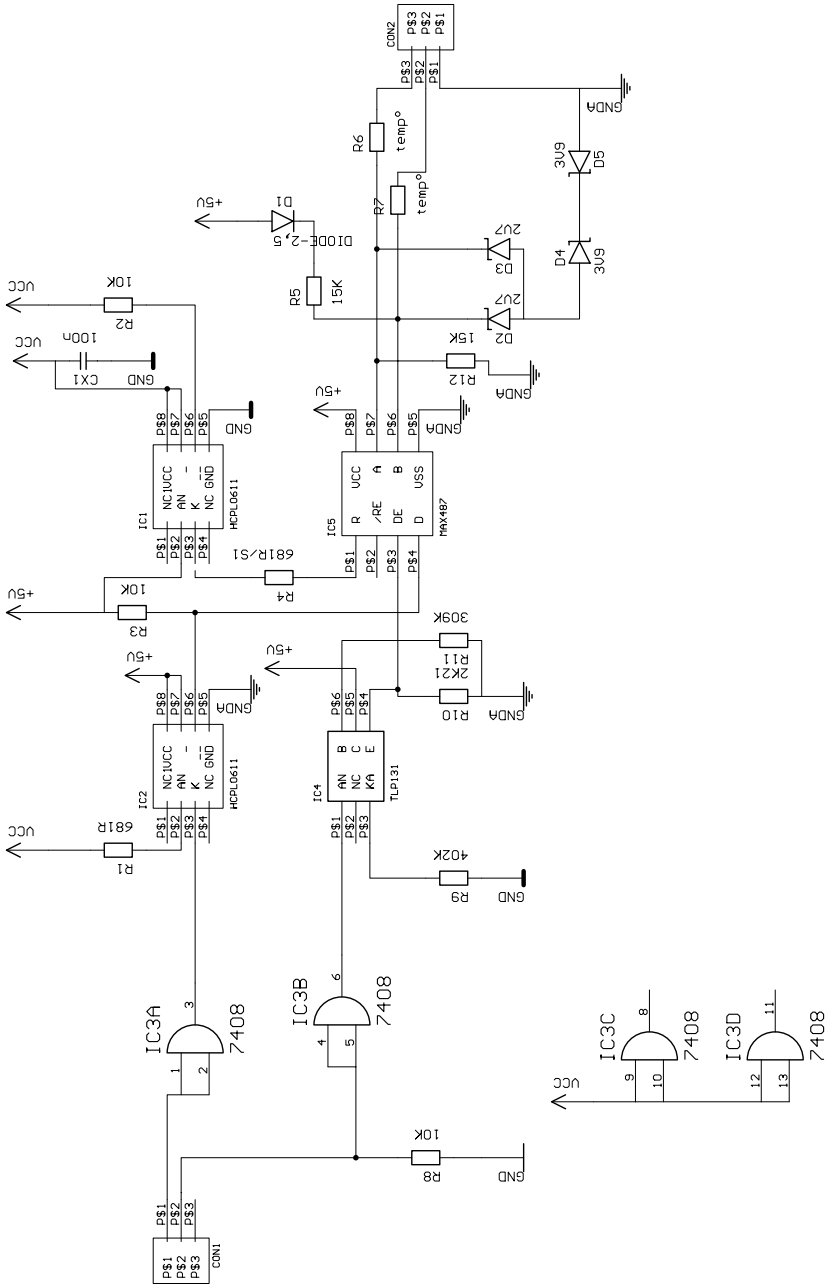


Abbildung 7.5: P-NET-Interface

7.3.4 Spannungsversorgung

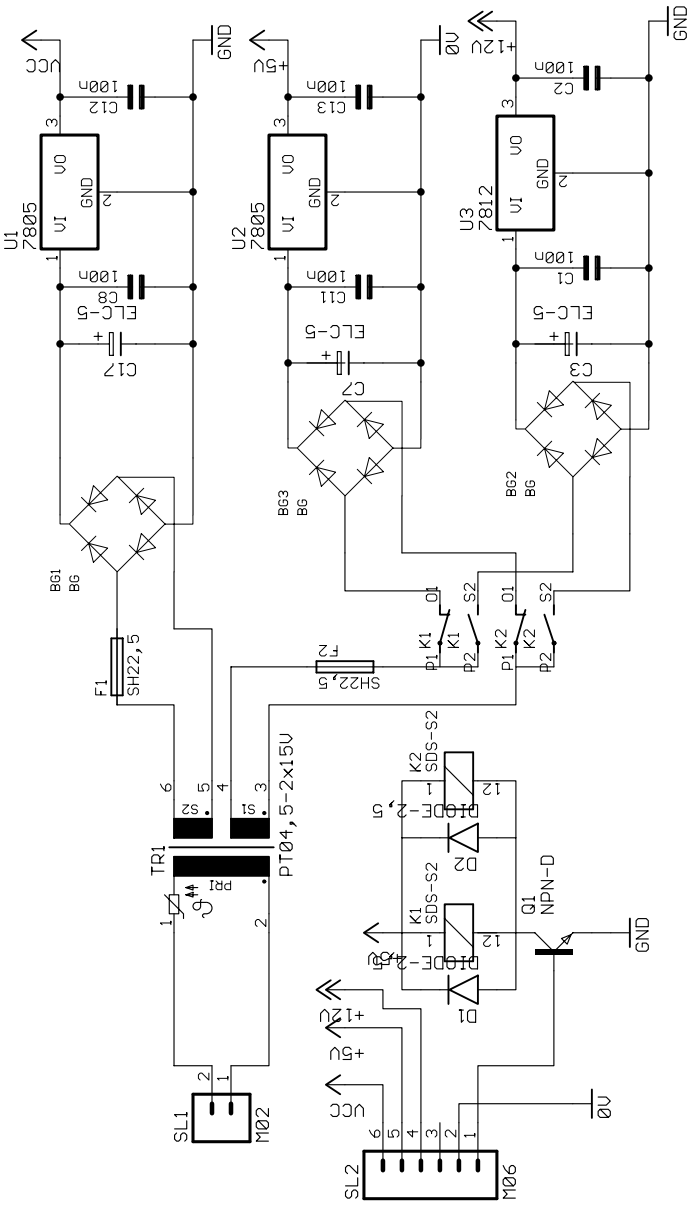


Abbildung 7.6: Spannungsversorgung

Literaturverzeichnis

[Bon95] Karl Walter Bonfig. *Feldbus-Systeme.* expert verlag, 2. auflage edition, 1995.

[Dat92] Process Data. *P-NET Standardized general purpose channel types.* International P-NET User Organization, 1992.

[Dat93] Process Data. *P-NET Standard.* International P-NET User Organization, 1993.

[DD98] Hans-Jörg Schweinzer Dietmar Dietrich, Dietmar Loy. *LON-Technologie Verteilte Systeme in der Anwendnung.* Hüthig, 1998.

[Die96] Dietmar Dietrich. *Bussysteme und Rechnerkommunikation.* Institut für Computertechnik - TU-Wien, 1996.

[Joh93] Reiner Johannis. *Handbuch des 80C166.* SIEMENS AG, 1993.

[Joh95] John Johansen. *General Purpose Fieldbus Objects.* Proces-Data, feldbustagung 95 edition, 1995.

[KHM95] Steffen Storandt Karl-Heinz Mattheis. *MC-Tools 17 - Arbeiten mit C166-Controllern.* Feger: Hardware- und Software-Verlag, 1995.

[MM96] Dietmar Dietrich Martin Manninger. *P-NET in Comparison with other Fieldbus System.* Institut für Computertechnik - TU-Wien, 4th p-net fieldbus conference edition, 1996.

[Nie95] Ole Cramer Nielson. *P-NET Protocol overview.* Proces-Data, feldbustagung 95 edition, 1995.

[Nie97] Jens Dalsgaard Nielson. *Comparison of P-NET with other networks.* Aalsborg University, DK, 5th p-net fieldbus conference edition, 1997.

[PD91] Proces-Data. *Process-Pascal v2.* Proces-Data, users manual edition, 1991.

[PD95] Proces-Data. *Universal Process Interface.* Proces-Data, manual edition, 1995.

[PD96] Proces-Data. *VIGO Users Manual*. Proces-Data, 1996.

[PD97] Proces-Data. *P-NET Controller PD4000 V2.2 and V2.3*. Proces-Data, users manual edition, 1997.

[Rei98] B. Reißenweber. *Feldbussysteme*. Oldenburg, 1998.

[Sch96] Gerhard Schnell. *Busssysteme in der Automatisierungstechnik*. Vieweg, 1996.

[S.D95] S.Dörrhöfer/J.Hofer. *Messe, Steuern und Regeln mit dem 80C166*. Franzis, 1995.

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Beispiele

Index