Masterarbeit

Evaluierung und Realisierung des Einsatzes von B&R-Steuerungssystemen als Kommunkationsslave in Feldbussen am Beispiel von Modbus-TCP

Michael Rieder

Institut für Technische Informatik
Technische Universität Graz
Vorstand: Römer, Kay Uwe, Univ.-Prof. Dipl.-Inform. Dr.sc.ETH

Begutachter: Brenner, Eugen, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.
Betreuer: Brenner, Eugen, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.

Graz, im Dezember 2013

Kurzfassung

Die Arbeit beschäftigt sich mit der Implementation eines Modbus TCP Slaves im Softwarepaket der Firma Bernecker + Rainer Industrie Elektronik Ges.m.b.H. Zu Beginn wird das Modbus-Protokoll analysiert, mit Augenmerk auf die Anwendung mit dem TCP/IP-Protokoll. Danach folgt eine Analyse des firmeneigenen Softwarepakets, bestehend aus Desktopanwendung und Echtzeit-Betriebsystem, wobei vor allem auf die Komponenten, welche für die Einbindung der neuen Funktionalität nötig sind, wert gelegt wird. Als nächstes werden die Anforderungen der Kunden analysiert und zusammengefasst. Aus den bis dahin gewonnenen Erkenntnissen wird ein Konzept für die Konfiguration des Slaves vorgestellt. Am Ende wird die eigentliche Implementation des Protokoll-Stacks in einem Gerätetreiber und dessen Einbindung in das System erläutert.

Abstract

This thesis is about the implementation of a Modbus TCP Slave in the software package of the company Bernecker + Rainer Industrie Elektronik Ges.m.b.H. At the beginning the Modbus-protocol is discussed with attention to the application together with the TCP/IP-network protocol. Afterwards an analysis of the proprietary software package is given, consisting of a desktop application and a real-time operating system, where mainly the components, which are important for the integration of the new functionality, are described. In the next step the requirements of the costumers are summarized and analysed. These premises are used to create a concept for the conguration of the Modbus TCP Slave. At the end the implementation of the protocol-stack and its integration into the system are explained.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz,am ....

(Unterschrift)

Danksagung

An dieser Stelle möchte ich mich bei Allen, die mir beim Erstellen dieser Arbeit geholfen haben bedanken. Als Erstes möchte ich mich bei der Firma Bernecker + Rainer Industrie Elektronik Ges.m.b.H. für die Möglichkeit diese Arbeit erstellen zu dürfen, bedanken. Im Besonderen gilt mein Dank Herrn Dipl. Ing. Heinz Fürnschuss und Thomas Würnschimmel. Vonseiten des Instituts hat mich Herr Professor Dr. Eugen Brenner unterstützt, auch ihm gilt mein besonderer Dank.
Am Schluss möchte ich mich noch bei meiner Familie bedanken, die mich sowohl finanziell wie auch moralisch zu jeder Zeit meines Studiums unterstützt hat. Im Rahmen dieser Arbeit hat mich vor allem meine Verlobte immer wieder aufgebaut und zum Weitermachen motiviert.

Graz, im Dezember 2013 Michael Rieder

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In der Automatisierungstechnik ist Kommunikation seit jeher ein zentrales Thema. Jede Maschine oder Prozess muss an mehreren Stellen überwacht und gesteuert werden. Bei kleineren Anwendungen kann eine direkte Verkabelung zur zentralen Steuereinheit noch zielführend sein. Ab einer gewissen Anzahl an zu überwachenden Sensoren und Aktoren muss sowohl aus logistischer als auch aus finanzieller Sicht an eine Kommunikation mittels Bus gedacht werden. Bei der Auswahl des richtigen Bussystems gibt es sehr viele Punkte die betrachtet und abgewogen werden müssen. Hier können als Beispiel die Übertragungsgeschwindigkeit und die maximale Anzahl an Busteilnehmer genannt werden, wobei eines der naheliegenden die Einfachheit der Anwendung ist. Es gibt eine Vielzahl an Bussen und es liegt auf der Hand, den zu wählen welcher am verständlichsten ist.

Modbus ist aufgrund seiner leichten Verständlichkeit und Anwendbarkeit sehr weit verbreitet und wird schon seit Jahrzehnten in der Automatisierungstechnik verwendet. Modbus liegt im OSI-Schichtenmodell auf der Anwendungsschicht und kann deshalb auf verschiedenen Übertragungsmedien angewandt werden.
Die Anwendung von Modbus über Ethernet ist aus heutiger Sicht nahezu eine Selbstverständlichkeit, da es überall zu finden ist und deshalb von einer Großzahl an Herstellern unterstützt wird. Heutzutage wird Modbus meist mit dem TCP/IP-Protokoll verwendet, diese Art der Anwendung ist unter dem Namen „Modbus TCP“ bekannt.

1.1 Motivation

Die breite Anwendung von Modbus in vielen Bereichen der Automatisierungstechnik macht es, für jeden Anbieter in diesem Bereich, nahezu zur Pflicht einige Formen zu unterstützten. Bei der Firma Bernecker + Rainer Industrie Elektronik Ges.m.b.H., kurz B&R, wurde bereits die Nutzung von Modbus über eine serielle Schnittstelle, sowie die Nutzung einer Steuerung als Modbus TCP Master unterstützt. Von Kundenseite kam aber bereits des Öfteren die Forderung, dass es möglich sein sollte eine B&R-Steuerung auch als Modbus TCP Slave zu konfigurieren.

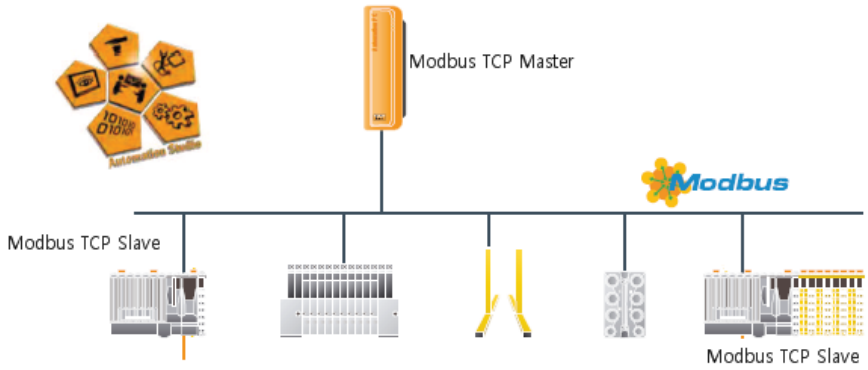

Abbildung 1.1: Mögliche Bus Konfiguration mit Modbus TCP

Diese Wünsche waren der Ausgangspunkt für diese Arbeit. Es sollte analysiert werden, wie die Konfiguration am Besten in die bereits vorhandenen Strukturen integriert werden kann, wobei darauf zu achten war, dass sich die Konfiguration an ähnlichen bereits existierenden Möglichkeiten orientiert, damit der Benutzer seine gewohnte Arbeitsweise mit B&R-Geräten beibehalten kann. In Abbildung 1.1 sieht man die Möglichkeiten, welche nach erfolgreichen Abschluss der Arbeit vorhanden sein sollten. B&R-Steuerung können dann sowohl als Modbus TCP Master, als auch als Modbus TCP Slave in Erscheinung treten. Durch die Vervollständigung der Modbus TCP Palette wird die Kommunikation mit Geräten von Fremdherstellern erleichtert und bietet so auch neue Märkte, da B&R-Steuerungen leichter in bereits bestehende Systeme integriert werden können.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, dass die neueste Generation von B&R-Steuerungen als Modbus TCP Slave fungieren können.
Dazu muss dem Anwender als Erstes eine angemessene Möglichkeit der Konfiguration gegeben werden. Darauf aufbauend muss sich die Steuerung gemäß der Modbus-Spezifikation verhalten. Um diese Ziele zu erreichen wurden einige Etappen festgelegt, mit deren Hilfe die Umsetzung Schritt für Schritt realisiert werden kann. In den folgenden Absätzen sind die Etappen aufgelistet.

- **Modbus**
  Die Modbus-Spezifikation wird analysiert, damit ein Verständnis für Modbus geschaffen wird. Im Speziellen wird dann die Anwendung von Modbus mit dem TCP/IP Protokoll untersucht.

- **Bestehendes System**
  Das bestehende Software-System wird untersucht, damit die neue Funktionalität gemäß den vorhandenen Struktur implementiert werden kann. Es ist darauf zu achten, dass die Software-Architektur nicht verletzt wird und die gegebenen Schnittstellen verwendet werden.

- **Design**
  Nach der Analyse des Modbus-Protokolls und des Bestandsystems ist es möglich, anhand der Anforderungen ein Design zu erstellen.

- **Implementierung**
  Die Implementierung soll die im Design beschriebenen Vorgaben umsetzten.

1.3 Gliederung

In **Kapitel 2** wird auf das Modbus-Protokoll eingegangen. Es wird aufgezeigt, dass die Kommunikation auf Funktionskodes basiert und dass es nach dem Master/Slave bzw. Client/Server-Prinzip arbeitet. Die in dieser Arbeit zu berücksichtigenden Funktionskodes werden genau definiert und müssen bei der Implementation berücksichtigt werden.

In **Kapitel 3** wird das bestehende Software-System analysiert, damit beim Erstellen des Design darauf geachtet werden kann, dass die Software-Architektur nicht verletzt wird. Die zu verwendenden Schnittstellen werden erläutert und können somit bei Erstellen des Designs und vor allem bei der Implementierung berücksichtigt werden.

Kapitel 4 beschreibt die Anforderungen und das daraus resultierende Design. Hier wird besonderes darauf geachtet, dass die Punkte aus Kapitel 3 berücksichtigt werden, damit sich die neue Komponente homogen in das System integrieren lässt.

Eine kurze Diskussion über die neuen Möglichkeit eine B&R-Steuerung als Modbus-TCP Slave zu konfigurieren, sowie einen Ausblick auf die weiteren Entwicklungen in diesem Bereich, bietet **Kapitel 5**.

Kapitel 2

Modbus

In diesem Kapitel wird zuerst kurz allgemein auf das Modbus-Protokoll eingegangen. Wann und wie es entstanden ist, was es auszeichnet und welche Eigenheiten es hat. Es werden die grundlegenden Mechanismen und die wichtigsten Befehle erklärt. Besonderes Augenmerk wird auf die Anwendung des Modbus Protokolls auf Basis des Netzwerkprotokolls TCP/IP gelegt, da diese in den folgenden Kapiteln genauer untersucht und implementiert wird.

2.1 Allgemein

Das Modbus-Protokoll gilt seit 1979 als defacto Standard in der Industrie und zählt zu den Feldbussen. Aufgrund seines einfachen Aufbaus und seiner einfachen Verwendung findet es großen Anklang in der Automatisierungstechnik. Das Protokoll befindet sich in der Applikationsschicht des OSI 7 Schichten Modells und kann dadurch auf vielen unterschiedlichen Übertragungsmedien aufbauen(siehe Abbildung 2.1).

Modbus arbeitet nach dem Client/Server- bzw. Master/Slave-Prinzip, d.h der Client sendet eine Anfrage an den Server, welcher diese auswertet und eine entsprechende Antwort zurück sendet. Jede Protocol Data Unit(kurz **PDU**), sowohl die Anfrage als auch die Antwort, beginnen mit einem Byte, welches den sogenannten Funktionskode enthält. Der Funktionskode könnte auch als Kommando-Id bezeichnet werden, anhand derer der Empfänger erkennt wie der Rest der Nachricht zu interpretieren ist. Es gibt 20 Funktionskodes, welche im Folgenden genauer erklärt werden. Die Größe einer gesamten Modbus-Application Data Unit (**ADU**) ist historisch bedingt auf 256 Byte begrenzt. Modbus wurde anfänglich vor allem in Verbindung mit seriellen Verbindungen wie RS232 oder RS485 verwendet und hier beträgt die maximale Paketgröße 256 Byte. Diese maximale Größe gilt auch auf anderen Übertragungsprotokollen, da es über Gateways möglich sein muss Modbus-Pakete über verschiedene Medien zu übertragen(siehe Abbildung 2.2).

Wie bereits erwähnt zählt Modbus zu den Feldbussen und wird hauptsächlich dazu eingesetzt eine zentrale Steuereinheit über einen Bus mit einem oder mehreren verteilten I/O-Knoten zu verbinden. Durch dieses Einsatzgebiet lassen sich die definierten Funktion-

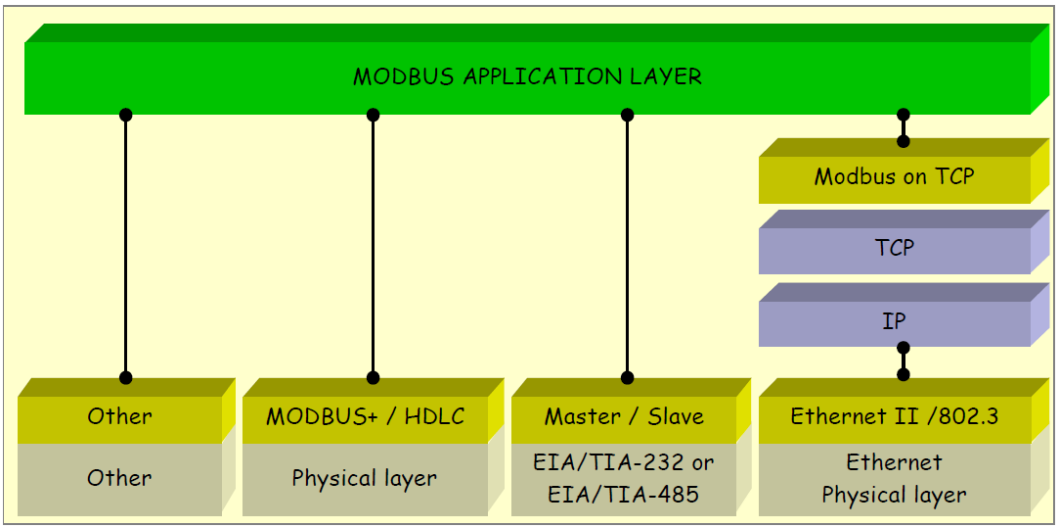

Abbildung 2.1: Modbus Schichtenmodell [Org13]

kodes und ihre Namen leicht erklären. Die Steuereinheit kann über den Bus bzw. über das Modbus-Protokoll auf Eingänge bzw. Ausgänge von I/O-Knoten zugreifen. Wenn es am Bus nur eine Steuereinheit gibt, greift diese auf Eingängen lesend und auf Ausgängen schreibend zu, sind aber mehrere Steuereinheiten mit dem Bus verbunden, muss auch ein lesender Zugriff auf Ausgänge möglich sein. Des Weiteren muss noch berücksichtigt werden, dass es zwei unterschiedliche Typen von I/O-Knoten gibt, zum einen digitale und zum anderen analoge. Daraus lassen sich nun die Bezeichnungen der einzelnen I/O-Punkte ableiten.
Coil steht bei Modbus für digitale Ausgänge, deshalb gibt es für sie Funktionskodes zum Schreiben und Lesen.
Als *Discrete Input* wird ein digitaler Eingang bezeichnet, für welchen es einen Funktionskode zum Lesen gibt.
Analoge Ein- und Ausgänge werden bei Modbus als Register bezeichnet und bestehen aus zwei Byte. Die Bezeichnung *Holding Register* steht für analoge Ausgänge, was bedeutet das Funktionskodes sowohl fürs Schreiben als auch fürs Lesen benötigt werden.
Input Register bezeichnen analoge Eingänge und es wird ein Funktionskode fürs Lesen benötigt.

2.2 Spezifikation

Die Kommunikation beim Modbus-Protokoll basiert, wie bereits erwähnt, auf Funktionskodes die in jedem Paket angegeben werden müssen. Da das Client/Server-Prinzip anwendung findet, geht jede Kommunikation vom Client aus. Dieser initiiert den Datenaustausch durch senden einer Anfrage an den Server. Das Format einer Anfrage ist durch

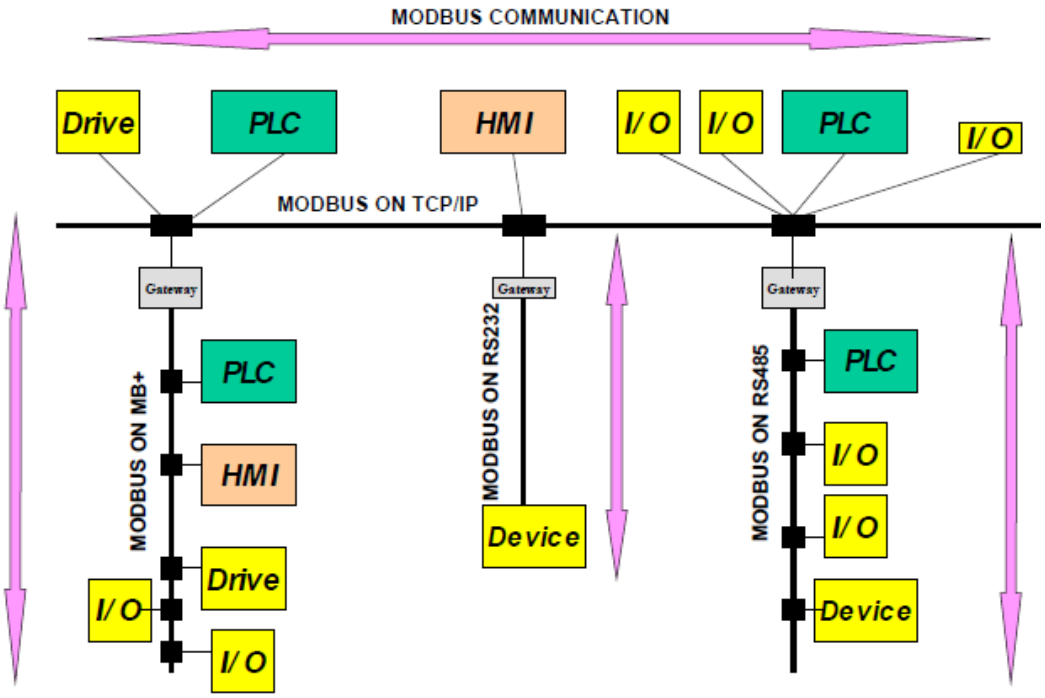

Figure 2: Example of MODBUS Network Architecture

Abbildung 2.2: Modbus Netzwerkarchitektur [Org13]

den verwendeten Funktionskode gegeben, das bedeutet jeder Funktionskode definiert den genauen Aufbau der Anfrage. Es gibt genau 20 Anfragen von denen einige oft und einige fast nie verwendet werden. Diese Klassifizierung lässt sich auch durch die Betrachtung von bereits auf dem Markt befindlichen Produkten untermauern. Viele Hersteller implementieren bei ihren Produkten nur die häufig verwendeten und für den jeweiligen Zweck nötigen Funktionskodes und sparen sich dadurch Entwicklungskosten, ohne dass der eingeschränkte Funktionsumfang auffällt. Aufgrund dieser Erkenntnis werden im Folgenden auch nur die für das Automation Runtime relevanten und zu implementierenden Funktionskodes bzw. Anfragen erklärt und behandelt.

2.2.1 Datenmodell

Das Datenmodell bei Modbus basiert auf vier Bezeichnern, welche man als Datentypen bezeichnen könnte. In Tabelle 2.1 findet man Aufstellung mit Namen, Größe und Zugriffsrechten.

Jeder der in Tabelle 2.1 genannten Datentypen hat seine eigenen Funktionskodes, wobei bei jedem ein Adresse von 0-65535 angegeben werden kann. Diese Umstände lassen nun eine Vielzahl von Möglichkeiten zu, wie der Modbus-Adressraum auf den Physikali-

Name	Größe	Zugriffsrecht
Coils	Single bit	lesen/schreiben
Discrete Inputs	Single bit	nur lesen
Holding Register	16-bit Wort	lesen/schreiben
Input Register	16-bit Wort	nur lesen

Tabelle 2.1: Auflistung der Modbus-Datentypen

schen Speicher angewendet wird. Um diese Möglichkeiten besser zu verstehen sieht man in Abbildung 2.3 und Abbildung 2.4 zwei mögliche Anwendungen. In Abbildung 2.3 werden alle vier Modbus Datentypen auf einen eigenen physikalischen Speicherbereich gelegt. In Abbildung 2.4 werden die boolschen(1-bit) Datentypen und die word(16-bit) Datentypen auf jeweils eigene Adressbereiche gelegt. Diese Zuweisung wird auch später bei der Implementation verwendet.

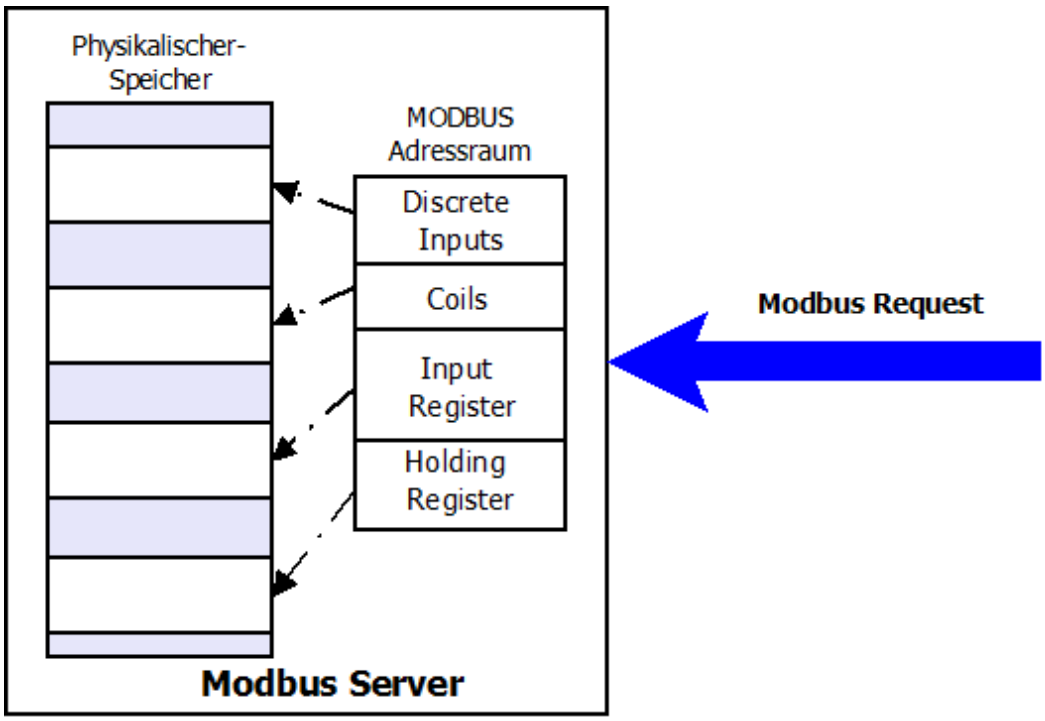

Abbildung 2.3: Modbus-Adresslayout Beispiel 1

2.2.2 Pakete

Das Modbus-Protokoll gibt einen allgemeinen Aufbau vor. In Abbildung 2.5 sieht man ein solches Paket. Die ADU ist applikationsspezifisch und ändert sich je nach dem über welches Medium kommuniziert wird und in welches andere Protokoll die Modbus PDU verpackt wird. Die PDU selbst beginnt immer mit dem Funktionskode und anschließend kommen

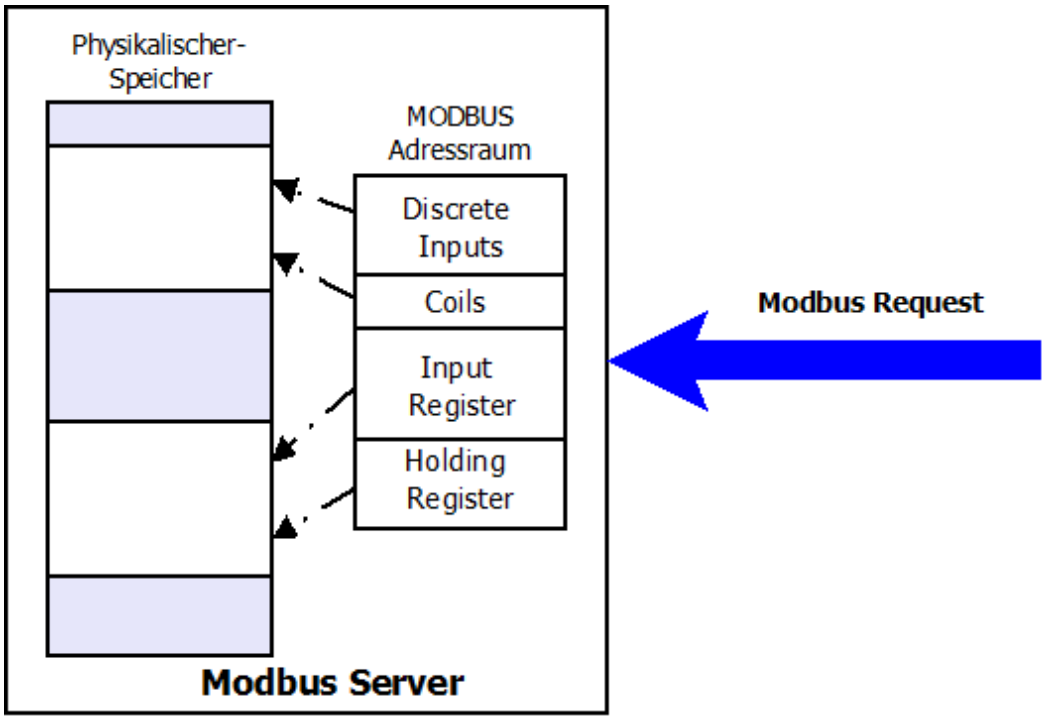

Abbildung 2.4: Modbus-Adresslayout Beispiel 2

die vom Funktionskode abhängigen Daten. Um die Daten richtig zu interpretieren muss noch eine Byte-Reihenfolge festgelegt werden. Diese ist bei Modbus immer Big-Endian, d.h. das erste empfangene Byte ist das höchstwertige [Coh81].

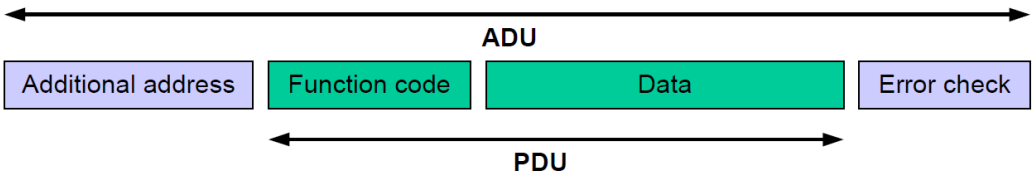

Abbildung 2.5: Allgemeines Modbus Paket [Org13]

Bei einer Modbusübertragung gibt es drei Pakete. Diese sind:

- Anfrage-Paket (Request)
- Antwort-Paket (Response)
- Fehler-Paket (Exception-Response)

Um eine Kommunikation zu beginnen, sendet der Client einen Request an den Server. Dieser wertet den Request aus und führt, je nachdem, eine Lese- oder Schreib-Operation aus. Danach wird ein entsprechendes Response-Paket erstellt und an den Client zurück

gesendet. Tritt beim Ausführen der Operation ein Fehler auf, wird eine dem Fehler entsprechende Exception-Response zurück gesendet.

2.2.3 Funktionskodes

Jede Anfrage des Client an den Server beginnt mit einem Funktionskode. Anhand dieses Funktionskode weiß der Server wie er die folgenden Daten interpretieren soll. Tritt bei der Auswertung des Requests ein Fehler auf, so muss der Server den ganzen Request verwerfen und eine Exception-Response senden. Das Modbus-Protokoll spezifiziert 21 Funktionskodes, von denen 5 nur bei einer serielle Verbindung Anwendung finden. Des Weiteren können einige Funktionskodes für spezielle Anwendungen weggelassen werden, was uns zu den, in der Tabelle 2.2, dargestellten Funktionskodes führt. All diese Modbus-Funktionskodes sollen im Rahmen dieser Diplomarbeit implementiert werden. In der Folge müssen diese Anfragen noch genauer spezifiziert und erklärt werden, insbesondere welche Antwort auf eine entsprechende Anfrage gegeben werden muss.

Modbus Funktionskode	**Bezeichnung**	**Beschreibung**
0x01	Read Coils	Digitale Ausgänge lesen
0x02	Read Discrete Inputs	Digitale Eingänge lesen
0x03	Read Holding Registers	Analoge Ausgänge lesen
0x04	Read Read Input Registers	Analoge Eingänge lesen
0x05	Write Single Coil	Digitalen Ausgang schreiben
0x06	Write Single Register	Analogen Ausgang schreiben
0x0F	Write Multiple Coils	Digitale Ausgänge schreiben
0x10	Write Multiple Registers	Analoge Ausgänge schreiben
0x17	Read/Write Multiple Registers	Analoge Ausgänge lesen und schreiben

Tabelle 2.2: Auflistung der zu implementierenden Funktionskodes

FC 1: Read Coils

Mit diesem Funktionskode ist es möglich bis zu 2000 hintereinander liegende Coils von einem Server zu lesen. Im Request(siehe Tabelle 2.3) wird eine Startadresse und eine Anzahl an zu lesenden Coils angegeben. Die Adressierung beginnt bei 0.
Die Antwort (siehe Tabelle 2.4) auf so einen Request beginnt mit dem Funktionkode, dann folgt ein Byte in dem steht wieviele Datenbytes folgen werden. Die Datensektion enthält die Status der einzelnen Coils, bitweise gepackt. Eine Coil kann entweder EIN(1) oder AUS(0) sein. Das niederwertigste Bit (**LSB**) des ersten Datenbytes enthält den Status der ersten Coil, die in der Anfrage angegeben wurde. Die nächsten sieben Coils finden sich aufsteigend im ersten Datenbyte wieder. Die neunte Coil ist wiederum das LSB des nächsten Datenbyte.

Sollte die Anzahl an zu lesenden Coils nicht ein Vielfaches von 8 sein, so ist das letzte Byte mit 0 aufzufüllen.

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x01**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Coils	2 Byte	1 - 2000 = **n**

Tabelle 2.3: FC 1: Request Aufbau

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x01**
Anzahl an Datenbytes	1 Byte	**N**= **n**/8 (+1)
Coils Status	**N** Byte	

Tabelle 2.4: FC 1: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes(siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 2: Read Discrete Inputs

Mit diesem Funktionskode ist es möglich bis zu 2000 hintereinander liegende Discrete Inputs von einem Server zu lesen. Der Request(siehe Tabelle 2.5) ist gleich aufgebaut wie beim Funktionskode 1. Die Adressierung beginnt wiederum bei 0.
Auch die Antwort (siehe Tabelle 2.6) ist bis auf das Funktionskode-Feld gleich wie beim Funktionkode 1. Ein Discrete Input hat die selben Status wie eine Coil und auch diese werden gepackt in der Datensektion übertragen.
Sollte die Anzahl an zu lesenden Discrete Inputs nicht ein Vielfaches von 8 sein, so ist das letzte Byte mit 0 aufzufüllen.

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x02**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Discrete Inputs	2 Byte	1 - 2000 = **n**

Tabelle 2.5: FC 2: Request Aufbau

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x02**
Anzahl an Datenbytes	1 Byte	**N**= **n**/8 (+1)
Discrete Input Status	**N** Byte	

Tabelle 2.6: FC 2: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 3: Read Holding Registers

Mit Funktionskode 3 kann ein zusammenhängender Block an Holding Register gelesen werden. Der Request (siehe Tabelle 2.7) gibt die Startadresse und die Anzahl an zu lesenden Registern an.
In der Datensektion der Antwort (siehe Tabelle 2.8) wird der Wert eines Register in jeweils zwei Byte geschrieben, wobei das erste Byte das höherwertige ist (Big-Endian).

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x03**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Holding Registers	2 Byte	1 - 125 = **N**

Tabelle 2.7: FC 3: Request Aufbau

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x03**
Anzahl an Datenbytes	1 Byte	2 ***N**
Holding Register Wert	2***N** Byte	

Tabelle 2.8: FC 3: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 4: Read Input Registers

Der Funktionskode 4 wird eingesetzt um Werte von Input Registern zu lesen. Der Request (siehe Tabelle 2.9) ist hier bis auf das Funktionskode-Feld gleich wie bei Funktionskode 3. Auch hier können nur zusammenhängende Register-Blöcke gelesen werde.
Die Antwort (siehe Tabelle 2.10) ist auch bis auf den Funktionskode gleich.

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x04**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Input Registers	2 Byte	1 - 125 = **N**

Tabelle 2.9: FC 4: Request Aufbau

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x04**
Anzahl an Datenbytes	1 Byte	2 ***N**
Input Register Wert	2***N** Byte	

Tabelle 2.10: FC 4: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 5: Write Single Coil

Mit Funktionskode 5 kann der Status einer Coil auf dem Server auf EIN(1) oder aUS(0) gesetzt werden. Der Request (siehe Tabelle 2.11) besteht auch hier aus einem Funktionskode-Feld und einer Adresse. Hinzu kommen noch zwei Byte in denen der zu setzende Status steht. Sind beide Bytes 0x00 dann wird die Coil auf den Zustand AUS gesetzt, ist das Erste der beiden Bytes auf 0xFF und das zweite auf 0x00 wird die Coil auf EIN gesetzt. Die Antwort (siehe Tabelle 2.12) ist im Erfolgsfall ein Echo des Requests.

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x05**
Startadresse	2 Byte	0x0000 - 0xFFFF
Zu setzender Status	2 Byte	0x0000 oder 0xFF00

Tabelle 2.11: FC 5: Request Aufbau

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x05**
Startadresse	2 Byte	0x0000 - 0xFFFF
Zu setzender Status	2 Byte	0x0000 oder 0xFF00

Tabelle 2.12: FC 5: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 6: Write Single Register

Dieser Funktionskode wird verwendet um am Server ein einzelnes Register zu beschreiben. Die Request-PDU (siehe Tabelle 2.13) gibt die Adresse des Registers und den zu schreibenden Wert an.
Die Antwort (siehe Tabelle 2.14) ist im Erfolgsfall ein Echo des Requests.

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x06**
Startadresse	2 Byte	0x0000 - 0xFFFF
Zu schreibender Wert	2 Byte	0x0000 - 0xFFFF

Tabelle 2.13: FC 6: Request Aufbau

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x06**
Startadresse	2 Byte	0x0000 - 0xFFFF
Zu schreibender Wert	2 Byte	0x0000 - 0xFFFF

Tabelle 2.14: FC 6: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 15: Write Multiple Coils

Mit diesem Funktionskode lassen sich mehrere hintereinander liegende Coils beschreiben. Im Request (siehe Tabelle 2.15) wird die Startadresse, die Anzahl an Registern und die Status der Coils angegeben. Zusätzlich muss noch angegeben werden, wie viele Bytes die Status benötigen. Dies ergibt sich aus $\mathbf{N} = \lceil \frac{Anzahl\ an\ Coils}{8} \rceil$. Die Datensektion enthält die Status der einzelnen Coils, bitweise gepackt. Das LSB des ersten Datenbytes enthält den Status der ersten Coil. Die nächsten sieben Coils finden sich aufsteigend im ersten Datenbyte wieder. Die neunte Coil ist wiederum das LSB des nächsten Datenbyte.

Die Antwort (siehe Tabelle 2.16) enthält den Funktionskode die Startadresse und die Anzahl an geschriebenen Coils.

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x0F**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Coils	2 Byte	0x0000 - 0x07B0
Anzahl an Datenbytes	1 Byte	**N**
Zu setzende Status	**N** Byte	0x0000 - 0xFFFF

Tabelle 2.15: FC 15: Request Aufbau

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x0F**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Coils	2 Byte	0x0000 - 0x07B0

Tabelle 2.16: FC 15: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 16: Write Multiple registers

Dieser Funktionsblock wird verwendet um einen zusammenhängenden Block an Registern zu beschreiben.
Der Request (siehe Tabelle 2.17) gibt eine Startadresse, eine Anzahl an Registern und die entsprechenden Werte an. Zusatzlich muss noch in einem Byte angegeben werden wie groß die Datensektion ist.
Die Antwort (siehe Tabelle 2.18) enthält den Funktionskode die Startadresse und die Anzahl an geschriebenen Registern.

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x10**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Registern	2 Byte	0x0000 - 0x007B
Anzahl an Datenbytes	1 Byte	2***N**
Zu schreibende Werte	2***N** Byte	0x0000 - 0xFFFF

Tabelle 2.17: FC 16: Request Aufbau

Beschreibung	Größe	Werte
Funktionskode	1 Byte	**0x10**
Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an Registern	2 Byte	0x0000 - 0x007B

Tabelle 2.18: FC 16: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

FC 23: Read/Write Multiple registers

Dieser Funktionskode ist eine Kombination aus Lese- und Schhreiboperation. Das Schreiben wird vor dem Lesen ausgeführt.

Der Request (siehe Tabelle 2.19) gibt zuerst eine Startadresse und eine Anzahl an Registern für die Leseoperation an. Dann folgt das gleiche für die Schreiboperation und zusätzlich müssen noch die zu schreibenden Werte angegeben werden.
Die Antwort (siehe Tabelle 2.20) besteht aus den gelesenen Werten und deren Größe. Wenn das Schreiben oder das Lesen fehlschlägt so darf keines der beiden ausgeführt werde. Das heißt, wenn der Schreibbefehl korrekt ausgeführt werden könnte, aber der Lesebefehl eine falsche Adresse angibt, darf weder gelesen noch geschrieben werden und es muss eine Exception-Response gesendet werden.

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x17**
Lese Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an zu lesenden Registern	2 Byte	0x0001 - 0x007D = $\mathbf{N}_r$
Schreib Startadresse	2 Byte	0x0000 - 0xFFFF
Anzahl an zu schreibenden Registern	2 Byte	0x0001 - 0x0079= $\mathbf{N}_w$
Anzahl an Datenbytes	1 Byte	2*$\mathbf{N}_w$
Zu schreibende Werte	2*$\mathbf{N}_w$ Byte	0x0000 - 0xFFFF

Tabelle 2.19: FC 23: Request Aufbau

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	**0x17**
Anzahl an Datenbytes	1 Byte	0x0000 - 0xFFFF
Registerwerte	2*$\mathbf{N}_r$ Byte	

Tabelle 2.20: FC 23: Response Aufbau

Tritt beim Auswerten bzw. Ausführen des Requests ein Fehler auf, können die Exceptionkodes (siehe Kapitel 2.2.4) 1-4 zurück geliefert werden.

2.2.4 Exceptions

Bei der Kommunikation mit Modbus können natürlich Fehler auftreten. Es gibt in der Regel folgende vier Fälle:

- Der Client sendet einen Request und der Server empfängt diesen fehlerfrei. Er führt ihn aus und sendet eine entsprechende Antwort.
- Der Client sendet einen Request an den Server, dieser empfängt diesen aber nicht. Dies kann zum Beispiel passieren, wenn die Leitung zwischen den Beiden defekt ist. In diesem Fall wird am Client ein Timeout ausgelöst und eine entsprechende Maßnahme ergriffen.

- Der Request wird vom Server empfangen, jedoch schlägt die Prüfsummenüberprüfung fehl bzw. es wird ein unvollständiger Request empfagen. In diesem Fall macht der Server nichts und der Client reagiert nach dem Timeout.
- Der Request wird korrekt empfangen, jedoch kann das angegebene Kommando nicht ausgeführt werden. Dies könnte zum Beispiel der Fall sein, wenn die angegeben Adresse am Server nicht existiert. In diesem Fall antwortet der Server mit der entsprechenden Exception.

Eine Exception-Antwort hat immer den selben Aufbau (siehe Tabelle 2.21).

Beschreibung	**Größe**	**Werte**
Funktionskode	1 Byte	Funktionskode des fehlerhaften Request mit gesetztem MSB
Exception-Kode	1 Byte	siehe Tabelle 2.22

Tabelle 2.21: Exception Response

Wert [HEX]	**Name**	**Beschreibung**
01	Illegale Funktion	Empfangener Funktionskode wird nicht unterstützt.
02	Illegale Adresse	Die angegebene Adresse ist auf dem Server nicht verfügbar.
03	Illegaler Wert	Der angegebene Wert ist ungültig. Dies kann auftereten, wenn der zu schreibende Wert außerhalb des zulässigen Bereichs liegt, aber auch wenn andere übergebene Werte, wie zum Beispiel Längenangaben nicht korrekt sind.
04	Server Fehler	Am Server ist beim Ausführen des Befehls ein Fehler aufgetreten.
0A	Gateway Pfad-Fehler	Dieser Fehler kann auftreten, wenn der Server als Gateway zu einem Modbus-Netzwerk verwendet wird. In diesem Fall bedeutet der Fehler, dass der Server nicht auf das andere Netzwerk zugreifen kann.
0B	Gateway Fehler	Dieser Fehler kann auftreten, wenn der Server als Gateway zu einem Modbus-Netzwerk verwendet wird. Er bedeutet, dass das gesuchte Gerät im Netzwerk nicht erreichbar ist.

Tabelle 2.22: Exception-Kodes

2.3 Modbus über TCP/IP

Das Modbus-Protokoll kann auch mit dem weit verbreitete Netzwerkprotkoll TCP/IP verwendet werden. In diesem Fall wird häufig von „Modbus TCP“ gesprochen. Hierzu werden die Modbus ADUs als Payload eines TCP-Paketes versendet. Bei dieser Anwendung von Modbus besteht die anwendugsspezifische ADU aus einem Modbus Application Protocol (**MBAP**)-Header und der Modbus-PDU. In Abbildung 2.6 ist der Aufbau eines Paketes dargestellt.
Der Header hat, im Falle der Verwendung mit TCP/IP, die in der Tabelle 2.23 dargestellten Felder. Der *Transaction-Identifier* wird vom Client gesetzt und vom Server in der Antwort übernommen, damit der Client diese dem entsprechenden Request zuordnen kann. Der *Protocol-Identifier* ist bei Modbus immer 0. Die Länge gibt an wie viele Bytes noch folgen. Dies ist notwendig, wenn eine Modbus-Paket auf mehrere TCP-Pakete aufgeteilt wird. Das letzte Feld ist der *Unit-Identifier*. Dieser wird zum Adressieren eines zum Beispiel seriellen Modbus-Slaves verwendet. In so einem Fall dient der Modbus TCP Server als Gateway zum seriellen Bus.

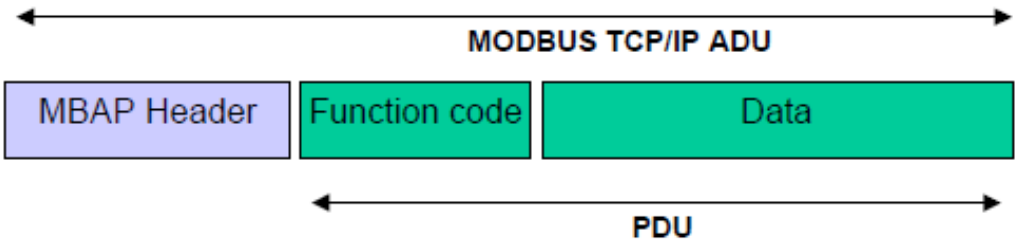

Abbildung 2.6: Modbus TCP Paket [Org13]

Name	Größe	Beschreibung
Transaction-Identifier	2 Byte	Identifikationsnummer des Request bzw. der Response
Protocol-Identifier	2 Byte	0 für Modbus-Protokoll
Länge	2 Byte	Identifikationsnummer für einen Slave der auf einem anderen Bus hängt.
Unit-Identifier	1 Byte	Wird zur Adressierung eines anderen Slaves verwendet, wenn es sich um einen Gateway handelt

Tabelle 2.23: MBAP-Header Aufbau

Bei Modbus TCP wird die ganze Verbindungsverwaltung vom TCP-Protokoll übernommen. TCP ist verbindungsorientiert und deshalb muss vor dem Datenaustausch eine Verbindung zwischen dem Client und dem Slave hergestellt werde. Der TCP-Port 502 ist speziell für Modbus reserviert. Auf diesem Port empfängt der Server die Requests vom Client. Für den Client ist kein spezieller Port definiert, dieser verwendet eine Port größer

als 1024 und verbindet sich mit dem Server auf Port 502. Wenn der Verbindungsaufbau erfolgreich war, kann der Client nun Daten an den Server senden. Bis hierher ist es der normale Ablauf einer TCP-Verbindung. Der Server muss nun nur noch darauf warten, dass auf dem geöffneten Socket Modbus-Pakete ankommen und diese dann verarbeiten. Auch den Verbindungsabbau übernimmt das TCP-Protokoll. Schließt einer der Kommunikationspartner den Socket, so bekommt der Andere dies mit und kann seinen Socket ebenfalls schließen.

2.3.1 TCP-Socket Parameter

Der TCP-Layer des Netzwerkstacks lässt sich parametrieren und für die Verwendung mit Modbus gibt es einige wichtige Parameter, die in der Folge erklärt werden. Die Bedeutung und Auswirkungen er einzelnen Parametern wurde aus [Ste98] entnommen.

SO-RCVBUF, SO-SNDBUF

Diese beiden Parameter bestimmen die maximale Empfangs- und Sendepuffergröße. Die maximale Puffergröße definiert die maximale Größe des angebotenen Fensters (Windowssize).
Die Empfangspuffergröße ist entscheidend für die Performance einer Modbus TCP Verbindung. Ein großer Empfangspuffer bedeutet, dass der Client mehrere Anfragen gleichzeitig an den Server senden kann, jedoch muss beachtet werden, dass der Server diese größere Menge an Anfragen, auch schnell genug verarbeiten kann. Wenn der Server ausgelastet ist, kann durch Reduzierung der Empfangspuffergröße und der damit verbundenen Verkleinerung der Fenstergröße, eine Entlastung des Server gewährleistet werden.

TCP-NODELAY

Mit diesem Parameter kann der Nagle-Algorithmus deaktiviert werden. Der *Nagle-Algorithmus* dient zum Reduzieren von kleinen Paketen in einem WAN. Als kleine Pakete gelten dabei, alle deren Größe unter der „Maximum Segment Size“ liegt. Der Algorithmus überprüft ob zurzeit die Quittierung von gesendeten Daten aussteht und wenn dies der Fall ist, so werden keine kleinen Pakete gesendet. Der Algorithmus wird bei der Verwendung in einem LAN gar nicht bemerkt, da hier die Übertragungszeiten innerhalb von wenigen Millisekunden liegen. In WANs hingegen kann die Quittierung durchaus auch bis zu einer Sekunde dauern. Dadurch wartet der Netzwerkstack nicht auf größere Datenmengen, sondern versendet auch kleine Pakete sofort. Dies ist in Bezug auf das Echtzeitverhalten sehr wichtig, da es sonst passieren kann, dass der Server die Antwort bereits an den Stack übergeben hat, dieser aber, wartet noch auf andere Daten bevor er sie versendet. Dadurch kann es zu Verzögerungen kommen, welche bei der Anwendung zu Fehlern führen kann.

SO-REUSEADDR

Durch diese Einstellung kann die sofortige Wiederverwendung eines Ports sichergestellt werden. Dies ist wichtig, weil wenn die Verbindung beendet wird, geht der Socket in den „Time-wait“-Status(2*Maximum Segment Lifetime) über und kein anderer Socket kann in dieser Zeit auf den Port zugreifen. Mit diesem Parameter kann dieser Wartezustand umgangen werden und jeder Socket kann zu jeder Zeit auf den Port zugreifen.
Es ist aber zu beachten, dass wenn mehrere Sockets gleichzeitig auf einen Port zu greifen, das Verhalten des Stacks nicht definiert ist, da er nicht weiß welchem Socket er die empfangenen Daten geben soll.

SO-KEEPALIVE

Durch die Aktivierung dieses Parameters werden bei TCP Leere-Pakete versendet um sicherzustellen, dass die Verbindung noch aufrecht ist. Standardmäßig ist dieser Parameter deaktiviert, was bedeutet, dass bei einer momentan nicht verwendeten Verbindung, gar keine Daten gesendet werden und deshalb auch nicht erkannt wird ob der jeweilige Verbindungspartner noch da ist.
Bei der Verwendung mit Modbus wird der Parameter aktiviert, um feststellen zu können ob der Server bzw. der Client noch aktiv ist. Es ist jedoch zu beachten, dass das Zeitintervall für die Keepalive-Pakete nicht zu klein ist und so das Netzwerk unnötig belastet wird.

SO-LINGER

Mit diesem Parameter lässt sich das Verhalten beim Schleißen eines Sockets beeinflussen. Standardmäßig schließt sich der Socket sofort nachdem Aufruf der Funktion, sollten sich aber noch Daten im Sendepuffer sein, so wird versucht diese noch erfolgreich zu übertragen. Dieses Verhalten kann nun mit dieser Option geändert werden. Wird sie aktiviert, so hat man die Möglichkeit anzugeben wie Lange versucht werden soll, die restlichen Daten zu senden. Setzt man die Zeit auf 0 so werden alle zu sendenden Daten sofort gelöscht.

Kapitel 3

Bestehendes System

In diesem Kapitel wird zuerst ein Überblick über das gesamte System gegeben. Dann werden die einzelnen, für diese Diplomarbeit wichtigen, Komponenten genauer untersucht und versucht beim Leser ein Grundverständnis für die Zusammenhänge der einzelnen Komponenten zu schaffen. Hier wird vor allem darauf Wert gelegt, dass der Weg von der Konfiguration über die grafische Oberfläche bis hin zur eigentlichen Abarbeitung von Modbus-Kommandos möglichst detailliert und korrekt erklärt wird, damit die in den folgenden Kapitel zu findenden Argumentation und Schlussfolgerungen möglichst klar und nachvollziehbar sind.

3.1 Allgemein

Im Wesentlichen besteht das System aus zwei große Software-Komponenten, dem Automation Studio, kurz AS und dem Automation Runtime, kurz AR. Diese beiden Komponenten bestehen aus sehr vielen Subkomponenten. In dieser Arbeit werden nur Ausgewählte und für die Implementation des Modbus TCP Slaves wichtige, Komponenten genannt und erklärt. Die Erläuterung aller vorhandenen Komponenten würde den Rahmen dieser Diplomarbeit bei weitem sprengen.
Das ganze System ist stark an die Norm EN 61131, welche ident mit der IEC 61131 ist, angelehnt und folgt in großen Teilen deren Ausführungen.

3.2 IEC 61131

Die IEC 61131 [IEC03] gilt für Speicherporgrammierbare Steuerungen (SPS) und besteht aus 8 Teilen. Der erste Teil dient als Einleitung und führt grundlegende Definitionen ein, erklärt grundlegende Eigenschaften von SPSen, die bei der Auswahl und Anwendung von Relevanz sind. Der dritte Teil beschäftigt sich mit der Software, vor allem mit den Programmiersprachen und deren Syntax. Aus diesen beiden Teilen werden in der Folge einige wichtigen Definitionen entnommen. Die anderen Teile der Norm beschäftigen sich unter anderem mit der Hardware und der Anwendung der definierten Programmiersprachen.

3.2.1 Speicherporgrammierbare Steuerung (SPS)

Eine speicherprogrammierbare Steuerung arbeit laut IEC[IEC03] digital und elektronisch, wobei sie für die Verwendung in einer industriellen Umgebung entworfen wurde. Sie hat einen programmierbaren Speicher in welchem anwenderorientierte Anweisungen gespeichert werden können. Durch diese Anweisungen können spezifische Funktionen wie Logik, Arithmetik, Ablauf und Zeit ausgeführt und durch digitale und analoge Ein- und Ausgänge, verschiedenste Arten von Prozessen und Maschinen gesteuert werden.

3.2.2 Funktionsbaustein

Ein Funktionsbaustein ist laut IEC 61131 [IEC13]eine Programm-Organisationseinheit(POU). Die Norm besagt, dass eine POU ein genau definiertes Stück des Programms enthält. Die POU hat ein definiertes Interface mit Ein- und Ausgängen und kann auch öfters aufgerufen werden.
Bei einem Funktionsbaustein muss zwischen dem Funktionsbaustein-Typ und der Funktionsbaustein-Instanz unterschieden werden.

- Funktionsbaustein-Typ besteht aus
  - der Definition einer Datenstruktur, mit Eingängen, Ausgängen und internen Variablen
  - einer Reihe von Befehlen, welche auf den Elementen der Datenstruktur einer Instanz des Funktionsbaustein-Typs ausgeführt werden können
- Funktionsbaustein-Instanz
  - eine mit Namen versehende Kopie des Funktionsbaustein-Typs
  - es kann mehrere von ihnen geben, wobei jede ihre eigene Datenstruktur haben muss

Beim Arbeiten mit Funktionsbausteinen ist zu bedenken, dass die Ausgänge und internen Variablen zwischen Operationen, die auf dem Funktionsbaustein ausgeführt werden, erhalten bleiben, das bedeutet, dass der Aufruf eines Funktionsbausteins mit den selben Eingangsparametern zu unterschiedlichen Ergebnissen führen kann.
Bei der Deklaration eines Funktionsbaustein-Typs kann für die internen Datenstruktur ein bereits definierter Funktionsbaustein-Typ verwendet werden. Dies schließt aber aus, dass die Datenstruktur den Funktionsbaustein-Typ selbst beinhaltet.

3.2.3 Datentypen

Die IEC 61131-3 [IEC13]spezifiziert einige elementare Datentypen, von denen ein Teil auch vom Automation Studio unterstützt werden. Die unterstützten Datentypen sind in Tabelle 3.1, mit Größe und Bezeichnung dargestellt. Es ist vor allem auf die Größe der Datentypen zu achten, da diese von den Heute gängigen Größen abweichen.

Schlüsselwort	Größe	Beschreibung
BOOL	1 Bit	Boolean
SINT	1 Byte	Short Integer
INT	2 Byte	Integer
DINT	4 Byte	Double Integer
USINT	1 Byte	Unsigned Short Integer
UINT	2 Byte	Unsigned Integer
UDINT	4 Byte	Unsigned Double Integer
REAL	4 Byte	reelle Zahl
BYTE	1 Byte	Bitfolge mit Länge 8
WORD	2 Byte	Bitfolge mit Länge 16
LREAL	8 Byte	lange reelle Zahl
DATE_AND_TIME	4 Byte	Datum und Uhrzeit
STRING	1 Byte	ASCII String mit variabler Länge
WSTRING	2 Byte	String mit 2 Byte pro Buchstaben und mit variabler Länge

Tabelle 3.1: Unterstützte IEC-Datentypen

3.2.4 B&R Eigenheiten

Im Automation Studio und im Automation Runtime gibt es einige Punkte, welche nicht der Norm entsprechen bzw. leicht von ihr abweichen. Im Folgenden werden jene erklärt, welche für diese Arbeit von Bedeutung sind.

Prozess Variable (PV)

Im Automation Studio werden Variablen, welche in den Variableneditoren angelegt werden als *Prozess Variable* oder kurz *PV* bezeichnet. Der Grund dafür ist, dass man sie so besser von Variablen, welche im Sourcecode deklariert werden unterscheiden kann. In der Regel verwendet man immer *PVs*, da zum Beispiel nur diese mit I/O-Datenpunkten verbunden werden können oder über das Automation Studio zur Laufzeit verändert werden können.

Datentypen

Bei den Datentypen gibt es in Bezug auf diese Arbeit, vor allem den B&R eigenen Datentyp *OCTET* zu erwähnen. Dieser wurde deshalb eingeführt, da die IEC nur Zuweisungen von Variablen mit gleichen Datentyp zulässt, mit der Ausnahme, dass wenn der Compiler eine Konvertierungsfunktion zwischen den beiden Datentypen kennt, darf er diese beim Kompilieren einfügen.
Aufgrund dieser Vorgabe wurde der Datentyp *OCTET* eingeführt um beim I/O-Mapping beliebige Datentypen mit einem Kanal zu verbinden.

3.3 Automation Studio

Das Automation Studio ist eine Desktopapplikation mit welcher dem Anwender die Möglichkeit geboten wird, das ganze System von der Hardware bis hin zu den einzelnen, von der Steuerung abzuarbeitenden, Programmen zu konfigurieren. Es soll hier kurz erwähnt werden, dass es nicht unbedingt nötig das Automation Studio zu verwenden um eine B&R-Steuerung zu betreiben. Es gibt mehr oder weniger gut definierte Schnittstellen über welche man die Automation Runtime konfigurieren kann, jedoch wird im Rahmen dieser Arbeit, die Konfiguration des Modbus TCP Slaves über das Automation Studio betrachtet und deshalb nicht weiter auf andere Konfigurationsmöglichkeiten eingegangen.

Das Automation Studio bietet neben den bereits erwähnten Konfigurationsmöglichkeiten, auch einen Debugger und diverse andere Tools, die SPS zu überwachen und zu Steuern.
Es gibt den sogenannten *Watch* mit welchem, Variablen zur Laufzeit betrachtet und geändert werden können.
Im *Monitor-Mode* werden unter anderem Unterschiede zwischen der Konfiguration und der online vorhandenen Hardware angezeigt. Es können die von I/O-Knoten gelieferten Werte betrachtet werden bzw. können diese Werte zu Testzwecken auch geändert bzw. simuliert werden. Dies ist beim Testen einer neuen Programmkomponente sehr hilfreich.
Zwei Weitere sehr nützliche Tools sind der Logger und der Profiler. Mit dem Logger kann man das Logbuch auf der SPS betrachten. Die SPS schreibt alle Informationen, die für den Anwender von Interesse sein könnten, in das Logbuch. Jeder Eintrag kann in eine der folgenden Kategorien zugeteilt werden:

- **Information**
  Unter dieser Kategorie findet man Punkte die nur als Information für Anwender dienen. Zum Beispiel wird ein Eintrag generiert werden wenn der Anwender die Steuerung neustartet.

- **Warnung**
  Einträge in dieser Kategorie deuten auf einen Fehler hin, der aber nicht so gravierend ist, dass die Steuerung in den sicheren Modus (*Service-Mode*) wechselt.

- **Fehler**
  Solche Einträge werden generiert, wenn ein Fehler auftritt, der es der Steuerung unmöglich macht zu starten bzw. weiter zu arbeiten. Dies kann zum Beispiel auftreten, wenn ein Anwenderprogramm eine Division durch Null ausführt.

Jeder Eintrag im Logbuch besitzt einen Zeitstempel, eine Fehlernummer und eine kurze Fehlerbeschreibung. Mit der Fehlernummer findet man in der Hilfe eine genauere Fehlerbeschreibung und Behebungsvorschläge.
Der Profiler ist ein Tool mit sehr großem Funktionsumfang, hier soll nur erwähnt werden, dass man mit ihm fast alle Aktivitäten der SPS aufzeichnen und anschauen kann. Das geht soweit, dass man sich die Ticks und Interrupts der CPU zeitlich darstellen lassen

kann und zum Beispiel die Taskwechsel darüberlegen kann.

Bis hierher wurde versucht einen groben Überblick über die Möglichkeiten des Automation Studios zugeben. Nun soll speziell auf die Konfiguration einer Steuerung eingegangen werden.

3.3.1 Konfiguration

Die Konfiguration im Automation Studio beginnt mit dem Erstellen eines neuen Projektes. Hier muss bereits eine gewünschte Steuerung ausgewählt werden. Nachdem das Projekt erstellt wurde, kann man, in einer der drei zur Verfügung stehenden Ansichten, mit der Konfiguration des Systems beginnen.
Um ein Verständnis für die Konfiguration mit dem Automation Studio zu schaffen, werden als Erstes die drei Sichten, die das Automation Studio auf das System bietet, erklärt. Jede der einzelnen Sichten bietet die Möglichkeit einen Teil des Systems zu konfigurieren. Wenn alle gewünschten Einstellungen in den drei Ansichten getätigt wurden, kann die Konfiguration auf die SPS übertragen bzw. an das Automation Runtime übergeben werden. Diese Übertragung basiert auf einem speziellen Dateiformat, welches sich BR-Modul (siehe Kapitel 3.4) nennt. Die ganzen Einstellungen des Anwenders werden in solchen BR-Modulen verpackt und das Automation Runtime wertet und führt diese dann aus.

Logical View

Die *Logical View*(siehe Abbildung 3.1) oder logische Sicht, betrachtet nicht die SPS an sich, sondern die Software bzw. die Anwendungen welche später auf der Steuerung laufen sollten. Die Betrachtung erfolgt hier, wie auch bei den beiden anderen Ansichten, über einen Baum. Die Wurzel bildet der Projektname. Darunter befinden sich in der Regel die globalen Deklarationsdateien. Es können sowohl globale Variablen, als auch globale Datenstrukturen angelegt werden. Diese sind dann in jedem Task verfügbar.
Als Nächstes folgt dann der Bibliotheksordner (*Libraries*), hier werden normalerweise alle verwendeten Bibliotheken abgelegt. Das Automation Studio beinhaltet eine große Menge an Standardbibliotheken. Viele dieser Bibliotheken bieten zur Laufzeit, einen direkten Zugriff auf die Hardwaretreiber. Eine dieser Bibliotheken wurde im Rahmen dieser Arbeit erstellt und wird im Kapitel 4.2.3 beschrieben.
Die bisher beschriebenen Elemente, befinden sich nach dem Erstellen eines neuen Projektes in der *Logical View*. Man kann alle Elemente bis auf die Wurzel auch entfernen oder man kann eine Reihe von neuen Elementen hinzufügen. Man kann selbst neue Bibliotheken erstellen oder andere Standardbibliotheken hinzufügen. In der *Logical View* werden auch die Anwenderprogramme hinzugefügt. Beim Hinzufügen eines neuen Programms erscheint ein Dialog-Fenster in dem man unter anderem den Namen und die Programmiersprache auswählen kann.

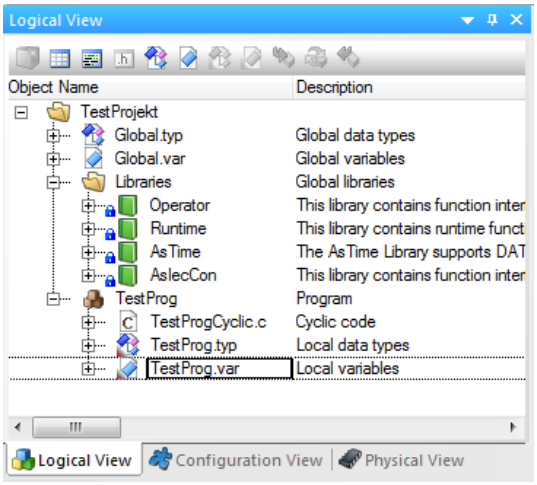

Abbildung 3.1: Die logische Ansicht im AS

Configuration View

In der *Configuration View*(siehe Abbildung 3.2) kann man seine angelegten Konfiguration verwalten. Wie bereits erwähnt muss bereits beim Erstellen eines Projektes, eine Steuerung konfiguriert werden, d.h. jedes Projekt hat mindestens eine Konfiguration. Es können beliebig viele Konfigurationen mit der gleichen oder mit unterschiedlichen Steuerungen angelegt und konfiguriert werden. In der *Configuration View* werden alle angelegten Konfigurationen verwaltet, wobei immer nur eine aktiv sein kann.

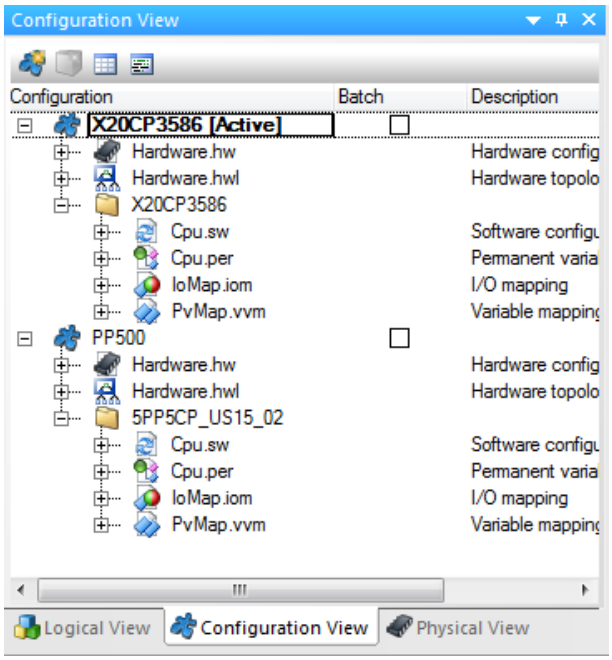

Abbildung 3.2: Die Konfigurationsansicht im AS

Physical View

Die *Physical View*(siehe Abbildung 3.3) stellt die Hardwaresicht der aktuellen Konfiguration zur Verfügung. Hier wird die SPS mit all ihren Anschlüssen und Schnittstellen dargestellt. Jede Komponente hier bietet gewisse Konfigurationsmöglichkeiten. Öffnet man zum Beispiel den Konfigurationseditor einer Ethernetschnittstelle, siehe Abbildung 3.4, kann man unter anderem die IP-Adresse einstellen. Jede Komponente hat noch einen zweiten wichtigen Editor. Dieser nennt sich *IO-Mapping* und dient dazu, dem Anwender die Möglichkeit zu geben, Variablen aus seinen Programmen mit Werten von I/O-Knoten und Schnittstellen zu verbinden. In Abbildung 3.5 sieht man ein Beispiel hierfür. Die Eingänge eines digitalen Eingangsmodul werden im *IO-Mapping* als Datenpunkte angeboten und die Variable „var1“ aus dem Programm „testProg“ wird damit verbunden. Diese Verbindung bedeutet, dass nun vor jedem Start des zyklischen Programms „testProg“, der aktuelle Zustand des digitalen Eingangs auf die Variable „var1“ geschrieben wird.
In Abbildung 3.3 sieht man zudem, dass an die einzelnen Schnittstellen der SPS passende Module angeschlossen werden können. In diesem Beispiel wurde an die X2X-Schnittstelle ein digitales Eingangsmodul angeschlossen. Bei allen Elementen im Baum sieht man zudem, in der Spalte „PLC Address“, die Adresse, welche sie später im Automation Runtime haben. Beim eingefügten X2X-Modul lautet diese *IF6.ST1*, das heißt das Modul hängt an der Schnittstelle *IF6* auf Steckplatz 1.

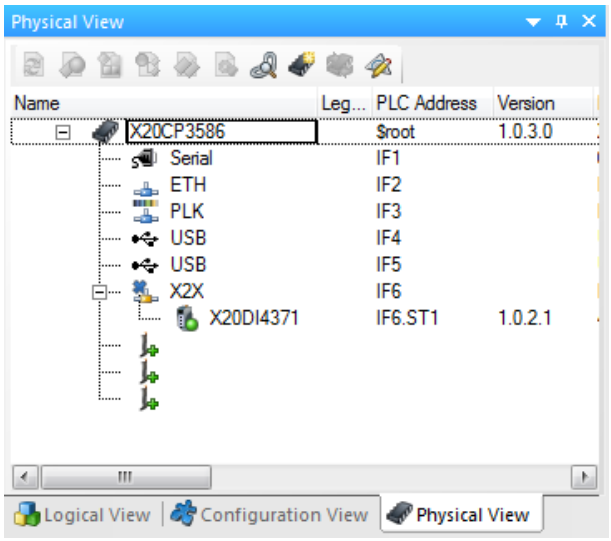

Abbildung 3.3: Die physikalische Ansicht im AS

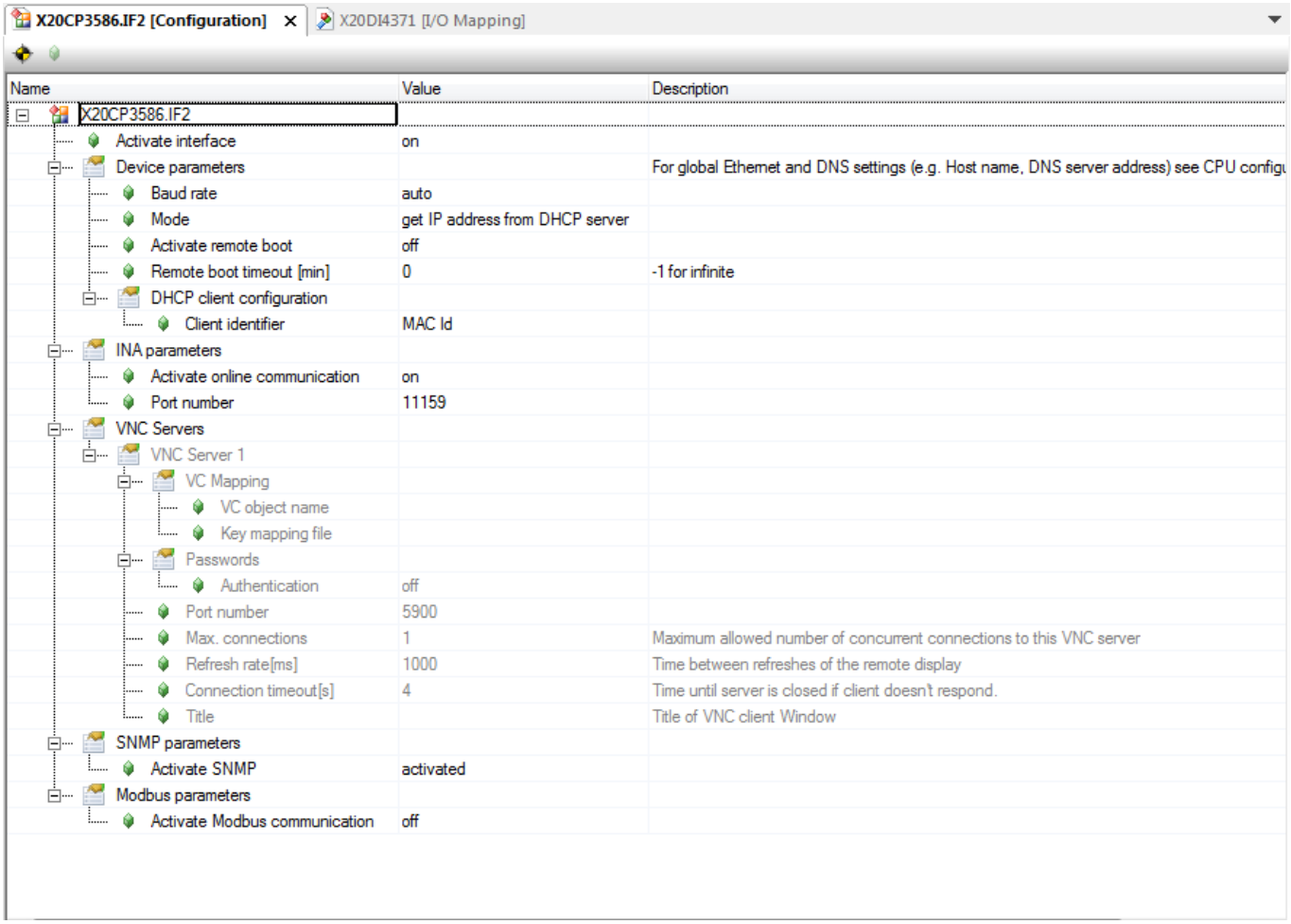

Abbildung 3.4: Konfigurationseditor einer Ethernetschnittstelle

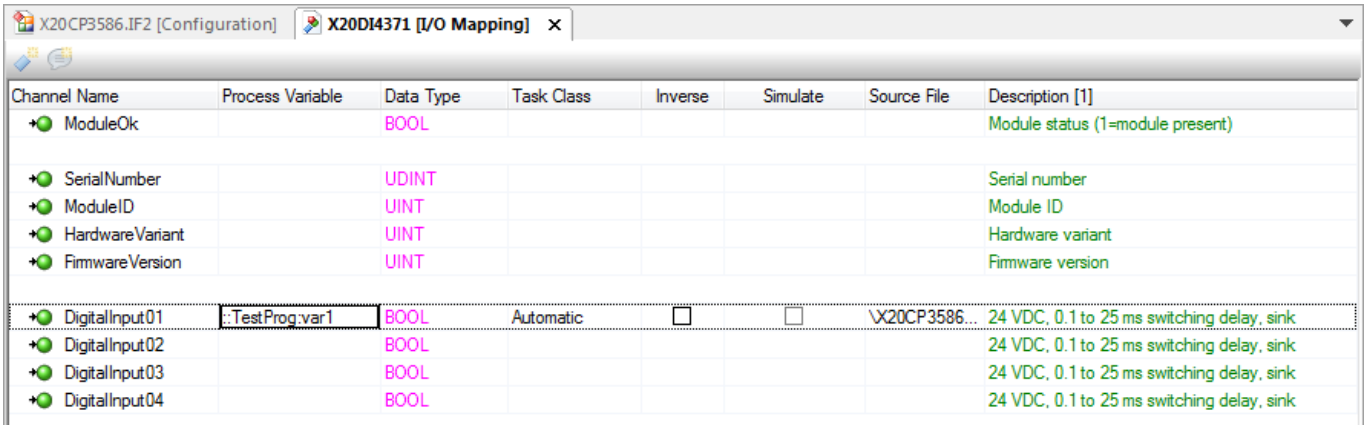

Abbildung 3.5: I/O-Mapping eines digitalen Eingangsmoduls

3.4 BR-Modul

Ein BR-Modul [Ber98] besteht aus einem Header, einem Bereich mit Offsets, bis zu 255 Sektionen und einem Bereich für Prüfsummen(siehe Abbildung 3.6).
Der Header beginnt immer mir der fixen Kennung 0x2b97. Im Header stehen des Weiteren der Name und des Typ des Moduls. Anhand des Typs ist definiert welche und wie viele Sektionen das Modul hat.
Im Offsetbereich steht für jede Sektion ein Offset um das Modul schneller durchsuchen zu können. Im letzten Bereich des BR-Moduls finden sich einige Prüfsummen um sicherzustellen, dass das Modul nicht beschädigt worden ist.
Der Name des Moduls, welcher im Header steht dient als Identifikator. Im Automation Runtime spielen einige Module für die Konfiguration eine entscheidende Rolle und werden deshalb im Folgenden genauer erklärt.

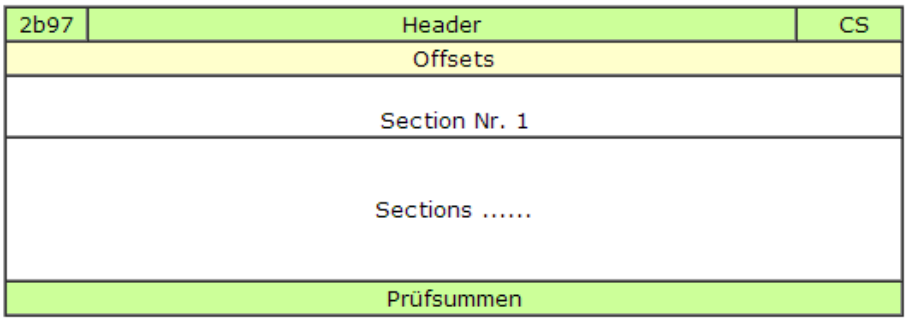

Abbildung 3.6: Aufbau eines BR-Moduls [Ber98]

3.4.1 arConfig: Automation Runtime Konfiguration (0x84)

Diese Modul wird kurz *arConfig* genannt und beinhaltet, wie der Name bereits vermuten lässt, die Konfiguration für das Automation Runtime. Hier sind alle konfigurierten Module mit ihren Parametern im XML-Format eingetragen. Der erste Eintrag ist immer das Modul mit der „ID“ „$root“, welches in Abbildung 3.7 dargestellt ist. Das „Hardware“-Attribut gibt in diesem Fall an um, welche Steuerung es sich handelt. Beim Start des Automation Runtimes wird diese Konfiguration in den Hardware-Tree(siehe Kapitel3.5.2) geladen und kann über ein definiertes Interface abgefragt werden.

```
<Module ID="$root" Hardware="X20CP3586">
```

Abbildung 3.7: Erster Eintrag in der *arConfig*

3.4.2 ioMap: I/O Mapping (0x32)

Das I/O-Mapping, kurz „ioMap“, beinhaltet Informationen, wie Variablen mit I/O-Punkten bzw. Variablen mit Variablen verbunden sind. Eine solche Verbindung oder auch Map-

ping genannt, besteht aus einem Produzent(Producer) und einem oder mehreren Verbrauchern(Consumer). Der Producer liefert sozusagen den Wert für die Consumer. Solche Verbindungen werden auch Link-Nodes genannt und in Abbildung 3.8 sieht man einen Eintrag des BR-Moduls. Die Einträge erfolgen im XML-Format, wobei der XML-Tag „LN“ heißt. Der „Prod“-Tag beinhaltet die Information wo die Automation Runtime den Producer findet. Das „Device“-Attribut gibt die Adresse des Gerätes an, welches den Wert liefert und das „DPName“-Attribut beinhaltet den Namen des entsprechenden Datenpunktes auf dem Gerät. Ein Datenpunkt kann im Prinzip als Variable gesehen werden.
Ein Consumer-Eintrag schaut gleich aus, bis auf das, dass der XML-Tag „Cons“ verwendet wird.

```
<LN ID="%IX.IF6.ST1.DigitalInput01" Type="BOOL">
<Prod Device="IF6.ST1" DPName="DigitalInput01" Kind="io"/>
<Cons Device="TC#4-CPYDEV" DPName="::TestProg:var1" Kind="pv"/>
</LN>
```

Abbildung 3.8: Link-Node Eintrag im ioMap BR-Modul

3.5 Automation Runtime

Das Automation Runtime ist eine Applikation, die dazu gedacht ist auf einem Echtzeit-Betriebssystem zu laufen. Sie bedient sich einer definierten Schnittstelle, dem sogenannten *Real-Time-Kernel*, welcher die Betriebssystem-Funktionen abstrahieren soll, damit das Automation Runtime unabhängig vom verfügbaren Echtzeit-Betriebssystem ist. Aus heutiger Sicht dient hauptsächlich VxWorks von Wind River Systems als Betriebssystem. In der Automation Runtime sind folgende Hauptkomponenten implementiert:

- **Kommunikation mit dem Automation Studio**
  Hier gibt es mehrere Protokolle, welche auf verschiedenen Übertragungsmedien aufbauen.

- **Taskklassen** [AG02]
  Der Begriff Taskklassen umfasst hier den gesamten Teil der Abarbeitung von Anwender-Programmen. Diese Programme werden bei B&R als Tasks bezeichnet und diese können in unterschiedliche Taskklassen eingeordnet werden. Jede Taskklasse hat eine Periode mit welcher ihre Tasks zyklisch aufgerufen werden. Bei der Abarbeitung einer Taskklasse, gibt es mehrere Aufgaben. Die Variablen der Task, welche mit Eingangs-Datenpunkten verbunden sind müssen vor dem Starten der Taskklasse, mit den aktuellen Werten befüllt werden und nach Abschluss der Taskklasse müssen die Werte der Variablen, welche mit Ausgangs-Datenpunkten verbunden sind, auf die Ausgangsmodule geschrieben werden.

- **Automation Runtime Scheduler** [Gra08]
  Der Automation Runtime Scheduler ist dafür zuständig, dass die einzelnen Taskklassen und ihre Subkomponenten dem vorgegebenen Ablauf folgen.

- **Geräte Verwaltung**
  Unter dem Begriff *Geräte Verwaltung* ist hier die gesamte Verwaltung der vorhandenen Geräte bzw. Treiber zu verstehen. Hier gehören Subkomponenten wie der *Device Managers* oder der *Hardware-Baum* dazu.

Im Folgenden werden einige Komponenten bzw. Subkomponenten erklärt, da sie für die Implementierung von Beudeutung sind.

3.5.1 Hardware Description-Manager

Der *(*Hardware Description Manager) bezieht seine Informationen aus XML-Dateien, in welchen alle für die Konfiguration notwendigen Hardware Beschreibungen enthalten sind. Diese Dateien werden auch als HWD-Dateien bezeichnet. Auf dem Speichermedium von welchem das Automation Runtime gebootet wird, befinden sich in der Regel zwei bestimmte Ordner, in welchen die Hardware Beschreibungen zu finden sind. Einer der beiden Ordner enthält einen Basissatz an HWD-Dateien. In diesen Dateien finden sich alle Hardware Informationen die für einen erfolgreichen Hochlauf der Steuerung notwendig sind. Im zweiten Ordner befinden sich HWD-Dateien, in denen sich, die speziell für die aktuelle Konfiguration benötigten, Hardware Informationen befinden. Diese Dateien werden vom Automation Studio je nach Konfiguration generiert.
Eine HWD-Datei besteht aus einzelnen Einträgen. Für jedes Gerät bzw. Modul wird ein eigener Eintrag erstellt. Ein Eintrag beinhaltet den Bezeichner des Geräts und all seine Parameter, zu denen auch seinen Kanäle gehören.

3.5.2 Hardware-Baum

Dem *Hardware-Baum* liegt, wie der Name bereits sagt, ein Baumstruktur zu Grunde. Die Wurzel bildet immer die CPU, an der dann mehrere Konten hängen, an denen wiederum Konten hängen können. Im Hochlauf wird der Baum mit Daten befüllt. Als Erstes kommt die CPU, diese wird je nach Typ auf unterschiedliche Arten ermittelt. Sobald klar ist um welche CPU es sich handelt, wird in den HWD-Informationen nachgesehen welche Schnittstellen sie besitzt. Diese Schnittstellen werden, dann als Konten in den *Hardware-Baum* eingetragen. Jeder Knoten kann zudem Parameter besitzen. Diese Parameter stammen entweder aus den HWD-Informationen oder kommen aus der *arConfig*.

3.5.3 Device Manager

Der *Device Manager* [Gra11] verwaltet die Geräte bzw. die Treiber der einzelnen Geräte, welche als Schnittstelle zum eigentlichen Gerät dienen. Er hat eine Liste aller bisher erzeugten Geräte und bietet folgende Möglichkeiten:

- Gerät erzeugen
- Gerät löschen
- Gerät initalisieren
- Gerät herunterfahren/deinitalisieren
- auf das Gerät zugreifen

Beim Starten des Automation Runtime werden die konfigurierten Geräte bzw. deren Treiber erzeugt und gestartet.Das Erzeugen der Treiber basiert auf dem Installieren von Modulen. Deshalb wird für jeden Treiber zur Laufzeit ein *BR-Modul* erzeugt, wobei eine Installations- und eine Deinstallationsfunktion angeben wird. In der Installationsfunktion wird der Treiber beim *Device Manager* angemeldet und die Schnittstelle zu ihm wird mit Funkitonszeigern befüllt. Beim Deinstallieren des Treibers, wird dieser aus der Liste des *Device Managers* gelöscht. In einer späteren Phase des Hochlaufs werden sie initialisert, was gleich bedeutend mit dem Starten des Treibers ist. Einige Geräte können zur Laufzeit an- und abgeschlossen werden, für diese gibt es eine eigene Plug&Play-Behandlung. Bei solchen Geräten kann es natürlich auch passieren, dass das der Treiber heruntergefahren und die interne Verwaltungsstruktur freigegeben werden muss. Geräte die sich nicht zur Laufzeit an- bzw. abschließen lassen, werden nur beim Herunterfahren des gesamten Systems aus dem *Device Manager* entfernt.

Der Zugriff auf die verschiedenen Treiber muss natürlich über eine definierte Schnittstelle erfolgen. Jeder Treiber muss diese Schnittstelle implementieren. Da dieser Teil des Automation Runtime in der Programmiersprache C implementiert ist, werden dazu in einer definierten Struktur eine bestimmte Anzahl an Funktionpointern gesetzt. Jeder Eintrag in der Geräteliste des *Device Managers* hat nun diese Funktionen, mit diesen kann der *Device Managers* nun einheitlich auf die Treiber und damit auf die Geräte zugreifen. Die Schnittstelle besteht aus 6 Funktionen von denen nicht alle implementiert werden müssen:

- **DeviceInit**
  Anlegen der internen Verwaltung
- **DeviceDeInit**
  Freigeben der internen Verwaltung
- **DeviceShutdown**
  Optionale Maßnahmen beim Abschalten des Treibers, zum Beispiel Beenden von gestarteten Tasks
- **DeviceOpen**
  Öffnen/Erzeugen eines Datenpunkts

- **DeviceClose**
  Freigeben eines Datenpunkts
- **DeviceRead**
  Lesen eines Datenpunkts
- **DeviceWrite**
  Schreiben eines Datenpunkts
- **DeviceIoControl**
  Über einen sogenannten „Controlcode“ kann eine spezielle Funktion des Treibers ausgführt werden. Es gibt einige global definierte „Controls“, welche jeder Treiber unterstützen muss, wie zum Beispiel das Auslesen von Datenpunkt-Adressen für die *Link Nodes*. Es ist aber auch möglich neue, speziell für diesen Treiber verwendete, „Controls“ zu definieren.

3.5.4 LinkNode Manager

Der *LinkNode Manager* [Hav02] verwaltet alle Link-Nodes aus der iomap(siehe Kapitel 3.4.2). Während des Startvorgangs der Automation Runtime werden alle Link-Nodes beim *LinkNode Manager* angemeldet und dieser versucht alle Producer und Consumer zu erreichen und die Speicheradresse des jeweiligen Datenpunktes abzufragen. Sollten Datenpunkte nicht gefunden werden, so wird dies als Warnung im Logbuch eingetragen.
Bevor die Taskklassen gestartet werden, werden die Link-Nodes aktiviert, das heißt das umkopieren der Producer-Werte auf die Consumer wird freigegeben. Dies geschieht, bei jedem Start und jedem Abschluss einer Taskklasse.

Kapitel 4

Spezifikation

In diesem Kapitel wird als Erstes versucht die Anforderungen wiederzugeben. In der Folge wird erklärt wie diese am besten erfüllt werden können. Hier sind natürlich die bereits bestehenden Strukturen(siehe Kapitel 3) zu beachten. Sowohl die Konfiguration, als auch der eigentliche Modbus-Stack müssen im System verankert werden und sollten im Einklang mit der gegebenen Software-Architektur sein.

4.1 Anforderungen

Die eigentliche Anforderung der Kunden lautete, dass es möglich sein soll eine B&R-Steuerung als Modus TCP Slave zu konfigurieren. Diese sehr allgemeine Anforderung wurde in mehreren Besprechungen genauer spezifiziert und erläutert, wobei es so zu spezifischeren Anforderungen gekommen ist. Bei den Besprechungen waren neben dem Abteilungsleiter und dem Gruppenleiter, auch ein Software-Architekt und als Repräsentant für die Anwender, ein B&R-Mitarbeiter aus der Applikationsabteilung anwesend. Diese Abteilung entwickelt die Applikationen für Kunden, welche keine eigene Software-Abteilung haben.

Die Anforderungen von Kundenseite waren nun Folgende:

- Es muss möglich sein eine B&R-Steuerung als Modbus TCP Slave zu betreiben.
- Die „openSafety“-Kommunikation darf nicht gestört werden.
- Es sollte möglich sein mehrere nebeneinander liegende Modbus-Register auf ein Array zu mappen.
- Es sollten folgenden Diagnose-Kanäle verfügbar sein:

Kanalname	Datentyp	Beschreibung
ActNrOfClients	UINT	Anzahl an momentan verbundenen Mastern
PacketCnt	DINT	Anzahl an empfangenen Paketen
ErrorCnt	DINT	Anzahl an fehlerhaften Paketen
TimeSinceLastRequest	DINT	Zeit seit dem letzten Request in [ms] (Auflösung 100ms)

Tabelle 4.1: Diagnosekanäle

- Es wäre wünschenswert, wenn mehrere verschiedene Modbus-Adressräume auf einer Steuerung konfiguriert werden könnten. Zur Adressierung der unterschiedlichen Adressräume könnte der *Unit-Identifier*(siehe Kapitel 2.3) verwendet werden.

Bei der Diskussion über die Kundenanforderungen wurden auch gleich mögliche Arten der Konfiguration diskutiert und einige weitere Eckpunkte für die Implementation festgelegt:

- Die Konfiguration soll an jene des Modbus TCP Masters angelehnt werden, da viele Kunden diesen bereits verwenden
- Die Verbindung zwischen den Modbus-Daten und Prozess-Variablen soll über das IO-Mapping hergestellt werden.
- Die Konfiguration sowie die Implementierung sollen kompatibel mit der System- bzw. Softwarearchitektur sein.
- Der Modbus TCP Slave soll ab der Automation Studio Version 4.1 und dem Automation Runtime Version E4.06 verfügbar sein.

Mit diesen Anforderungen wurde ein Design erstellt, welches im nächsten Abschnitt erläutert wird.

4.2 Design

Das Design wurde anhand der Anforderungen erstellt. Hier wurde vor allem auf die bereits bestehende Architektur geachetet. Sowohl die Konfiguration, als auch die Implementierung sollte vorgegebene Schnittstellen verwenden und Konventionen einhalten. In Kapitel 3 wurden diese bereits erklärt und werden beim Design jetzt berücksichtigt.

4.2.1 Allgemein

Adress- bzw. Speicherlayout

Das Adresslayout eines Modbus Slave kann, wie bereits in Kapitel 2.2.1 erklärt, unterschiedlich sein. Deshalb wurde dies beim Design als Erstes betrachtet. Der erste Ansatz

war für jeden Modbus-Datentyp einen eigenen Speicher zu reservieren, das heißt für jeden wären 65.536 Elemente möglich. Bei der Diskussion dieses Ansatzes wurde erwähnt, dass B&R bereits einen Buskoppler, welcher als Modbus TCP Slave arbeitet, anbietet. Dieser Umstand führte dazu, dass das Adress- bzw. Speicherlayout des Buskopplers verwendet wurde, damit sich die Kunden nicht umstellen müssen. In der Folge wird das angewandte Layout erläutert, wobei mehrere Punkte aus dem Handbuch des Buskopplers ([Ber09]) entnommen wurden.
Das implementierte Adresslayout ist in Abbildung 4.1 dargestellt. Es gibt zwei separate Adressräume für die digitalen und analogen Datentypen. Im digitalen Adressraum finden sich die *Coils* und *Discrete Inputs* und im analogen die *Holding Register* und die *Input Register*.

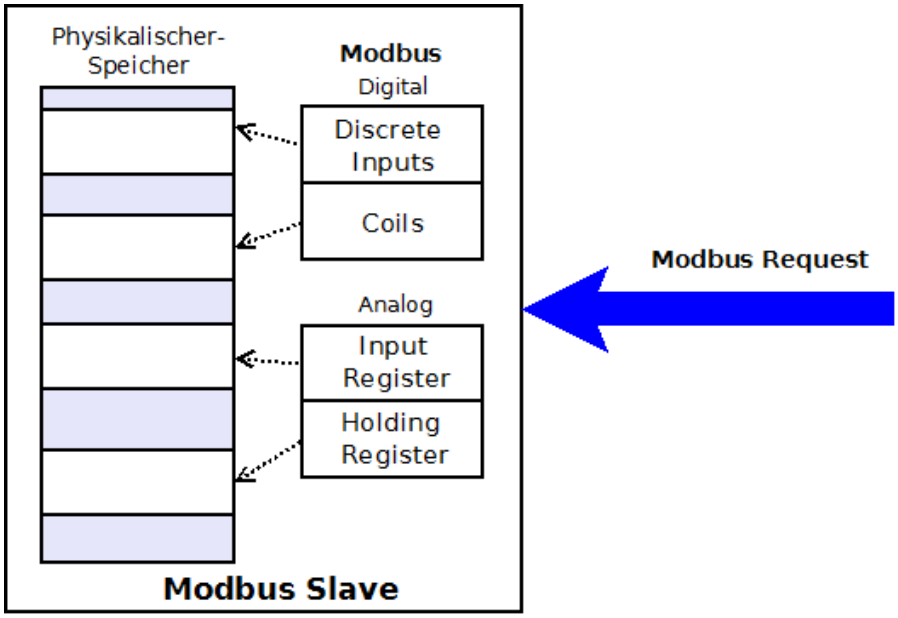

Abbildung 4.1: Modbus TCP Slave Adresslayout

In den jeweiligen Adressräumen findet eine Unterteilung der Datentypen anhand des Adressbereiches statt. In den Tabellen 4.2 und 4.3 sind die Bereiche der einzelnen Datentypen und die zulässigen Modbus-Funktionskodes dargestellt.

Adressbereich	Anzahl an Elementen	Verwendung	Zulässige FC
0x0000 - 0x1FFF	16348	Discrete Inputs	2
0x2000 - 0x3FFF	16348	Coils	1,5,15
0x4000 - 0xFFFF	32768	Reserviert	–

Tabelle 4.2: Digitale Adressbereiche

Die in Tabelle 4.3 angeführten Systemparameter, dienen als Diagnosewerkzeug für den Modbus TCP Master. In Tabelle 4.4 sind die belegten Adressen und ihre Bedeutung

Adressbereich	Anzahl an Elementen	Verwendung	Zulässige FC
0x0000 - 0x0FFF	4096	Reserviert	–
0x1000 - 0x1FFF	4096	Systemparameter	3, 4, 6, 16, 23
0x2000 - 0x5FFF	16348	Input Register	3, 4, 23
0x6000 - 0x9FFF	16348	Holding Register	3, 4, 6, 16, 23
0xA000 - 0xFFFF	32768	Reserviert	–

Tabelle 4.3: Analoge Adressbereiche

angeführt.

Adresse	Beschreibung	Zugriffsrecht
0x1000 - 0x1002	Mac-Adresse	lesend
0x1003 - 0x1004	IP-Adresse	lesend
0x1005 - 0x1006	Subnet-Maske	lesend
0x1007 - 0x1008	Standard-Gateway	lesend
0x1040	Watchdog Threshold[ms]	lesend, schreibend
0x1042	Watchdog Status: 0=kein WD; 1=WD abgelaufen	lesend
0x1043	Watchdog Mode: 0=WD nicht aktiv; 1=WD aktiv	lesend,schreibend
0x1080 - 0x1081	Seriennummer	lesend
0x1083	Produktcode; Hardware-ID	lesend
0x10C0	Anzahl der Client-Verbindungen	lesend
0x10C1 - 0x10C2	Globaler Telegramm Zähler (alle Verbindungen)	lesend
0x10C3 - 0x10C4	Lokaler Telegramm Zähler (nur diese Verbindung)	lesend
0x10C5 - 0x10C6	Globaler Fehler Zähler	lesend
0x10C7 - 0x10C8	Lokaler Fehler Zähler	lesend

Tabelle 4.4: Systemparameter

openSafety Behandlung

Die openSafety Kommunikation kann bei entsprechender Konfiguration auch über Modbus TCP erfolgen. Wenn dies der Fall ist, so wird der Standard TCP-Port für Modbus, Port 502, reserviert und kann nicht für die Konfiguration eines Modbus TCP Slave verwendet werden.

4.2.2 Automation Studio

Im Automation Studio musste die Konfiguration des Modbus TCP Slaves implementiert werden. Diese beginnt bei der grafischen Darstellung und endet bei der Generierung der BR-Module, die für die Konfiguration des Automation Runtimes verwendet wird.

Die Anforderung, dass die Konfiguration ähnlich der des Modbus TCP Masters sein soll, wurde untersucht, jedoch gibt es sehr wenige Gemeinsamkeiten. Der Slave reagiert nur auf Anfragen, wobei der Master Anfragen generieren muss, deshalb wurde untersucht ob es bereits eine ähnliche Konfiguration im Automation Studio gibt. Hier wurde die Funktion der „intelligent Control Node“(iCN) des Ethernet-Powerlink-Protokolls gefunden. Hier gibt es zwei mögliche Konfigurationsmöglichkeiten, zum einem die Erzeugung von Kanälen für das I/O-Mapping und zum Anderen die Definition eines Speichers, welcher mit FUBs beschrieben bzw. gelesen werden kann.

In der Folge wird die Konfiguration des Modbus TCP Slaves in Anlehnung an die der iCN erläutert.

Konfiguration

Die Konfiguration des Modbus TCP Slaves wurde in jene der Ethernet-Schnittstelle integriert. Unter dem Punkt „Modbus Parameters“ gab es bereits die Konfiguration des Masters. Zu dieser Gruppe wurde nun die Konfiguration des Modbus TCP Slaves hinzugefügt. In Abbildung 4.2 ist die Konfiguration für Modbus TCP dargestellt. Der neue Parameter „Use as Modbus slave“ gibt den Anwender die Möglichkeit den Slave zu aktivieren bzw. dekativieren. Ist der Parameter aktiviert, so hat der Anwender die Möglichkeit zwischen bis zu 5 Modbus TCP Slaves zu konfigurieren.

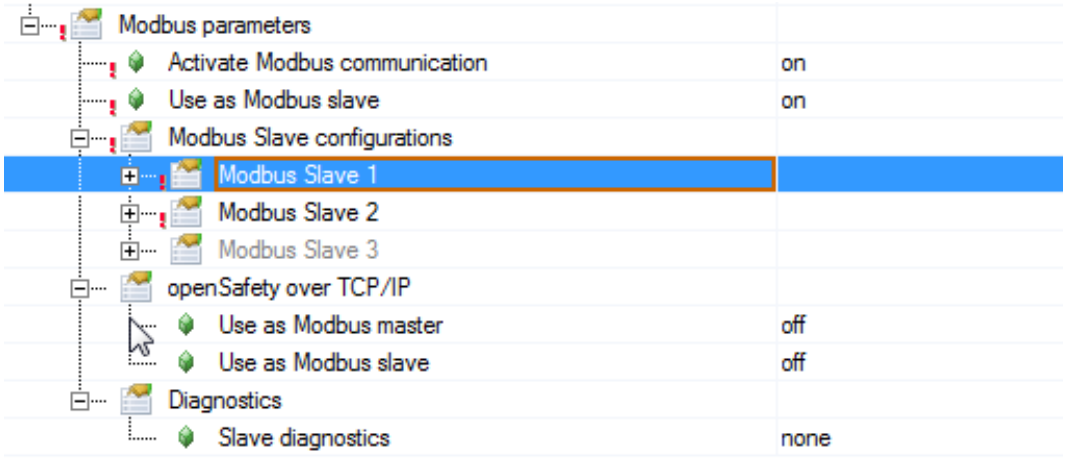

Abbildung 4.2: Modbus TCP Konfiguration

Bei jedem Slave kann als Erstes der Port, auf welchem er läuft, festgelegt werden. Hier ist zu beachten, dass wenn der „openSafety Modbus Slave“ aktiviert ist, darf bei der

Konfiguration der Port 502 nicht mehr verwendet werden. Der nächste Parameter gibt den Typ der Konfiguration an. Bei beiden Typen müssen jeweils für *Coils*, *Discrete Inputs*, *Input Registers* und *Holding Registers* Einstellungen vorgenommen werden. In der Folge werden die beiden Konfigurationsarten näher erläutert:

- **dynamic Channels**
  In Abbildung 4.3 sind die Parameter, welche bei dieser Konfigurationsart vorhanden sind, dargestellt. Für jeden Modbus-Datentyp können Kanäle erstellt werden, welche dann im I/O-Mapping-Editor mit Prozess-Variablen verbunden werden können. Diese Kanäle werden im Automation Studio als dynamische Kanäle bezeichnet, da sie je nach Konfiguration verschieden sind. Kanäle fügt man einfach durch Ausfüllen des grauen Eintrags hinzu. Wird ein grauer Kanal ausgefüllt, so wird dieser schwarz und damit aktiv und es kommt ein neuer grauer Kanal hinzu.
  Jeder Kanal besteht aus einem „Namen“, welcher ihn auf der Schnittstelle eindeutig identifiziert, einer „Adresse“ und einem „Datentyp“. Der „Datentyp“ dient nur als Information für den Anwender und kann nicht geändert werden. Bei den digitalen Datentypen, *Coil* und *Discrete Input*, ist der „Datentyp“ immer *BOOL* und bei den anlogen, *Input Register* und *Holding Register*, immer *OCTET*, wobei zu beachten ist das ein *OCTET* einem Byte entspricht und deshalb ein Modbus-Registern einem *OCTET*-Array mit zwei Elementen entspricht. Bei den Register-Datentypen kommt zu den drei, bereits genannten Parametern, noch die „Anzahl an Registern“ dazu. Mit diesem Parameter ist es möglich, einen zusammenhängenden Block an Modbus-Registern zu konfigurieren und den gesamten Block mit einer Prozess-Variable zu verbinden. In Abbildung 4.4 sieht man am Beispiel dafür. Die Prozess-Variable „intArray16“ ist ein INT-Array und hat 16 Elemente. Die Speichergröße dieses Array ist demzufolge 32 Byte, das heißt es kann mit dem Kanal „HoldingRegChannel1“ verbunden werden, da beide die gleiche Anzahl an Bytes, im Speicher belegen.

- **fixed buffer size**
  Hier ist die Konfiguration wesentlich einfacher. Für jeden Modbus-Datentyp wird einfach die gewünschte Anzahl an Elementen angegeben(siehe Abbildung 4.5). Die Konfiguration der dynamischen Kanäle entfällt, ebenso wie das I/O-Mapping. Die Adressvergabe erfolgt in diesem Fall fortlaufend, dass heißt wenn man zum Beispiel 100 *Coils* angibt, so liegen diese auf den Adressen 8192(0x2000) bis 8291(0x2063). Bei dieser Art der Konfiguration steht dem Anwender eine Bibliothek zur Verfügung, mit welcher er lesend und schreibend auf den Speicherbereich des Modbus TCP Slaves zugreifen kann. Die Bibliothek heißt *AsMbTCPS* und wird in Kapitel 4.2.3 erläutert.

Modul-Generierung

Alle Einstellungen und Konfigurationen, welche im Automation Studio gemacht werden, werden in verschiedene *BR-Module* verpackt und auf die SPS übertragen. Die Konfigura-

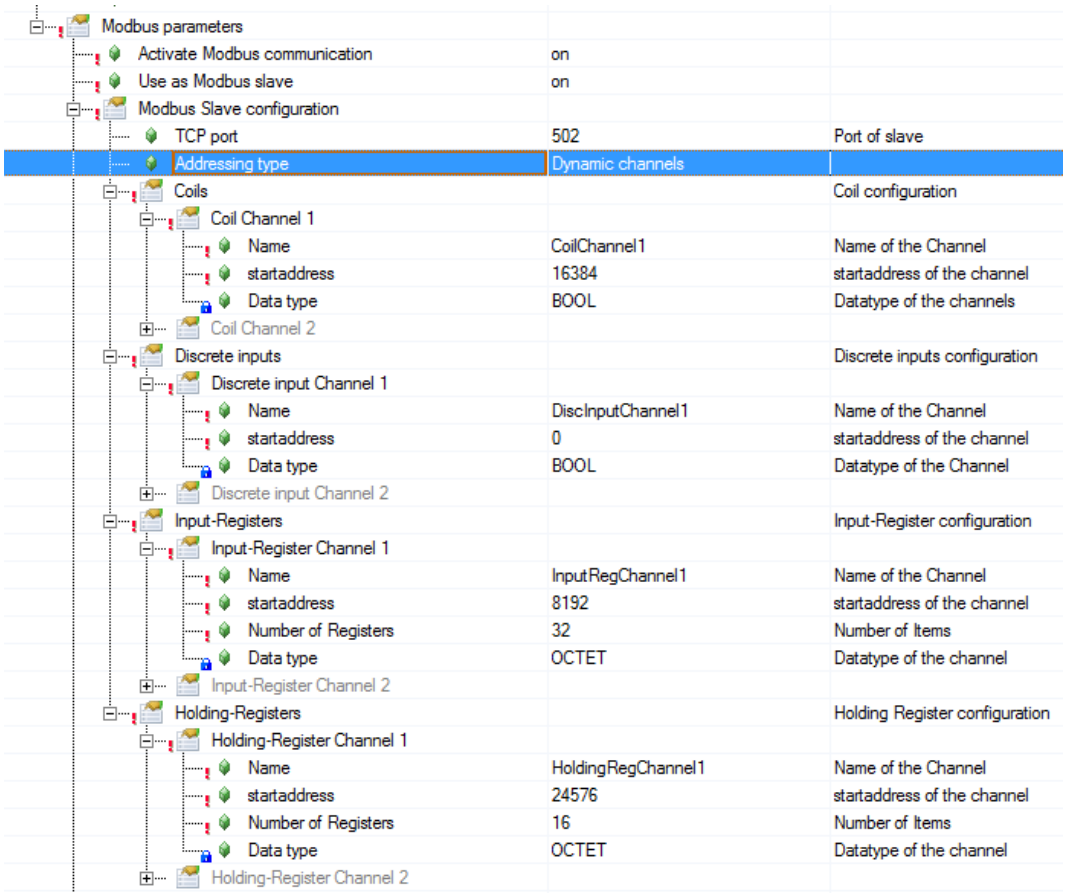

Abbildung 4.3: Konfiguration bei „dynamic Channels“

tion des Modbus TCP Slaves wurde in diesen Prozess integriert.
Die Konfiguration des Modbus TCP Slave wird in zwei *BR-Module* abgelegt:

- **arConfig**
  Wenn ein Modbus TCP Slave im Automation Studio konfiguriert wurde, so wird in der *arConfig* eine neues Modul angelegt. Der Beginn eines solchen Eintrags ist in Abbildung 4.6 dargestellt. Das „ID“-Attribut gibt den Pfad im Hardware-Baum wieder. In diesem Beispiel ist der Slave auf der Ethernetschnittstelle „IF2“ konfiguriert und es ist der erste Slave, das erkennt man am Index am Ende. Das „Hardware“-Attribut gibt an um welches Hardware-Modul es sich handelt und dient dem Automation Runtime zum Finden des entsprechenden HWD-Eintrages. Innerhalb des „Module“-Tags finden sich nun die Parameter und Kanäle, welche konfiguriert wurden.

```
<Module ID="IF2.MODBUSSLAVE_1" Hardware="MODBUSSLAVE">
```

Abbildung 4.6: arConfig: Modbus TCP Slave Module XML-Tag

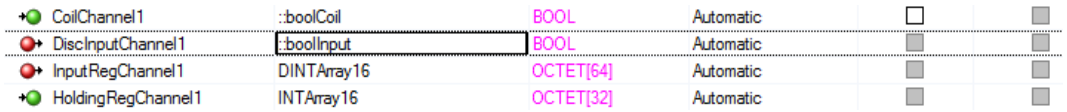

Abbildung 4.4: I/O-Mapping für einen Modbus TCP Slave

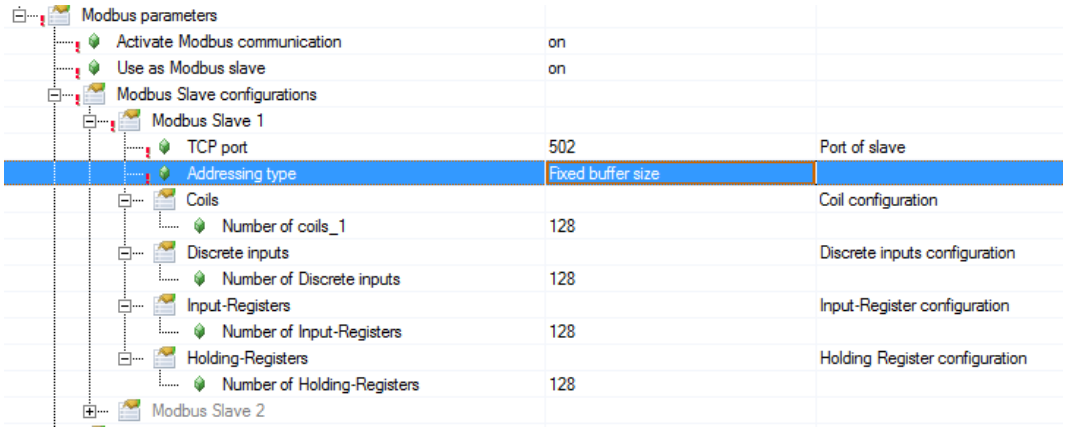

Abbildung 4.5: Konfiguration bei „fixed buffer size“

In Abbildung 4.7 sieht man die Parameter, welche für den Slave eingestellt wurden. Der Erste, mit dem „ID“-Attribut „PortNumber“, enthält als Wert den eingestellten TCP-Port und ist vom Datentyp UDINT. Der Zweite Prameter hat als „ID“ „AddressType“ und gibt an ob der Slave mit *dynamic Channels* oder *fixed buffer size* konfiguriert wurde.

```
<Parameter ID="PortNumber" Value="502" Type="UDINT" />
<Parameter ID="AddressType" Value="DynamicChannels" />
```

Abbildung 4.7: arConfig: Modbus TCP Slave Parameter

Bei der Konfiguration mit *dynamic Channels* folgen die einzelnen Kanäle. Dabei ist zwischen Kanälen für die digitalen Modbus-Datentypen, *Coil* und *Discrete Input*, und den analogen, *Input Register* und *Holding Register*, zu unterscheiden. In Abbildung 4.8 sind zwei Kanaleinträge dargestellt. Der Obere zeigt die Konfiguration eines *Coil*-Kanals. Er besteht aus vier Parametern:

- „EXT“: Gibt an dass es sich um einen Dynamischen Kanal handelt.
- „Address“: Adresse des Kanals
- „Direction“: Gibt an ob es sich um einen Eingang („IN“) oder einen Ausgang („OUT“) handelt
- „Type“: Gibt den Datentyp des Kanals an.

Der untere Eintrag zeigt hingegen einen *Input Register*-Kanal. Dieser hat mit dem Parameter „NrElements“ noch einen zusätzlichen Parameter, welcher angibt wie groß der Kanal in Byte ist.

```
<Channel ID="CoilChannel1">
  <Parameter ID="Ext" Value="/DYNAMIC" />
  <Parameter ID="Address" Value="16384" Type="UDINT" />
  <Parameter ID="Direction" Value="IN" />
  <Parameter ID="Type" Value="BOOL" />
</Channel>

<Channel ID="InputRegChannel1">
  <Parameter ID="Ext" Value="/DYNAMIC" />
  <Parameter ID="Address" Value="8192" Type="UDINT" />
  <Parameter ID="Direction" Value="OUT" />
  <Parameter ID="Type" Value="OCTET" />
  <Parameter ID="NrElements" Value="64" Type="UDINT" />
</Channel>
```

Abbildung 4.8: arConfig: Modbus TCP Slave Kanäle

Bei der Konfiguration mit *fixed buffer size* werden die vier Parameter in Abbildung 4.9 dargestellt. Jeder der vier gibt an wie viele Elemente es vom jeweiligen Modbus-Datentyp geben soll.

```
<Parameter ID="NrOfCoils" Value="128" Type="UDINT" />
<Parameter ID="NrOfDiscInputs" Value="128" Type="UDINT" />
<Parameter ID="NrOfInputReg" Value="128" Type="UDINT" />
<Parameter ID="NrOfHoldingReg" Value="128" Type="UDINT" />
```

Abbildung 4.9: arConfig: Modbus TCP *fixed buffer size* Slave Parameter

- **ioMap**
  Wird der Modbus TCP Slave mit *dynamic Channels* konfiguriert, so kann bzw. muss auch ein I/O-Mapping gemacht werden. Diese Verbindungen zwischen PV und I/O-Kanal wird im *ioMap* BR-Modul gespeichert. In Abbildung 4.10 sieht man zwei beispielhafte Einträge der *ioMap*. Der Erste beschreibt eine Verbindung zwischen dem Slave Kanal „CoilChannel1“ und der PV „boolCoil“. Der Producer befindet sich auf der Schnittstelle „IF2.MODBUSSLAVE_1“, was heißt, dass es sich um einen Modbus TCP Slave auf der Ethernetschnittstelle „IF2“ handelt. Der Konsumer, „boolCoil“, ist eine PV und wird in einem Programm verwendet, welches in Taskklasse 1 läuft, das heißt vor jedem Start des Taskklasse 1 wird der Wert des Slave Kanals auf die PV geschrieben. Der Zweite Eintrag zeigt eine weitere Verbindung zwischen einem

Slave Kanal und einer PV, welche in Taskklasse 1 verwendet wird, jedoch ist in diesem Fall die PV der Producer und ihr Wert wird nach Abschluss der Taskklasse auf den Kanal geschrieben.

```
<LN ID="%IX.IF2.MODBUSSLAVE_1.IF2.CoilChannel1" Type="BOOL">
<Prod Device="IF2.MODBUSSLAVE_1" DPName="IF2.CoilChannel1" Kind="io"/>
<Cons Device="TC#1-CPYDEV" DPName="::boolCoil" Kind="pv"/>
</LN>
```

Abbildung 4.10: ioMap: Modbus TCP I/O-Mapping

4.2.3 Automation Runtime

Im Automation Runtime wird die Modbus TCP Slave Funktionalität in einem Gerätetreiber implementiert. Im Folgenden werden die einzelnen Punkte beschrieben, welche zum Einbinden des Treibers ins System nötig sind. Als Erstes muss für die neue Komponente eine HWD-Eintrag erstellt werden. Dann wird der Aufbau und die Funktionsweise des Treibers erklärt. Als eigener Punkt wird die Bibliothek, welche bei der Konfiguration mit *fixed buffer size* benötigt wird, beschrieben und spezifiziert. Am Ende wird dann das Zusammenspiel der einzelnen Punkte während des Hochlaufs erläutert.

HWD-Eintrag

Der HWD-Eintrag für den Modbus TCP Slave Treiber ist in Abbildung 4.11 dargestellt. Die ersten drei Parameter im Modul geben Auskunft über die Hardware. Da der Modbus TCP Slave Teiber kein Hardware-Gerät ist, werden diese auf ihre Standardwerte gesetzt. Der nächste Parameter gibt an, welchen Bus das entsprechende Gerät braucht. In Fall von Modbus TCP wird Ethernet beötigt. Der Parameter mit der „ID = Autoplug“ wird verwendet, um der Geräteverwaltung zu sagen, dass das Gerät vorhanden ist, wenn es konfiguriert wurde und nicht erst am Bus gesucht werden muss. Der nächste Parameter gibt an, dass das Modul bei der Pfadangabe nur optional verwendet werden muss. Der „Classcode“-Parameter gibt an welcher Treiber für das Modul geladen werden muss, steht dieser auf dem Wert „0x200000FF“ wie in diesem Fall, so folgt ein weiterer Parameter mit welchem der Name des Treibers spezifiziert wird. In diesem Fall lautet der Treibername „DdxxIoModbusTCPSlave“. Der letzte Parameter gibt an, welchen Bus das Modul zur Verfügung stellt.

```
<Module ID="MODBUSSLAVE">
  <Parameter ID="Modno" Type="UDINT" Value="-1"/>
  <Parameter ID="UseType" Type="UDINT" Value="0"/>
  <Parameter ID="Family" Type="UDINT" Value="1"/>
  <Parameter ID="NeededBus" Value="ETHERNET"/>
  <Parameter ID="AutoPlug" Type="UDINT" Value="1"/>
  <Parameter ID="Transparent" Type="UDINT" Value="1"/>
  <Parameter ID="Classcode" Type="UDINT" Value="0x200000FF"/>
  <Parameter ID="DDriverName" Type="STRING" Value="DdxxIoModbusTCPSlave"/>
  <Parameter ID="IOSuffix" Value=":IO"/>
  <Parameter ID="OfferedBus" Value="MODBUSS"/>
</Module>
```

Abbildung 4.11: HWD-Eintrag des Treibers

Treiber Aufbau

Der Treiber wird über den *Device Manager* in das System eingebunden. Das Einbinden eines Treibers wurde bereits in Kapitel 3.5.3 erläutert und soll hier anhand des konkreten Beispiels nochmals beleuchtet werden. Der Treiber muss als Erstes eine Installations- und Deinstallationsfunktion zur Verfügung stellen. Bei der Installation des Treibers befüllt dieser eine Struktur mit Funkitonszeigern, welche die Schnittstelle zum *Device Manager* darstellt und meldet sich bei ihm an. Beim Deinstallieren wird der Treiber aus der Liste des *Device Managers* gelöscht. Beide Funktionen werden im Hochlauf in ein *BR-Modul* verpackt und das Modul wird installiert. Dabei wird die Installationsfunktion aufgerufen und meldet den Treiber mit seinem Name beim *Device Manager* an. Sollte es zur Laufzeit notwendig sein den Treiber zu deinstallieren, so muss das entsprechende *BR-Modul* deinstalliert werden. In diesem Zuge wird die Deinstallationsfunktion aufgerufen und meldet den Treiber wieder ab.

In Kapitel 3.5.3 werden auch die Funktionen für die Schnittstellen definiert. Im Fall des Modbus TCP Slave Treibers werden folgende sechs Funktionen der Schnittstelle implementiert:

- DeviceInit
- DeviceDeInit
- DeviceShutdown
- DeviceOpen
- DeviceClose
- DeviceIoControl

In der **DeviceInit** Funktion wird die Verwaltungsstruktur des Treibers allokiert und mit den Parametern aus dem *hwtree* initialisiert. Es wird das Speicherabbild des konfigurierten Modbus TCP Slave Adressraums angelegt und ein Thread gestartet, welcher den

Treiber schlussendlich startet. Im diesem Thread wird auf das Event des Ethernettreibers gewartet, welches angibt, dass die Ethernetschnittstelle fertig initialisiert und betriebsbereit ist. Danach wird noch eine Callback-Funktion am DHCP-Client eingehängt damit auf Änderungen der IP-Adresse reagiert werden kann. Anschließend wird der Verbindungsthread des Treibers gestartet.
Der Ablauf des Verbindungsthreads ist in Abbildung 4.12 dargestellt. Die Routine entspricht dem Standardverhalten eines TCP-Servers. Als Erstes wird ein Stream-Socket erzeugt. Dieser wird dann auf die Ip-Adresse und den konfigurierten Port gebunden. Hier ist zu beachten, das auf einer B&R-Steuerung mehrere Ehternetschnittstellen sein können und deshalb darf als Ip-Adresse nicht die Konstante *INADDR_ANY* verwendet werden, sondern es muss die explizite Adresse angegeben werden. Als nächstes wird der „listen"-Befehl aufgerufen, welcher den Socket vom Status „CLOSED" in den „LISTEN" Status bringt. In diesem Status werden vom Ethernetstack alle Verbindungsanforderungen akzeptiert. Im nächsten Schritt wird gewartet bis es fertig aufgebaute Verbindungen gibt, dies geschieht mit dem Aufruf der Funktion „accept". Diese wartet solange bis eine Verbindung zum Client hergestellt ist und gibt dann einen entsprechenden Socket zurück. Dieser Socket kann dann zum Kommunizieren mit dem Client verwendet werden. Im nächsten Schritt wird ein neuer Thread gestartet, welcher für die Abarbeitung der Anfragen auf diesem Socket zuständig ist. Dies bedeutet es gibt für jeden verbundenen Client einen eigenen Thread, der dessen Anfragen abarbeitet. Bevor nun näher auf den besagten Thread eingegangen wird, soll noch die Erläuterung des Verbindungsthreads abgeschlossen werden. Ist der neue Thread gestartet so beginnt der Ablauf wieder mit dem Aufruf des „listen"-Kommandos und es wird auf einen neuen Client gewartet. Auch das Verhalten im Fehlerfall ist in Abbildung 4.12 dargestellt. Sollte eine der Socket-Funktionen fehlschlagen so wird der erzeugte Socket geschlossen und die Routine beginnt von vorne. Schlägt allerdings das Erzeugen des neuen Threads fehl, so wurde die Verbindung zum Client bereits aufgebaut und wird deshalb mit dem Aufruf „shutdown" wieder beendet werden.

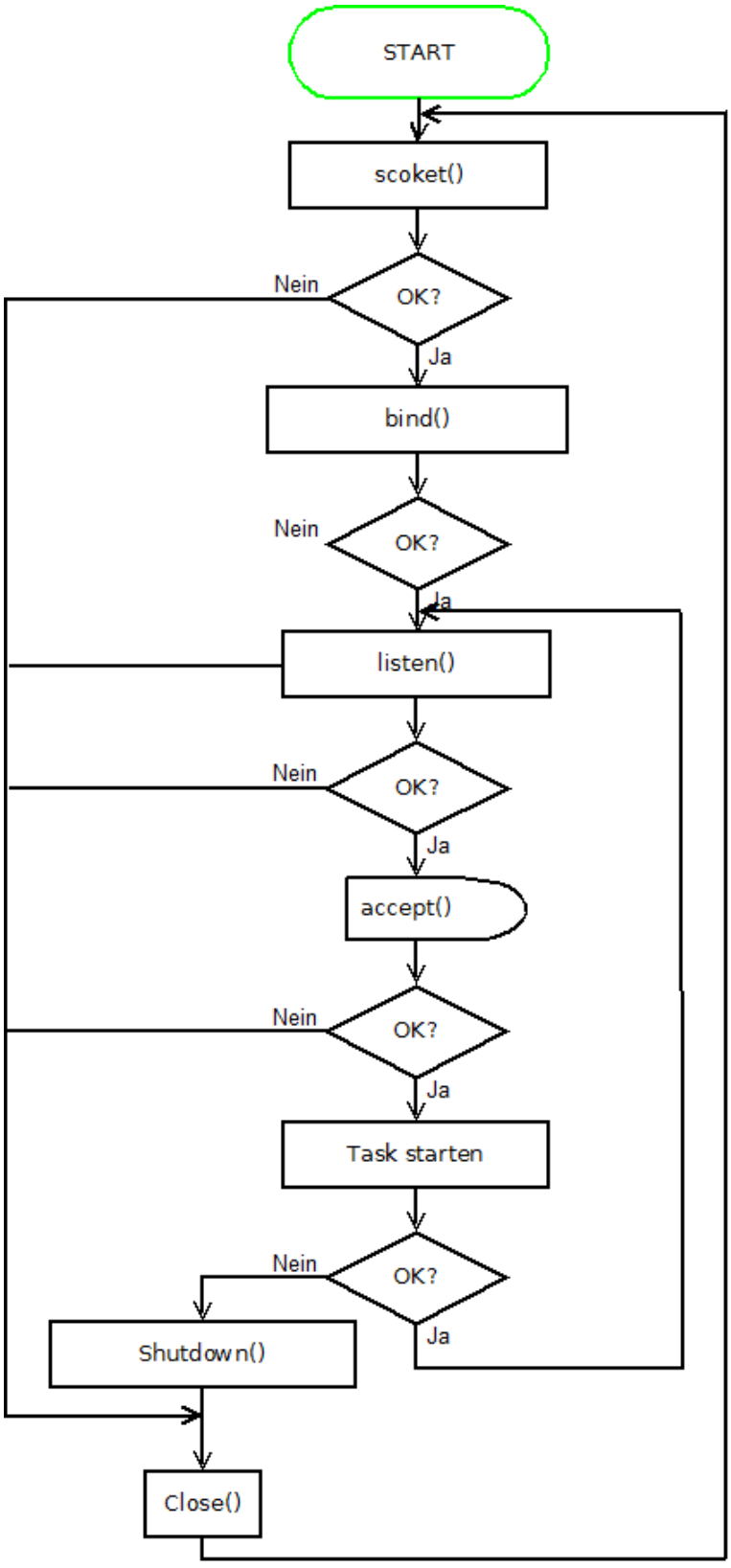

Abbildung 4.12: Ablauf: Task für Verbindungsaufbau

Die vom Verbindungsthread gestarteten Threads behandeln die Anfragen des jeweiligen Clients, aus diesem Grund werden diese Threads in der Folge Client-Threads genannt. Zu Beginn des Client-Threads werden die Socket Optionen gesetzt. In Kapitel 2.3 wurden die für Modbus TCP wichtigen Optionen bereits erklärt und deshalb werden sie hier nur kurz nochmals aufgezählt:

- SO_RCVBUF = 8192 (Eingangspuffergröße)
- SO_KEEPALIVE
- SO_LINGER ein (Socket wird sofort geschlossen)
- SO_REUSEADDR (Port sofort wieder verwenden)
- TCP_NODELAY (NAGLE-Algorithmus aus)

Nach dem Setzen der Socket-Optionen beginnt die eigentliche Abarbeitung von Anfragen. Mit dem Aufruf der Funktion „select“, wird auf Daten gewartet. Der Aufruf kehrt erst zurück wenn das angegeben Timeout abgelaufen ist oder etwas im Empfangsbuffer steht. Ist dies der Fall so wird mit Hilfe von „recv“ der MBAP-Header ausgelesen. Im Header steht die Länge des gesamten Pakets, mit der nun die restlichen Daten des Paketes ausgelesen werden können.
Das Paket wird nun in seine einzelnen Felder zerlegt. Dadurch erhält man unter anderem den Funktionscode, die Startadresse und die Anzahl an Elementen. Mit diesen Parameter ist es nun möglich, die vom Client gewünschte Antwort zu generieren. Gibt es beim Zerlegen bzw. Abarbeiten einer Anfrage einen Fehler, so wird eine Exception-Antwort generiert. Als nächster Schritt wird die generierte Antwort an den Client gesendet. Danach wird wieder mit „select“ auf neue Anfragen gewartet.
Bei Fehlern die beim Auswerten oder Ausführen der Anfrage auftreten wird eine Exception-Antwort generiert. Treten hingegen Fehler bei den Socket-Operationen auf so wird, der Socket geschlossen und der Thread beendet.

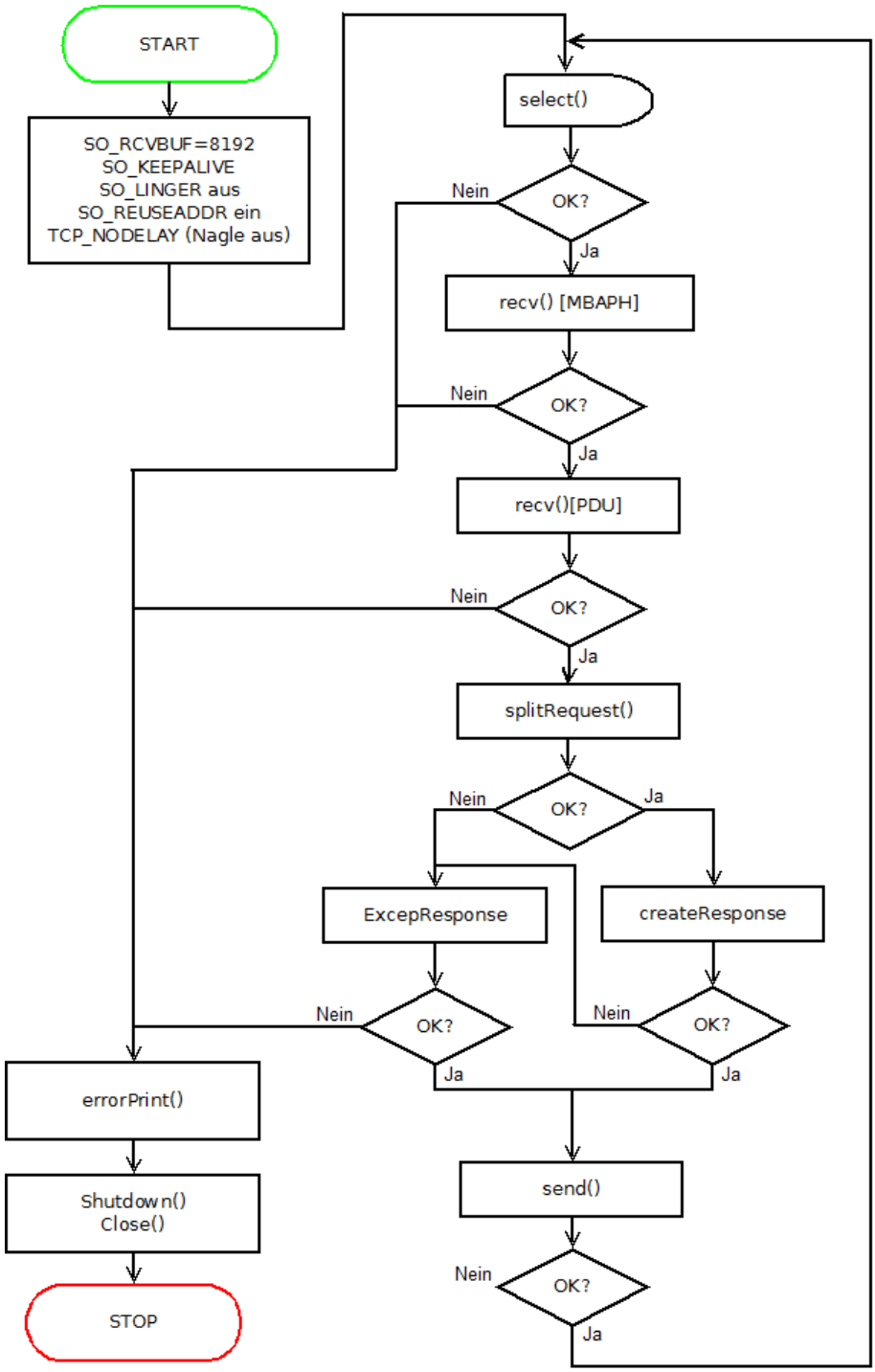

Abbildung 4.13: Ablauf: Task zum Abarbeiten von Client-Anfragen

Um dem Anwender einen besseren Einblick in die Arbeit und den aktuellen Zustand des Treibers zu geben, werden Fehler, Warnungen und Informationen in das Logbuch eingetragen. In Tabelle 4.5 ist eine Aufstellung der möglichen Logbucheinträge des Modbus

TCP Slave Treibers dargestellt. Alle Einträge des Treiber sind in der Sektion „Fieldbus“ zu finden und zusätzlich steht bei jedem Eintrag der entsprechende Thread-Name dabei.

Logbuchtext	Level	Beschreibung
allocation error	Warnung	Wird erstellt wenn beim Anfordern von Speicher ein Fehler auftritt
create socket error	Warnung	Dieser Fehlertext wird ausgegeben, wenn im Verbindungsthread, das Erzeugen des Sockets fehlschlägt
receive error on connection „Ip-Adresse des Masters“	Fehler	Wird ausgegeben wenn im Client-Thread das Empfangen von Daten nicht funktioniert
send error on connection „Ip-Adresse des Masters“	Fehler	Dieser Eintrag wird generiert wenn das Senden der Antwort an den Master nicht funktioniert
connection on „Ip-Adresse des Masters“ closed	Info	Dieser Eintrag dient dem Anwender als Information, dass die Verbindung zu einem bestimmten Master beendet wurde. Wenn vor diesem Eintrag kein Fehler des Treibers zu finden ist, so hat der Master die Verbindung beendet

Tabelle 4.5: FUB *mbSlBoolPut*: Parameter

Die nächste Funktion der Schnittstelle zum *Device Manager* ist **DeviceShutdown**. Hier werden alle laufenden Threads beendet, was bedeutet der Treiber arbeitet nicht mehr. Beim Aufruf **DeviceDeInit** wird zuerst **DeviceShutdown** aufgerufen und dann der, für die Verwaltung allokierte Speicher freigegeben.
Mit den Funktionen **DeviceOpen** erhält der Aufrufende ein Device-Handle des Treibers in der ein Zeiger auf die Verwaltungsstruktur enthalten ist. In der Regel wird dieses Handle nur verwendet um es der Funktion **DeviceIoControl** mit zu geben. Dies ist notwendig, da es von einem Treiber auch mehrere Instanzen geben kann. Mit der Funktion **DeviceClose** wird der für das Device-Handle notwendige Speicher wieder freigegeben. Die Funktion **DeviceIoControl** erwartet unter anderem das Device-Handle von DeviceOpen und einen sogenannten „Controlcode“. In der Folge werden die vom Modbus TCP Slave-Treiber unterstützten „Controlcodes“ aufgezählt:

- DMIOCTL_IO_GET_IMAGE_DESC
  Dieser Code dient dem *LinkNode Manager* zum Auslesen der Speicheradressen der Datenpunkte.

- DMIOCTL_MODSLAVE_MEMACCESS: Dieser Code ist speziell für diesen Treiber eingeführt worden. Über ihn können die FUBs der *AsMbTCPS*-Bibliothek auf den Speicher des Slaves zugreifen.

Hochlauf des Treibers

Hier soll nochmals das Einbinden und Starten des Treibers, Schritt für Schritt erklärt werden. Als Erstes muss der Treiber beim *Device Manager* angemeldet werden. Dies geschieht im Rahmen der Installation eines, zur Laufzeit generierten, *BR-Moduls*. Hierbei meldet sich der Treiber mit seinem Namen und den unterstützten Schnittstellenfunktionen, am *Device Manager* an. Durch diesen Schritt ist der Treiber nun im System verfügbar und kann beim installieren der *arconfig* gefunden werden. In Abbildung 4.6 ist der Eintrag eines Modbus TCP Slaves dargestellt. Anhand des „Hardware“-Attribut kann der entrsprechende HWD-Eintrag ausgelsen werden. In diesem findet sich unter anderem der Name des Treibers. Mit dem Namen kann nun der *Device Manager* beauftragt werden, den Treiber zu installieren. Bei der Installation wird die DevieInit-Funktion des Treiber aufgerufen. Hier werden die Threads gestarten, welche dem Modbus TCP Slave-Protokollstack abarbeiten.

AsMbTCPS

Die Bibliothek *AsMbTCPS* wurde im Rahmen dieser Arbeit erstellt und stellt die Schnittstelle vom Anwenderprogramm zum Slave-Speicher dar. Die Bibliohtek kann nur verwendet werden, wenn der Slave mit der Option *fixed buffer size* konfiguriert wurde, da ansonsten das I/O-Mapping verwendet wird.
Die Bibliothek besteht aus vier FUBs, welche synchron ausgeführt werden. Es gibt jeweils zwei FUBs für den digitalen und analogen Adressraum, wobei jeweils einer um Lesen und Schrieben verwendet werden kann. Die FUBs, deren Bezeichnung mit „Get“ endet werden zum Lesen verwendet, jene mit der Endungen „Put“ zum Schreiben. Die FUB-Bezeichnungen beginnen alle mit der Kennung „mbSl“, welche sie eindeutig zur *AsMbTCPS* zuordnet.
Die Bibliothek muss auf den Speicher des Modbus TCP Slaves zu greifen können. Hierfür wurde die Schnittstelle des Treibers zum *Device Manager* verwendet. Zu Beginn jedes FUBs wird der Treiber geöffnet. Mit dem erhaltenen Device-Handle wird dann die Funktion DeviceIOControl mit DMIOCTL_MODSLAVE_MEMACCESS als „Controlcode“. Als Übergabeparameter muss ein Zeiger auf eine Struktur übergeben werden, welche folgenden Elemente hat:

- Lesen(1) oder Schreiben(0)
- Startadresse
- Anzahl an Elementen
- Zeiger auf ein Array
- Datentyp des Array (BOOL oder UINT)

Tritt beim Ausführen des „Controls“ ein Fehler auf, so wird eine entsprechende Fehlermeldung an die Bibliothek zurückgeliefert.

In der Folge werden nun die Schnittstellen bzw. Parameter aller vier FUBs aufgelistet und kurz deren Anwendung erklärt. Bei allen FUBs wird als Datenpuffer eine Array mit 128 Elementen verwendet, d.h. es muss auch genau ein solches übergeben werden. Mit dem Parameter „nrOfItems“ kann man bestimmen wieviele der Elemente im Array verwendet werden.

- **mbSlBoolPut**
  Dieser FUB wird zum Schreiben des digitalen Adressbereiches verwendet. Es muss eine Startadresse und eine Anzahl an Elementen angegeben werden. Durch den Parameter „data“ werden die zu schreibenden Daten angegeben. Der Pfad gibt an, welcher Slave auf welcher Ethernetschnittstelle beschrieben werden soll.

I/O	Parameter	Datentyp	Beschreibung
IN	enable	BOOL	FUB wird nur ausgeführt, wenn enable 1 ist
IN	startAddress	UINT	Adresse an der begonnen wird zu Schreiben
IN	nrOfItems	USINT	Anzahl an Elementen die geschrieben werden
IN/OUT	station	STRING	Pfad zum Gerät (z.B. „IF2.MODBUSSLAVE_1“)
IN/OUT	data	BOOL[128]	Array vom Typ BOOL, welches die Daten beinhaltet
OUT	status	UINT	Fehlerwert (siehe Tabelle 4.10)

Tabelle 4.6: FUB *mbSlBoolPut*: Parameter

- **mbSlBoolGet**
  Hiermit können die Daten aus dem digitalen Adressbreich gelesen werden, auch hier gilt wieder das eine Startadresse und eine Anzahl an Elementen angegeben werden muss. Die Daten werden in das Array, welches mit dem Parameter „data“ übergeben werden muss, geschrieben.

I/O	Parameter	Datentyp	Beschreibung
IN	enable	BOOL	FUB wird nur ausgeführt, wenn enable 1 ist
IN	startAddress	UINT	Adresse an der begonnen wird zu Lesen
IN	nrOfItems	USINT	Anzahl an Elementen die gelesen werden
IN/OUT	station	STRING	Pfad zum Gerät (z.B. „IF2.MODBUSSLAVE_1“)
IN/OUT	data	BOOL[128]	Array vom Typ BOOL, in welches die Daten gelesen werden
OUT	status	UINT	Fehlerwert (siehe Tabelle 4.10)

Tabelle 4.7: FUB *mbSlBoolGet*: Parameter

- **mbSlWordPut**
  Mit diesem FUB kann der analoge Adressbereich beschriebn werden. Hier muss als Datenpuffer ein Array vom Typ UINT verwendet werden.

I/O	Parameter	Datentyp	Beschreibung
IN	enable	BOOL	FUB wird nur ausgeführt, wenn enable 1 ist
IN	startAddress	UINT	Adresse an der begonnen wird zu Schreiben
IN	nrOfItems	USINT	Anzahl an Registern die geschrieben werden
IN/OUT	station	STRING	Pfad zum Gerät (z.B. „IF2.MODBUSSLAVE_1“)
IN/OUT	data	UINT[128]	Array vom Typ UINT, welches die Daten beinhaltet
OUT	status	UINT	Fehlerwert (siehe Tabelle 4.10)

Tabelle 4.8: FUB *mbSlWordPut*: Parameter

- **mbSlWordGet**
  Der letzte FUB in der Bibliothek wird zum Lesen vom analogen Adressbereich verwendet und kopiert die Daten in ein Array vom Typ UINT.

I/O	Parameter	Datentyp	Beschreibung
IN	enable	BOOL	FUB wird nur ausgeführt, wenn enable 1 ist
IN	startAddress	UINT	Adresse an der begonnen wird zu Lesen
IN	nrOfItems	USINT	Anzahl an Registern die gelesen werden
IN/OUT	station	STRING	Pfad zum Gerät (z.B. „IF2.MODBUSSLAVE_1“)
IN/OUT	data	UINT[128]	Array vom Typ UINT, in welches die Daten gelesen werden
OUT	status	UINT	Fehlerwert (siehe Tabelle 4.10)

Tabelle 4.9: FUB *mbSlWordGet*: Parameter

Beim Ausführen eines FUBs können Fehler auftreten, welche dem Anwender über die Variable „status“ mitgeteilt werden. In Tabelle 4.10 sind alle Fehlernummern der Bibliothek aufgelistet. Die ersten beiden sind allgemeine Fehlernummern, welche von jedem FUB im System geliefert werden, die restlichen sind speziell für die *AsMbTCPS* angelegt worden.

Fehlernummer	**Konstante**	**Beschreibung**
0	ERR_OK	Kein Fehler.
65534	ERR_FUB_ENABLE_FALSE	FUB-Parameter „enable“ ungleich 1.
37450	mbSlERR_INVALID_CONFIG	Slave muss mit *fixed buffer size* konfiguriert sein.
37451	mbSlERR_INVALID_STATION	Gerät existiert nicht; Pfad falsch.
37452	mbSlERR_INVALID_ADDRESS	Adresse ungültig. Parameter „startAddress“ und „nrOfItems“ überprüfen.
37453	mbERR_NULLPOINTER	Ein Nullpointer wurde übergeben. Parameter „data“ und „station“ überprüfen.
37454	mbSlERR_INVALID_ NUMBER_OF_ITEMS	Die Anzahl an Elemnten muss zwischen 1-128. Parameter „nrOfItems“ überprüfen.

Tabelle 4.10: AsMbTCPS Fehlernummern

Kapitel 5

Schlußbemerkung und Ausblick

Das Ziel der Arbeit war es, dass eine B&R-Steuerung als Modbus TCP Slave konfiguriert werden kann. Um diese Funktionalitt geeignet umsetzten zu knnen, wurden hnliche Anwendungen von anderen Herstellern untersucht und diese Ergebnisse haben zu dieser exemplarischen Implementierung gefhrt.
Zu Beginn der Arbeit wurde das Modbus-Protokoll studiert und analysiert, um dessen Funktionalität überblicken zu können. Mit diesem Wissen wurde die Anwendung des Protokolls mit dem TCP/IP-Protokoll betrachtet. Als nächster logischer Schritt folgte die Analyse des Bestandsystems um eine sinnvolle Einbindung in das System durchführen zu können. Hier wurden zuerst das Echtzeit-Betriebssystem auf der Steuerung, die sogenannte *Automation Runtime*, betrachtet. Die wichtigsten Komponenten wurden untersucht und der Schluss wurde gezogen, dass die Funktionalität des Modbus TCP Slaves in einem Gerätetreiber implementiert werden sollte. Mit dieser Erkenntnis konnte nun die Konfiguration für den Anwender betrachtet werden. Der Anwender kann das System über eine Desktopanwendung mit dem Namen *Automation Studio* nach seinen Vorstellungen konfigurieren. In diese Software musste auch die Konfiguration für den Modbus TCP Slave einfließen. Da das System bereits die Konfiguration eines Modbus TCP Masters unterstützte war es naheliegend, die des Slave parallel dazu anzubieten. Der Kunde findet dadurch die Konfiguration der Modbus TCP Funktionalität in der Konfiguration der Ethernetschnittstelle. Im Automation Runtime wird die Konfiguration geladen und der Modbus TCP Slave-Treiber, sofern konfiguriert, gestartet. Im Treiber wird für jeden Master ein eigener Thread gestartet, welcher dessen Anfragen bearbeitet. Dieser Art der Einbindung ist sehr modular und bei eventuellen Erweiterungen des Modbus TCP Slaves muss nicht das ganze System, sondern nur der Treiber angepasst werden.

Die neue Funktionalität wurde bereits den Kunden weltweit präsentiert und es gab sehr viele positive Rückmeldungen. In diesen Rückmeldungen sind auch einige Wünsche enthalten, die aufgenommen und diskutiert worden sind. Hierbei geht es vor allem um die Unterstützung des Unit-Identifier des MBAP-Headers. Zum einen soll der Modbus TCP Slave als Gateway zum seriellen Modbus, welcher bereits unterstützt wird dienen, das ist vor allem bei älteren Anlagen von Bedeutung, da hier manchmal noch serielle Busse zum

Einsatz kommen und bei der Erweiterung dieser Anlagen das funktionierende System nicht verändert werden soll. Zum anderen wollen einige Kunden, dass der Modbus TCP Slave nur Anfragen mit spezieller Unit-Id bearbeiten soll, dies ermöglicht, wie die Verwendung von unterschiedlichen Ports, mehrere Modbus TCP Slaves auf einer Ethernetschnittstelle.

Literaturverzeichnis

[AG02] Thomas Havlovec Anton Graf. Spezifikation Taskklassenschaufeldevice. Technical Report 8, Bernecker + Rainer, August 2002.

[Ber98] Bernecker + Rainer. B&R Modul Beschreibung. Technical Report 1, Bernecker + Rainer, August 1998.

[Ber09] Bernecker + Rainer. Modbus/tcp anwenderhandbuch. http://www.br-automation.com/downloads_br_productcatalogue/ BRP4440000000000000099814/MAMB-GER.pdf, 2009.

[Ber13] Bernecker + Rainer. Automation Studio Help V4.1, 2013.

[Coh81] D. Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, 1981.

[Gra08] Anton Graf. Spezifikation Sequentielles Scheduling. Technical Report 1, Bernecker + Rainer, August 2008.

[Gra11] Anton Graf. Spezifikation IO Device Driver. Technical Report 1, Bernecker + Rainer, February 2011.

[Hav02] Thomas Havlovec. Spezifikation Linknodes. Technical Report 3, Bernecker + Rainer, August 2002.

[IEC03] Programmable controllers Part 1: General information. Norm IEC 61131-1/Ed.2, IEC, February 2003.

[IEC13] Programmable controllers Part 3: Programming languages. Norm IEC 61131-3/Ed.3, IEC, February 2013.

[Org13] Modbus Organization. Modbus_Application_Protocol_V1_1b3. Technical report, Modbus Organization, nov 2013.

[SG06] Wiedemann Bernhard Schnell Gerhard. *Bussysteme in der Automatisierungs- und Prozesstechnik*. Friedr. Vieweg & Sohn Verlag, 6 edition, 2006.

[Ste98] W. Richard Stevens. *Programmieren von UNIX-Netzwerken*. Carl Hanser Verlag, 2 edition, 1998.