Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Stand: 10.02.2016

Vector Informatik GmbH

- kurze Fassung -

Evaluation der Integration von Feldbusgeräten über Ethernet in Vector CANoe

Evaluation how to integrate Ethernet connected fieldbus I/O in Vector CANoe software.

Jonathan Wendeborn (35852)
me@jonny007-mkd.de

Inhaltsverzeichnis

1 Problembeschreibung

Die Komplexität einer Flugzeugkabine hat sich in den letzten Jahrzehnten drastisch erhöht: Geräte zur Kommunikation der Kabinenbesatzung untereinander, mit dem Cockpit und den Passagieren, Kabinen- und Einzelplatzbeleuchtung, Rauchdetektion und das Feuerlöschsystem im Cargobereich müssen angesprochen und verwaltet werden. Alle genannten Geräte werden bei Flugzeugen von *Airbus S.A.S. (Airbus)* im *Cabin Intercommunication Data System* (*CIDS*) zusammengefasst, neben dem es noch viele weitere gibt. Alle diese Systeme sollen und müssen im Flug und am Boden zuverlässig funktionieren und deshalb vor Auslieferung der Flugzeugreihe ausführlich getestet werden. Dies geschieht bei Kabinensystemen der *A380* von *Airbus* in Hamburg-Finkenwerder. Die Ansteuerung der einzelnen Geräte und die Kommunikation untereinander funktioniert heutzutage über diverse Bussysteme. Da deshalb z.B. aufgrund ungewöhnlicher Systemzustände oder versteckter Implementierungsfehler unerwünschtes Verhalten auftreten könnte, ist es mit einem einfachen „geht, geht nicht“-Test nicht getan. Nötig ist eine umfassende Testprozedur; am besten sogar mit Hilfe einer voll integrierten Testumgebung, mit der alle Eingänge beliebig verändert und die Zustände aller Ausgänge überwacht werden können. Wünschenswert ist eine Anlage, die diese Tests vollautomatisch (z.B. über Nacht) durchführen kann.

In einem modernen Großraumflugzeug von *Airbus* beträgt alleine die Anzahl der elektrischen Ein- und Ausgänge der Flugzeugkabine ca. 15.000, welche in Hamburg-Finkenwerder zurzeit in großen, proprietären Schaltschränken verarbeitet werden. Die an diesem Standort für Betrieb, Fortbestand und Entwicklung der Testumgebung zuständigen Projektleiter Andre Schäfer und Daniel Bilir entschieden gemeinsam mit Jörn Haase von der *Vector Informatik GmbH (Vector)*, dass zukünftig die Software *CANoe* als zentrales Element zur Steuerung und Überwachung des Systems in der Testumgebung verwendet werden soll. Für Überwachung und Einbindung der Ein- und Ausgänge in *CANoe* sollen Standard-Busklemmen verschiedener Hersteller verwendet werden, um die Preise durch verstärkte Konkurrenz zu senken, während die Klemmen durch offene Standards zueinander kompatibel bleiben. Die ins Auge gefassten Hersteller *WAGO Kontakttechnik GmbH & Co. KG* (*Wago*) und *Bernecker + Rainer Industrie Elektronik Ges.m.b.H.* (*B+R*) zeichnen sich dadurch aus, dass die zur Verfügung stehenden Geräte den relativ geringen Leistungsanforderungen hinsichtlich der Genauigkeit der Werte und des Zeitpunktes entsprechen und somit im Gegensatz zum *VT*-System der *Vector Informatik GmbH* nicht in einem so hohen Preissegment liegen.

Die Projektleiter entschieden zudem, dass das zukünftige Testsystem auf Ethernet basieren sollte. Da *CANoe* bisher außer TCP/IP und UDP/IP keine hohen Ethernet-Protokolle unterstützt, ergibt sich daraus die Aufgabe dieser Bachelorthesis: Ein von den Busklemmen unterstütztes, Ethernet-basiertes Kommunikationsprotokoll auszuwählen und dessen Integration in *CANoe* auszuloten, zu entwerfen und falls möglich durchzuführen, um die einfache Ansteuerung der an die Busklemmen angeschlossenen Geräte in *CANoe* zu ermöglichen. Darüber hinaus sollen die Grenzen des entworfenen Bussystems hinsichtlich Zuverlässigkeit und Belastbarkeit überprüft und weitere eventuelle Einschränkungen oder Verbesserungsmöglichkeiten entdeckt werden.

Wie sich im Verlauf dieser Thesis herausstellen wird, ist Modbus-TCP ein geeignetes Protokoll zur Lösung dieser Aufgabe. Es ist einfach zu implementieren und fügt nur einen geringen Overhead hinzu; darüber hinaus lässt es sich gut in *CANoe* integrieren und ist auf einem geswitchten Ethernet leistungsfähig genug für die Übertragung mehrerer Millionen Ein- und Ausgangswerte pro Sekunde; mehr, als *CANoe* derzeit auf Standard-Hardware zu verarbeiten mag und heute sinnvoll erscheint.

2 Problemanalyse

Im Folgenden soll die gestellte Aufgabe ausführlich analysiert und das verwendete Protokoll sowie die Software *CANoe* beschrieben werden.

2.1 Problemstellung

Von Airbus wurden die Busklemmen *Wago 750-881* und *B+R X20BC0087* zur Verfügung gestellt. An diese können IO-Module angeschlossen werden, welche die Umwandlung zwischen den Daten und den physikalischen Zuständen vornehmen. Diese Busklemmen sollen über ein klassisches Ethernet-Netz von der Software *CANoe* von *Vector* angesprochen werden können. Hauptziel ist das Lesen und Schreiben vieler komplexer und digitaler Ein- und Ausgänge. Auf zeitliche Genauigkeit unter 100 ms wird hier kein besonderer Wert gelegt. Gewünscht ist, den Konfigurationsaufwand für den Endnutzer nach einer Art Plug & Play möglichst gering zu halten. Ebenso soll dieser keine Kenntnisse zu den Details der Implementierung besitzen müssen, um mit den Busklemmen in *CANoe* zu arbeiten.

Das Bild kann zurzeit nicht angezeigt werden.

Abbildung 1: Der Modbus-Buscontroller X20BC0087 von *B+R*. Alle Rechte bei *B+R*.

Gefordert ist somit eine Abstraktionsschicht (Modbus-Stack), welche die Daten der angeschlossenen Busklemmen nahtlos in *CANoe* integriert und somit die Nutzung genau so leicht ermöglicht wie bereits von *CANoe* unterstützte Systeme.

2.2 Feldstudie: Mögliche Bussysteme

In einem ersten Schritt sollen verfügbare Bussysteme verglichen und entsprechend der Anforderungen bewertet werden. Diese sind:

1. Nutzung eines Ethernet-basierten Bussystems,
2. welches von den angedachten Busklemmen der Hersteller *Wago* und *B+R* unterstützt wird,
3. keine oder nur geringe Lizenzierungskosten verursacht,
4. recht einfach zu implementieren ist und
5. möglichst mehrere 10.000 digitale und komplexe Ein- und Ausgänge alle 100 ms übertragen kann.

Abbildung 2: Der programmierbare Feldbuscontroller *750-881* von *Wago*. Alle Rechte bei *Wago*.

Von den Busklemmen *Wago 750-881* (s. Abbildung 2) und *B+R X20BC0087* (s. Abbildung 1) werden Ethernet-basiert nur die beiden Protokolle *EtherNet/IP* und *Modbus* unterstützt. Darüber hinaus wurden noch weitere Bussysteme betrachtet, um mögliche Vorteile durch eine Wahl anderer Busklemmen auszuräumen. Eine vollständige Auflistung findet sich im Anhang (s. Kapitel 9.1). Im Folgenden sollen nur die drei Protokolle *VARAN*, *EtherNet/IP* und *Modbus* näher behandelt werden, da diese beinahe alle Anforderungen erfüllen.

2.2.1 VARAN – Versatile Automation Random Access Network

VARAN wird hauptsächlich im Bereich der Automatisierungstechnik eingesetzt und spezifiziert die ersten beiden OSI-Layer. Verantwortlich für die Pflege des Protokolls ist die *VARAN-BUS-Nutzerorganisation* (*VNO*).

Das bisher noch nicht standardisierte Protokoll folgt dem Master-Slave-Prinzip und ermöglicht harte Echtzeit (< 100 µs) bei zu IEEE 802.3 (Ethernet) kompatibler physikalischer Schicht. Deshalb ist eine Implementierung in FPGAs und ASICs üblich. Die spezifizierten Operatoren ermöglichen einen direkten Zugriff auf große Teile des Speichers und die darin liegenden Prozessdaten. Die nötigen Spezifikationen und Unterlagen werden von der *VNO* nach Aufnahme in die Organisation kostenfrei zur Verfügung gestellt; jedoch werden für verkaufte Hardware-Komponenten Lizenzgebühren berechnet[1].

Aufgrund der Echtzeitfähigkeit, des damit verbundenen höheren Implementierungsaufwands, der fehlenden Unterstützung durch die geplanten Busklemmen und die Lizenzgebühren stellt VARAN jedoch hier keine optimale Lösung dar.

2.2.2 EtherNet/IP – EtherNet Industrial Protocol

Dieses Bussystem folgt ebenfalls dem Master-Slave-Prinzip und ermöglicht weiche Echtzeitanwendungen (< 10 ms), kann jedoch durch die Erweiterung *MotionSync* optimiert und somit in fast allen Bereichen der Automatisierungstechnik eingesetzt werden. Es setzt auf TCP/IP und UDP/IP auf und ermöglicht den einfachen zyklischen Austausch von Daten.

EtherNet/IP wird von der *Open DeviceNet Vendor Association* (ODVA) verwaltet und ist in der Feldbusnormenreihe IEC 61158 standardisiert. Um Produkte mit EtherNet/IP erstellen und vertreiben zu dürfen, werden jedoch Lizenzgebühren fällig[2]. Deshalb und aufgrund der hohen Komplexität des *Common Industrial Protocol*s fiel die Wahl auf Modbus.

2.2.3 Modbus

Dieses Protokoll setzt ebenfalls auf das Master-Slave-Prinzip und hat sich durch seine offene und freie Verfügbarkeit zu einem industriellen Standard entwickelt. Modbus wurde für die serielle Schnittstelle als physikalische Schicht entwickelt, später für die Verwendung von TCP/IP spezifiziert und in IEC 61158 standardisiert[3]. Aufgrund seiner Bedeutung und Verbreitung wurde ihm sogar der TCP-Port 502 fest zugeteilt. Modbus ermöglicht lediglich den bit- und wortweisen Lese- und Schreibzugriff auf das Prozessabbild und ist somit sehr einfach zu implementieren.

[1] http://www.varan-bus.net/de/implementierung.htm
[2] http://www.control.com/thread/1026200383
[3] http://www.feldbusse.de/Normung/protokollfamilien.shtml

Die *Modbus Organization*, eine Gruppe von unabhängigen Nutzern und Herstellern, entwickelt den freien Standard sowie einen Konformitätstest weiter und unterstützt kostenpflichtig bei der Implementierung durch Code-Beispiele und spezielle Tools[4].

Durch die Verwendung von TCP/IP auf Ethernet, die kostenfreie Bereitstellung der Spezifikation, die hohe Simplizität des Protokolls (vgl. Kapitel 2.3) und die Unterstützung durch die Busklemmen von *Wago* und *B+R* ist Modbus-TCP der ideale Kandidat für die gestellte Aufgabe.

2.3 Beschreibung des zu verwendenden Protokolls Modbus-TCP

Modbus wurde 1979 von Gould-Modicon entwickelt und setzte auf serielle Schnittstellen auf (RS-232 und RS-485). Seit 2007 ist auch der Versand von Modbus-Datagrammen über TCP/IP (und Ethernet) als Modbus-TCP spezifiziert. Somit müssen im Modbus-TCP-Standard keine Verbindungen, Adressen und Timeouts behandelt werden[5]. Obwohl das Protokoll nur für TCP/IP spezifiziert wurde, unterstützen beide Hersteller der Busklemmen ebenfalls den Versand der Modbus-Datagramme über UDP. Da Modbus-TCP jedoch ein eigenständiger Name ist, wird er in dieser Arbeit sowohl für die Übertragung mittels TCP/IP als auch mittels UDP/IP verwendet.

Nachfolgend soll auf wichtige Aspekte des Protokolls eingegangen werden. Im Verlauf dieser Arbeit wird – falls nicht anders deklariert – nur noch Modbus-TCP behandelt und somit Modbus analog zu seiner Ethernet-Variante verwendet.

2.3.1 Datenmodell

Modbus kennt nur zwei verschiedene Datentypen („digital“, 1 Bit, und „komplex“, 16 Bit), die nochmals in les- und schreibbar aufgeteilt werden (vgl. Tabelle 1). In der Spezifikation wird für jeden Datentyp ein eigener Name verwendet (vgl. ebd.). Die Busklemmen von *Wago* vereinfachen dies weiter und unterscheiden bei Lesevorgängen nicht zwischen les- und schreibbaren Ausgängen. Sie liegen lediglich an verschiedenen Adressen (vgl. Kapitel 2.3.2). Da die Busklemme von *B+R* bei digitalen Daten zwischen schreiben und lesen unterscheidet, ist jedoch keine Vereinfachung durch Implementierung lediglich eines Befehls möglich. Alles in Allem ist aufgrund der geringen Anzahl der Datentypen nur eine recht geringe Anzahl von Befehlen möglich und nötig.

Data type name	Object type	Type of	This type of data can be
Discrete Input	Single bit	Read-Only	provided by an (external) I/O system
Coil	Single bit	Read-Write	alterable by an application program
Input Register	16-bit word	Read-Only	provided by an (external) I/O system
Holding Register	16-bit word	Read-Write	alterable by an application program

Tabelle 1: Datentypen bei Modbus (nach MAPP, Kapitel 4.3)

In digitalen Daten werden meistens bestimmte Spannungspegel kodiert, welche z.B. Geräte einschalten können. Dass jedoch auch hierbei Unterschiede möglich sind, zeigen *B+R* und *Wago* recht deutlich: Bei *B+R* wird der negative Pol der Versorgungsspannung als ‚1‘ kodiert, bei *Wago* hingegen der positive Pol.

Komplexe Daten können – wie es der Name bereits andeutet – eine Vielzahl an Werten repräsentieren: Dies kann ein Messwert eines analogen Zustandes sein, es kann jedoch auch der Wert eines Zählers oder das letzte empfangene bzw. zu sendende Wort einer seriellen Schnittstellen

[4] http://modbus.org/docs/Toolkit_a.pdf
[5] http://de.wikipedia.org/wiki/Modbus

sein. Die Wandlung zwischen den Daten und physikalischen Werten übernimmt das jeweilige Modul der Busklemme.

2.3.2 Prozessabbild

Die Erstellung des Prozessabbilds einer Busklemme wird zwar nicht in der Modbus-Spezifikation beschrieben, ist jedoch von großer Bedeutung. Die Hersteller folgen hierbei ihren eigenen Vorstellungen von einer sinnvollen Anordnung.

Beide betrachteten Busklemmen erkennen beim Hochfahren die angeschlossenen Module und weisen ihren Daten Adressen im Speicher zu. Die Adressräume der komplexen und der digitalen Module sind grundsätzlich getrennt, meistens sind die digitalen Daten jedoch auch im komplexen Prozessabbild abrufbar.

Da die beiden Busklemmen wie in Kapitel 2.3.1 beschrieben teilweise nicht zwischen les- und schreibbaren Ein- und Ausgängen unterscheiden, es aber bei diesen Klemmen möglich ist, alle Ausgänge auch wieder zurück zu lesen, stellen die Klemmen die Ausgänge im Eingangsprozessabbild an einer gesonderten (höheren) Adresse zur Verfügung. Eine Ausnahme bilden hier bei *B+R* die digitalen Ausgänge, welche über den gesonderten Funktionscode `0x01` (vgl. Kapitel 2.3.4) zurück gelesen werden können.

Zusätzlich zu den Ein- und Ausgängen stellen beide Modbus-Implementierungen weitere Register-Adressbereiche zur Verfügung, in denen der Status und die Konfiguration der Busklemmen und der angeschlossenen IO-Module ausgelesen und teilweise auch verändert werden können. Auch die Anzahl der verfügbaren digitalen und komplexen Ein- und Ausgänge sind jeweils mit der normalen Lesefunktion abrufbar. Darüber hinaus können weitere Informationen abgerufen und Aktionen ausgeführt werden, auf die hier aufgrund der Vielzahl und Unterschiedlichkeit der beiden Hersteller nicht näher eingegangen werden soll. Sie sind in den Handbüchern beider Geräte ausführlich dokumentiert.

Adressbereich	Anzahl	Beschreibung der Zugriffsmethode	Modbus Funktionen
0x0000 - 0x07FF	2048	Analoge Eingänge lesen	3, 4, 23
0x0800 - 0x0FFF	2048	Analoge Ausgänge lesen/schreiben	3, 4, 6, 16, 23
0x1000 - 0x1FFF	4096	Systemparameter lesen/schreiben	3, 4, 6, 16, 23
0x2000 - 0x23FF	1024	Digitale Eingänge lesen	3, 4, 23
0x2400 - 0x27FF	1024	Digitale Ausgänge lesen/schreiben	3, 4, 6, 16, 23
0x2800 - 0x29FF	512	X2X Link Netzwerkstatus lesen	3, 4, 23
0x2A00 - 0x2BFF	512	Analoger bzw. digitaler Ausgangsstatus lesen	3, 4, 23
0x2C00 - 0x9FFF	29696	Reserviert lesen	3, 4, 23
0xA000 - 0xAFCF	4048	Moduldaten (Index) lesen/schreiben	3, 4, 6, 16, 23
0xAFD0 - 0xAFFF	48	Reserviert (Daten für 3 Module) lesen	3, 4, 23
0xB000 - 0xBFFF	4096	Moduldaten (Parameter) lesen/schreiben	3, 4, 6, 16, 23
0xC000 - 0xDFFF	8192	Modul-Konfigurationsdaten lesen/schreiben	3, 4, 6, 16, 23
0xE000 - 0xFFFF	16384	Reserviert lesen	3, 4, 23

Abbildung 3: Das Wort-Prozessabbild des *B+R X20BC0087* (nach BRMbHr, Tabelle 7)

2.3.3 Modbus-Application-Header

Um im Modbus-Netz weitere Funktionen wie z.B. eine Modbus-Gateway zur Verfügung zu stellen und die Verarbeitung für das empfangende Programm zu vereinfachen, wird jedem Paket der sogenannte *Modbus-Application-Header* vorangestellt (s. Tabelle 2). Er belegt acht Byte und ist damit

im Vergleich zu den Headern von TCP und IP relativ klein. Da im Sinne der hier gestellten Aufgabe kein Multiplexing und keine Gateways nötig sind, können die beiden Felder „Protocol ID“ und „Unit ID“ ohne weiteres fest auf 0 gesetzt werden.

Offset	Size	Field	Value hex	Description
0	2	Transaction ID	0xnn nn	Request identifier, copied into response
2	2	Protocol ID	0xnn nn	Used for intrastream multiplexing. Modbus = 0x00 00
4	2	Length	0xnn nn	Number of **following** bytes
6	1	Unit ID	0xnn	ID of remote slave on other bus (used for Gateways)
7	1	Function Code	0xnn	Function Code (vgl. Kapitel 2.3.4)

Tabelle 2: Struktur des *Modbus-Application-Header*s. Der Funktions-Code gehört laut Spezifikation zwar nicht zum Header, ist jedoch Teil eines jeden Modbus-Telegramms und deshalb hier hinzugefügt worden.

Ein Modbus-Paket könnte aufgrund des Length-Feldes bis zu 65.542 Bytes belegen, jedoch wurde von der Spezifikation für RS-485 eine maximale Payload[6]-Größe von 253 Bytes geerbt (um die Verwendung von Modbus-Gateways einfach zu halten). Somit ergibt sich für Modbus-TCP als Payload für TCP oder UDP eine maximale Größe von 260 Bytes.

Aufgrund dieser Beschränkung und weiterer Einschränkung in der Spezifikation können in einem Paket gleichzeitig maximal 2.000 Bits bzw. 125 Worte gelesen oder 1.968 Bits bzw. 123 Worte geschrieben werden. Letzteres sind weniger, da beim Schreibvorgang in der Anfrage noch zusätzlich vier Bytes von der Start-Adresse und der Anzahl der zu beschreibenden Elemente belegt werden, während beim Lesevorgang in der Antwort die Daten ohne diese Angaben gesendet werden.

2.3.4 Function Codes

Die Modbus-Spezifikation definiert insgesamt 21 öffentliche Funktionen, die durch spezielle „Function Codes“ repräsentiert werden. Diese bestimmen ebenfalls den Aufbau der gesendeten Pakete. Ein kurzer Auszug mit den wichtigsten, in dieser Arbeit implementierten Funktionen:

`0x01` Read Coils
Lesen der Ausgangsbits (DO)

`0x02` Read Discrete Inputs
Lesen der Eingangsbits (DI)

`0x03` Read Holding Registers
Lesen der Ausgangsregister (AO)

`0x04` Read Input Registers
Lesen der Eingangsregister (AI)

`0x05` Write Single Coil
Setzen eines einzelnen Bits

`0x06` Write Single Register
Setzen eines einzelnen Registers

`0x0F` Write Multiple Coils
Setzen mehrerer Bits

`0x10` Write Multiple Registers
Setzen mehrerer Register

`0x16` Mask Write Register
UND- und ODER-Maskieren eines Registers

`0x17` Read & Write Multiple Registers
Schreiben und lesen mehrerer (unterschiedlicher) Register gleichzeitig

[6] Payload: Die reinen Daten in einem Paket, ohne Header oder Ähnlichem

Darüber hinaus sind für Modbus unter anderem Funktionen zum Lesen von Dateien und zur Diagnostik auf der seriellen Schnittstelle definiert. Es ist zu beachten, dass *Wago* und *B+R* mehr als die Hälfte der spezifizierten Funktionen nicht oder anders implementiert haben. Dies ist möglich, da Modbus ein freier und offener Standard ist. So wurden beispielsweise die Funktionen `0x03` und `0x04` (sowie bei *Wago* `0x01` und `0x02`) vereint und erfüllen die jeweils gleiche Aufgabe (vgl. Kapitel 2.3.1). Ebenso ist es bei den untersuchten Klemmen möglich, die digitalen Eingänge auch als Register zu lesen und zu schreiben. Zusätzlich reservieren die beiden Hersteller gewisse Register-Adressbereiche für Konfigurationsparameter und Geräteinformationen, weshalb die eigens hierfür gedachte Funktion nicht implementiert wurde (vgl. Kapitel 2.3.2).

Das erste Bit des Function-Codes ist für die Angabe reserviert, ob bei der Bearbeitung der Anfrage ein Fehler (eine „Exception“) aufgetreten ist. Wird z.B. eine Anfrage mit dem Function-Code `0x03` gesendet und mit `0x83` beantwortet, so konnte diese Anfrage nicht fehlerfrei beantwortet werden. Im Telegramm folgt dann lediglich noch ein sogenannter „Exception Code“, welcher den Grund des Fehlers genauer spezifiziert. Dieser kann laut Spezifikation folgende Werte annehmen:

`0x01`	Unbekannter Funktionscode
`0x02`	Ungültige Adress-Angabe
`0x03`	Ungültiger Datenwert
`0x04`	Unbekannter Server-Fehler
`0x05`	Bestätigung. Die Bearbeitung der Anfrage benötigt mehr Zeit
`0x06`	Der Server ist überlastet
`0x0A`	Gateway-Pfad ist nicht verfügbar
`0x0B`	Das von der Gateway angesprochene Zielgerät hat nicht geantwortet

Die Spezifikation von Modbus räumt die Möglichkeit ein, eigene Function-Codes zu implementieren. Dies wurde von den beiden Herstellern jedoch nicht in Anspruch genommen.

				Function	
				code	sub code
Data access	Bit Access	Phys. Inputs	**Read Descrete Inputs (DI)**	0x02	
		Internal Bits or Physical coils	**Read Coils (DO)**	0x01	
			Write Single Coil (DO)	0x05	
			Write Multiple Coils	0x0F	
	16 bits access	Phys. Inputs	**Read Input Register (AI)**	0x04	
		Internal Registers or Phys. Output Registers	**Read Holding Register (AO)**	0x03	
			Write Single Register (AO)	0x06	
			Write Multiple Registers	0x10	
			Read & Write Multiple Registers	0x17	
			Mask Write Register	0x16	
			Read FIFO queue	0x18	0x00-0x12;0x16
	File record access		**Read File record**	0x14	
			Write File record	0x15	
Diagnostics			**Read Exception Status**	0x07	
			Diagnostic	0x08	
			Get Com event counter	0x0B	
			Get Com event log	0x0C	
			Report Slave ID	0x11	
			Read device ID	0x2B	0x0E
Other			**Encapsulated Interface Transport**	0x2B	0x0D; 0x0E
			CANopen General Reference	0x2B	0x0D

Abbildung 4: Vollständige kategorisierte Liste der Modbus-Function-Codes (nach MAPP, Kapitel 5.1). Grau hinterlegt: In dieser Arbeit nicht implementiert.

2.4 CANoe

„*CANoe* ist das vielseitige Werkzeug für die Entwicklung, den Test und die Analyse von ganzen Steuergerätenetzwerken, aber auch von einzelnen Steuergeräten. Es unterstützt Netzwerk-Designer, Entwicklungs- und Testingenieure bei OEMs und Zulieferern im kompletten Entwicklungsprozess – von der Planung bis hin zur Inbetriebnahme kompletter verteilter Systeme oder einzelner Steuergeräte.“[7] Der Simulations- und Messaufbau in *CANoe* wird konfiguriert, indem Netzwerkknoten in ein Bussystem eingefügt werden (vgl. Abbildung 6). Diese Knoten können Nachrichten mit Signalen empfangen (s. Kapitel 2.4.2) und es kann ein CAPL-Programm hinterlegt werden, welches die Reaktionen auf bestimmte Ereignisse vornimmt (siehe Kapitel 2.4.1).

CANoe ermöglicht einerseits den Zugriff auf verschiedene angeschlossene Bussysteme: Es können gleichermaßen Nachrichten empfangen und gesendet und somit der Bus auf allen Netzwerk-Ebenen beobachtet und mit entfernten Geräten kommuniziert werden. *CANoe* kann damit z.B. als Steuerzentrale eingesetzt werden. Andererseits können in *CANoe* sowohl Bussysteme als auch Geräte simuliert werden. Im Hinblick auf die Möglichkeiten zur Untersuchung rückgekoppelter Systeme wird es somit möglich, sowohl die Umwelt als auch das System zu simulieren und damit potenzielle Fehler leichter zu entdecken.

[7] Produktinformation *CANoe* (VecPI), S. 4

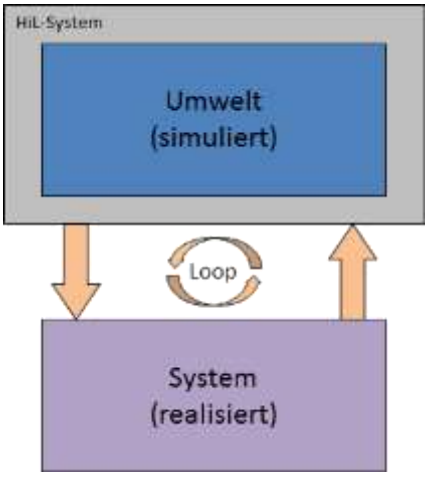


Abbildung 5: Schematische Darstellung der Untersuchung eines rückgekoppelten Systems nach dem Hardware-in-the-Loop-Prinzip (nach Prof. Marc Ihle, Hochschule Karlsruhe, RPES-09 Folie 17)

Bei *Airbus* in Hamburg-Finkenwerder wird das Kabinen-Management-System getestet; während die Umwelt (also die umgebenden Systeme) in *CANoe* simuliert werden soll (vgl. Abbildung 5). Modbus-TCP soll einen Teil der Kommunikation der simulierten Umwelt mit dem realisierten System wahrnehmen (in Abbildung 5 die orangenen Pfeile).

„Zur einfachen und automatisierten Durchführung von Tests enthält *CANoe* das *Test Feature Set*. Damit werden sequentielle Testabläufe modelliert, ausgeführt und automatisch ein Testreport erzeugt.“ [8] Dieses Feature Set kann von *Airbus* zukünftig zur vollständigen Automatisierung der Testumgebung verwendet werden.

„*CANoe* unterstützt die folgenden Bussysteme: CAN, LIN, MOST, FlexRay, J1708, Ethernet, WLAN und AFDX®.“[9] Aufgrund der Vielzahl der vorhandenen Ethernet-basierten Protokolle und der Verschiedenheit ihrer Einsatzzwecke wurde jedoch auf die Implementierung höherer Protokolle verzichtet. Stattdessen stellt *CANoe* Möglichkeiten zum Versenden „roher“ Ethernet-Pakete sowie zum Behandeln von TCP- und UDP-Verbindungen zur Verfügung.

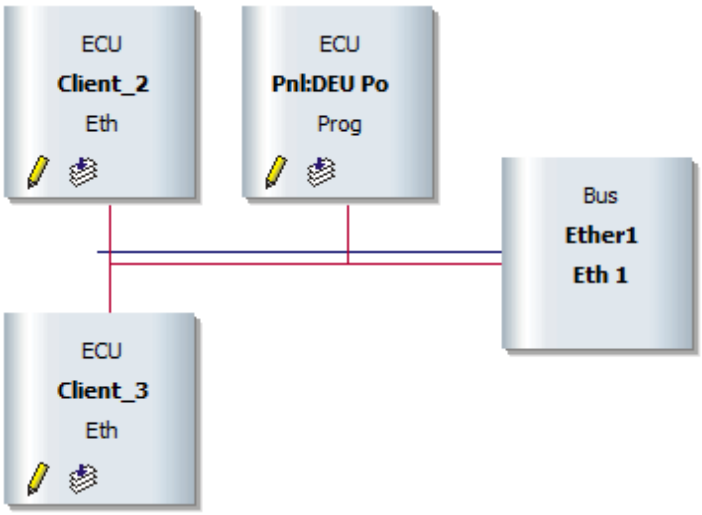


Abbildung 6: Beispielhafter Simulationsaufbau mit drei Netzwerkknoten an einem Ethernet-Bus

[8] Produktinformation *CANoe* (VecPI), S. 4
[9] Produktinformation *CANoe* (VecPI), S. 5

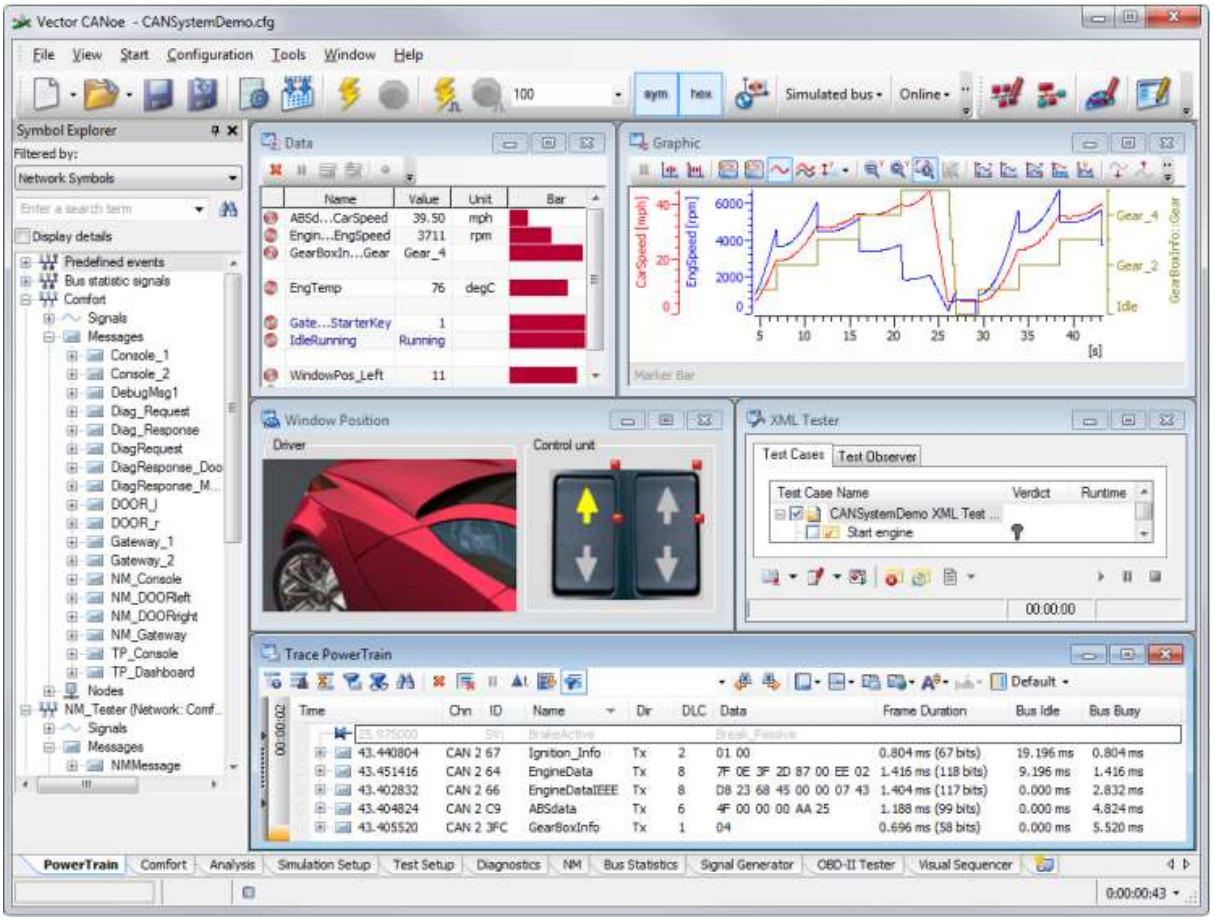


Abbildung 7: Bedienoberfläche von *CANoe* während eines simulierten Tests eines Fensters und der dazugehörigen Fensterheber. Alle Rechte bei *Vector Informatik GmbH*

2.4.1 Programmiersprache CAPL

„Mit der Programmiersprache CAPL (Communication Access Programming Language) ist der Funktionsumfang von *CANoe* stark erweiterbar. […] Im Gegensatz zu C [welches sequenziell ausgeführt wird, Anm. des Verfassers] stehen in CAPL spezielle vordefinierte Event-Handler (Ereignisprozeduren) zur Verfügung, die immer dann ausgeführt werden, wenn ein bestimmtes Ereignis eintrifft – entweder zeitgesteuert oder durch die Hardware bzw. *CANoe*-intern ausgelöst.“[10]

Ein solches CAPL-Programm wird einem Netzwerkknoten in *CANoe* zugewiesen, welcher daraufhin auf Ereignisse wie z.B. eine eingehende Nachricht reagieren kann. Im Verlauf dieser Arbeit wird diese Methode verwendet, um Modbus-Telegramme zu senden und zu analysieren.

Zur Verdeutlichung des Aufbaus und der Syntax soll hier ein kleines CAPL-Programm eingefügt werden, das zum Messungsstart eine Nachricht sendet und auf die Antwort wartet:

[10] Produktinformation *CANoe* (VecPI), S. 25f.

```
includes
{
      #include "SomeMethods.cin"
}
// Globale Variablen
variables
{
      msTimer timeout;                          // Ein ms-genauer Timer

      enum e_Status {INIT, OK, ERROR}           // Der Status des Knotens
      enum e_Status Status;
}
// Wird nach Start der Simulation/Messung ausgeführt
on start
{
      message ehlo;                             // Zu sendende Nachricht
      welcome();
      Status = INIT;
      ehlo.name = "%NODE_NAME%";                // Nachricht füllen
      output(ehlo);                             // Nachricht senden
      setTimer(timeout, 20);                    // Timer starten: 20ms
}
// Gibt eine Nachricht im Trace-Window aus
void welcome()
{
      byte i;                                   // Zähler
      write("Hello!");                          // Ausgabe für den Nutzer
      for (i = 0; i < 10; i++)
            write("%d", i);
}
// Wird beim Empfangen einer Nachricht ausgeführt
on message ehlo_response
{
      cancelTimer(timeout);                     // Timer stoppen
      Status = OK;
      write("EHLO worked!");
}
// Wird nach Ablauf des Timers ausgeführt
on timer timeout
{
      Status = ERROR;
      write("EHLO timed out!");
}
```

2.4.2 Datenmodelle in *CANoe*

In *CANoe* existieren zwei Modelle mit Systemzuständen und –werten umzugehen: Sie können sowohl in „Systemvariablen“ als auch in „Signalen“ repräsentiert werden. Eine Mischung beider Formen in einem Projekt ist ebenso möglich.

Signale und Systemvariablen können in *CANoe* grafisch gezeichnet und in einem Panel (grafische Oberfläche mit Schaltern etc.) hinterlegt werden. Auf Aktualisierungen und Änderungen kann in einem CAPL-Programm reagiert werden, z.B. um dem Netzwerk die Änderung mitzuteilen.

Darüber hinaus können Systemvariablen und Signale auf andere Variablen und Signale – auch untereinander – abgebildet werden („Mapping“). Eine Änderung wird dann sofort auf den definierten Partner übertragen. Dies ist insbesondere praktisch, wenn mehrere Namen und Ordnungen sinnvoll sind, wie z.B. ein übertragungs- (Klemme 7: Pins 4, 5, 6) und ein objektorientierter (Lichtschalter Sitz 6B, 6C, 7A) Ansatz. Gleichzeitig kann eine lineare Umrechnung vorgenommen werden.

1. Signale
   sind das Mittel der Wahl bei Verwendung eines dezentralen Bussystems wie z.B. CAN. Sie werden typischerweise direkt durch ankommende Nachrichten gesetzt. Hierfür werden in einer Projekt-Datenbank Signale und Nachrichten definiert. Anschließend wird festgelegt, welcher Netzwerkknoten über welche Signale verfügt und in welchen Nachrichten und an welcher Position (bitweiser Offset) die Werte der Signale übertragen werden. Anschließend aktualisiert *CANoe* jedes Mal, wenn eine Nachricht empfangen wird, die Werte der darin enthaltenen Signale. Somit wird die Payload eingehender Nachrichten automatisch in die Bestandteile zerlegt, eventuell konvertiert und ins lokale Prozessabbild übertragen.
   Damit dies funktioniert ist es jedoch notwendig, dass eine Nachricht eindeutig identifiziert werden kann. Beim CAN-Bus wird hierfür der Objekt-Identifier verwendet, anhand dessen somit sowohl die Quelle als auch die Nachrichtenstruktur erkannt werden kann. Um Ethernet-Pakete ebenfalls als Nachrichten verwenden zu können, muss der Ethernet-Interaction-Layer (s. Kapitel 3.1.1) für *CANoe* die Identifizierung übernehmen.
2. Systemvariablen
   sind hingegen von Nachrichten zunächst unabhängig und global les- und schreibbar. Die Möglichkeit, sie in Namespaces zu gliedern, erhöht die Übersichtlichkeit bei einer größeren Anzahl von Variablen erheblich.
   Wenn Systemvariablen die Werte der auf dem Bus gesendeten Nachrichten enthalten sollen, müssen diese Nachrichten in (CAPL-)Programmen verarbeitet und der Wert der entsprechenden Variable zugewiesen werden. Der Implementierungs-Aufwand ist hier generell höher, ermöglicht jedoch auch eine größere Flexibilität, wenn aufgrund des verwendeten Protokolls z.B. keine eindeutige Nachricht identifiziert werden kann oder die Positionen der Signale in der Nachricht nicht festgelegt sind.
   Darüber hinaus sind Systemvariablen sehr gut geeignet, Konfigurationsdaten zu speichern, die dann in CAPL-Programmen ausgelesen werden. Da der Zugriff auf die Variablen in Abhängigkeit vom Namen des Netzwerkknotens und weiterer Parameter erfolgen kann, kann hierdurch ein CAPL-Programm mehrfach mit verschiedenen Konfigurationen betrieben werden. Dies kann als Annäherung an Klasseninstanzen aufgefasst werden.

3 Entwurf und Analyse möglicher Lösungen

Grundsätzlich ist es sinnvoll, einen allgemeinen Modbus-Stack nach Abbildung 8 in *CANoe* zu implementieren und die Busklemmen- und Airbus-spezifischen Aktionen in eine Applikations-Schicht zu verschieben. Dadurch wird der Code übersichtlicher und kann leicht in weiteren Projekten verwendet werden.

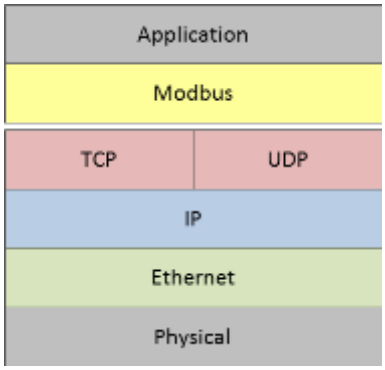


Abbildung 8: Abbildung des Modbus-Stacks, wie er in *CANoe* integriert werden soll

Für die Aufgabenstellung sind aufgrund der Beschaffenheit von *CANoe* (s. Kapitel 2.4.2) mehrere Lösungsansätze möglich, die im Folgenden entworfen und auf Vor- und Nachteile analysiert werden sollen. Außerdem wird auf generelle Probleme einer Modbus-Implementierung eingegangen und deren Lösung entworfen sowie die Applikations-Schicht konzipiert.

3.1 Verwendung des Ethernet-Interaction-Layers und von Signalen

Bei diesem Lösungsansatz soll der Ethernet-Interaction-Layer von *CANoe* verwendet werden. Der Interaction-Layer analysiert – wie in Kapitel 2.4.2 erwähnt – die empfangenen Nachrichten und teilt *CANoe* sowohl den Nachrichten-Identifier als auch den Offset der Payload mit. Die Nachrichten werden daraufhin, wie in der Projekt-Datenbank definiert, von *CANoe* zerlegt und die Werte der beinhalteten Signale aktualisiert.

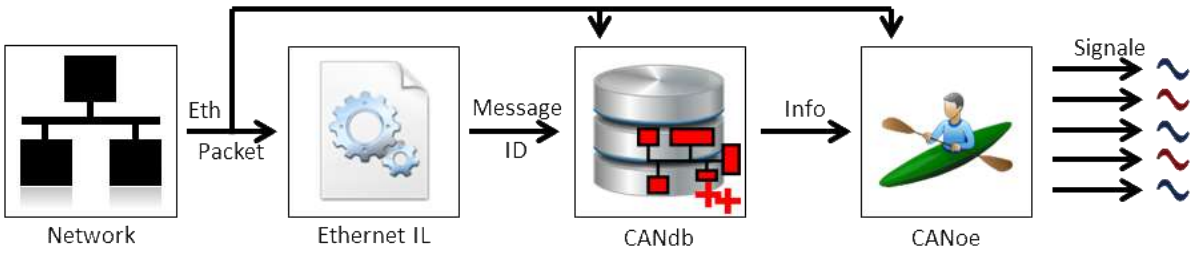


Abbildung 9: Verarbeitung eines eingehenden Ethernet-Pakets mit dem Ethernet-Interaction-Layer

Nun soll für diese Vorgehensweise ein Lösungsweg skizziert, validiert und auf eventuell vorhandene Probleme untersucht werden.

3.1.1 Der Ethernet-Interaction-Layer (EIL)

Der EIL ist eine C++-Bibliothek, welche von *CANoe* verwendet wird. Jedes Ethernet-Paket wird auf bekannte Protokolle hin Stack-aufwärts analysiert. *CANoe* bietet die Möglichkeit, zum Messungsstart eine zusätzliche DLL zu laden, welche dem EIL weitere Protokolle hinzufügt.

In diesem Fall kann so das Protokoll „Modbus“ als auf UDP/IP aufbauend anhand der Port-Nummern registriert werden (eine TCP-Verbindung ist nicht möglich, da der EIL verbindungslos arbeitet). Der EIL allokiert daraufhin den benötigten Speicherplatz für ein UDP-Paket. Ebenso registriert die DLL bei

CANoe, welche Header in einem Modbus-Paket zur Verfügung stehen und stellt Informationen über Offset, Größe und Inhalt zur Verfügung. Somit können mit wenigen einheitlichen Funktionen alle Felder des Pakets protokollweise gesetzt werden, z.B.:

```
this->SetTokenData      (eth,    kEthDestination,   6, broadcastMacId);  // broadcast
this->SetTokenUnsigned  (ipv4,   kIPv4Destination,  4, 0xFFFFFFFF);      // broadcast
this->SetTokenUnsigned  (udp,    kUDPDestination,   2, kServerPort);     // port: 502
this->SetTokenUnsigned  (modbus, mTxId,             2, 0x0123);          // some TxID
```

Dieser Zugriff ist dann auch in CAPL möglich:

```
EthSetTokenData  (Packet, "eth",     "destination", 6,   RemoteMac);     // server MAC
EthSetTokenInt   (Packet, "ipv4",    "destination",      RemoteIP);      // server IP
EthSetTokenInt   (Packet, "udp",     "destination",      502);           // server Port
EthSetTokenData  (Packet, "modbus",  "data",  length,    data);          // our data
```

3.1.2 Nachrichten-Identifizierung

Viel wichtiger als die Zugriffsmöglichkeiten auf das Modbus-Telegramm ist jedoch, dass der EIL die Nachricht für *CANoe* identifiziert. Für diese ID stehen insgesamt vier Byte zur Verfügung, welche eindeutig vergeben werden müssen.

Um die Nachricht einer Quelle zuordnen zu können, muss eine Adresse des Senders Teil der Nachrichten-ID sein. Hier bietet sich die IP-Adresse (Länge: 4 Byte) an. Da die Struktur einer Modbus-Nachricht vom Funktionscode bestimmt wird und darauf in der Projekt-Datenbank Bezug genommen werden muss, muss dieser ebenfalls in die ID einfließen (Länge: 1 Byte).

Die Struktur der Modbus-Pakete ist jedoch auch pro Funktionscode immer variabel: Es ist bei Modbus ohne weiteres möglich, die ersten 20 Register ab Adresse `0x00` zu lesen und beim nächsten Mal nur noch 19 Register ab Adresse `0x01` abzufragen. In diesem Fall fehlt also im Vergleich zum ersten ein Signal und die anderen sind um jeweils 16 Bit verschoben. Um die volle Flexibilität von Modbus zu erhalten, müssten somit die Start-Adresse (Länge: 2 Byte) und die Anzahl der Elemente (Länge: 2 Byte) mit aufgenommen werden. Da beide Felder jedoch in einer Antwort gar nicht gesendet werden, müssen sie beim Senden der Anfrage vom EIL gespeichert werden, um beim Empfang der Antwort für die Nachrichten-ID verwendet werden zu können.

Insgesamt müssten somit neun Byte lange Identifier verwendet werden. Um diese auf vier Byte zu reduzieren und die Größe der Projektdatenbank in einem annehmbaren Rahmen zu halten, sind gewisse Kompromisse und Einschränkungen nötig:

- Die IP-Adresse ist zwingend nötig. Das IP-Netz kann jedoch auf eine Größe von ein oder zwei Byte eingeschränkt werden (`/24` bzw. `/16`), wodurch immer noch 253 bzw. 65.533 Modbus-Server möglich sind.
- Der Funktionscode enthält große Redundanzen. Bei allen Funktionen (außer `0x2B`) werden insgesamt nur sechs Bit benötigt. Mit eingeschlossen sind hierbei die Exceptions, die bei jedem Funktionscode auftreten können. Insgesamt wurden jedoch nur zehn Funktionen implementiert, weshalb man durch eine Kodierung ein weiteres Bit sparen könnte. Eine Reduzierung um zwei oder drei Bit stellt jedoch keinen großen Gewinn dar und ist (bis auf weiteres) nicht nötig.

- Start-Adresse und Anzahl der Elemente können nicht reduziert und müssen deshalb zwangsläufig außen vor gelassen werden. Das führt dazu, dass pro Funktionscode nur noch eine Adresse und Elementzahl zugelassen wird. Dies ist bei dieser Aufgabe im zyklischen Betrieb durchaus machbar, schränkt jedoch die Flexibilität des Modbus-Protokolls stark ein. So führt dies unter anderem dazu, dass von einer Busklemme nicht mehr Bits bzw. Register abgefragt (und geschrieben) werden können, als in ein Telegramm passen (2.000 bzw. 125). Bei *B+R*-Geräten verringert dies die verfügbare Kapazität erheblich, da hier bis zu 16.384 digitale und 2.048 komplexe Eingänge möglich sind. Bei *Wago* wird die Kapazität halbiert.
  Um im azyklischen Dienst (z.B. bei der Konfiguration der Geräte) besondere Registerwerte abfragen zu können (vgl. Kapitel 2.3.2), wäre es auch denkbar, ein weiteres Feld einzuführen, das von der Adresse abgeleitet wird. Hierfür ist es jedoch notwendig, dass die herstellerspezifischen Start-Adressen dieser Felder sich nicht überschneiden und fest in die DLL einprogrammiert werden.

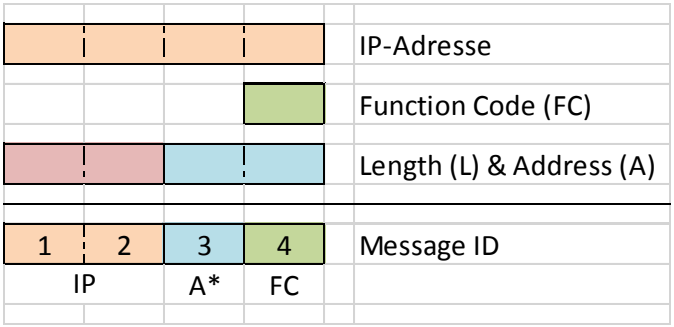


Abbildung 10: Geplante Belegung der vier Bytes der Message-ID

3.1.3 Probleme und Problembewertung

Das Verwenden des Ethernet-Interaction-Layers und der Signale stellt eine gute Möglichkeit dar, die bestehenden Fähigkeiten zur Nachrichtenanalyse in *CANoe* zu nutzen. Vor allem bei der zyklischen Verarbeitung mit festem Inhalt pro Funktionscode zeigt dieser Ansatz seine Stärken.

Die hohe Dynamik im Inhalt der Modbus-Pakete (mit Start-Adresse und Elementanzahl) führt jedoch vor allem im Hinblick auf azyklische Anfragen zu Problemen. Hierfür ist der EIL einfach nicht konzipiert. Ebenso ist der EIL derzeit nur unzureichend dokumentiert, was eine Implementierung erschwert.

3.2 Implementierung eines Modbus-Stacks in CAPL

Bei diesem Lösungsansatz werden die Modbus-Pakete vollständig in der Programmsprache CAPL zusammengestellt, versendet, empfangen und verarbeitet. Zur Verbesserung der Übersichtlichkeit und der Austauschbarkeit von TCP und UDP ist es ratsam, eine Schicht zu implementieren, welche die Netzwerkschicht abstrahiert. Die empfangenen Zustände der Ein- und Ausgänge werden in Systemvariablen gespeichert (vgl. Kapitel 2.4.2).

CAPL ist eine recht einfache und zugängliche Sprache und die Verwendung von Systemvariablen erleichtert die Abstraktion von der Aufgabenstellung. Es ist damit möglich, einen Modbus-Client zu implementieren, der anhand seines Netzwerkknotennamens in den Systemvariablen seine eigenen Parameter abrufen kann. Somit können viele Clients mit derselben Code-Basis betrieben werden.

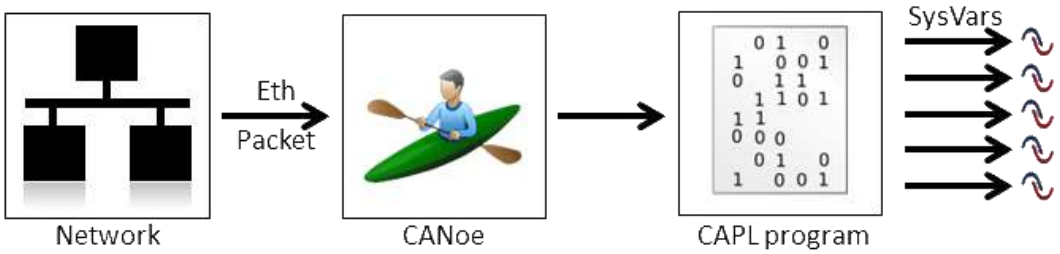


Abbildung 11: Verarbeitung eines eingehenden Ethernet-Pakets in einem CAPL-Programm

3.2.1 Senden der Netzwerkpakete

Bei der Implementierung des Modbus-Stacks stellt *CANoe* mehrere Möglichkeiten zur Verfügung, mit dem Ethernet-Netzwerk zu kommunizieren:

1. Verwendung des CAPL-TCP-Stacks
   Hierbei wird fast direkt mit dem TCP- bzw. UDP-Stack des Betriebssystems[11] gesprochen. Es müssen Sockets geöffnet und geschlossen werden und jedes ankommende Paket wird an eine spezielle Event-Funktion gemeldet. Der Stack weist zum Zeitpunkt dieser Arbeit noch verschiedene Fehler auf, welche die Arbeit etwas erschweren. Davon abgesehen kann bei dieser Methode jedoch von allen Vorteilen profitiert werden, die ein Betriebssystem-Stack bietet, z.B. eine hohe Zuverlässigkeit durch die häufige Verwendung in verschiedenen Applikationen sowie eine hohe Geschwindigkeit durch die systemnahe Integration.

2. Verwendung des Ethernet-Interaction-Layers ohne Modbus-Erweiterung
   Der EIL ermöglicht das Senden und Empfangen von Ethernet-Paketen (vgl. Kapitel 3.1.1). Dabei können über einfache Befehle die Felder für Protokolle wie IP und UDP gesetzt werden können, jedoch sind jegliche Kontrollen (wie z.B. Prüfsummen) außer Kraft gesetzt. Somit können auch fehlerhafte Pakete gesendet werden (was sich besonders für Tests des Stacks der Gegenseite anbietet). Der Layer ist verbindungslos, weshalb entweder ein eigener TCP-Stack implementiert werden müsste oder Modbus auf UDP begrenzt bliebe.
   Letzteres wurde probeweise implementiert. Eine Schwierigkeit dabei ist, dass auch die Ethernet-Adressen gesetzt werden sollten, um dem Switch die Arbeit zu erleichtern. Hierfür ist es nötig, ein korrektes ARP-Paket zu senden und die Antwort hierauf zu empfangen. Schlussendlich funktioniert auch diese Variante sehr gut, allerdings muss der Nutzer die lokale IP-Adresse angeben, damit diese in den IP-Header eingetragen werden kann. Eine Möglichkeit diese in *CANoe* auszulesen besteht derzeit noch nicht.

3. Verwendung des Ethernet-Interaction-Layers mit Modbus-Erweiterung.
   Wie im vorherigen Lösungsansatz (s. Kapitel 3.1) kann der EIL um das Modbus-Protokoll erweitert werden. Es sollen nun jedoch nicht die Werte automatisch in Signale übersetzt, sondern lediglich auf die Paket-Struktur zugegriffen werden, womit die Nachrichten-ID-Problematik entfällt (vgl. Kapitel 3.1.2). Die Modbus-Header werden somit (aus CAPL-Sicht) nicht als Payload in ein UDP-Paket geschrieben, sondern konkret mithilfe der EIL-Funktionen gesetzt. Lediglich die Modbus-Payload wird dann in das Daten-Feld geschrieben.
   Dieser Ansatz vereinfacht die Arbeit mit Modbus-Paketen ein wenig, birgt sonst jedoch keine großen Vorteile. Beim Lesen der Daten hingegen muss expliziert darauf geachtet werden,

[11] Es besteht auch die Möglichkeit, den TCP-Stack des Betriebssystems zu umgehen und stattdessen auf einen von *CANoe* angebotenen Stack zurückzugreifen. Dies ermöglicht unter Anderem die freie Wahl der IP-Adresse.

dass die Byte-Reihenfolge geändert wird, da dies von der entsprechenden Funktion nicht getan wird.

Da der EIL wie oben beschrieben keine großen Vorteile bietet, wurde der CAPL-TCP-Stack verwendet. Lediglich die Erkennung, ob das Netzwerk verfügbar ist, stammt aus dem Funktionsraum des Ethernet-Interaction-Layers.

3.2.2 Erstellung und Analyse der Modbus-Telegramme

Beim Erstellen von Netzwerk-Stacks ist es nach Erfahrung des Autors ratsam, auf Strukturen zurückzugreifen, welche die Felder eines Pakets repräsentieren. Diese Möglichkeit – inklusive eventuell nötiger Konvertierung der Bytereihenfolge beim Kopieren in einen Puffer – bietet CAPL. Somit kann für jeden Funktionscode jeweils eine Anforderungs- und eine Antwort-Struktur verwendet werden.

Darüber hinaus müssen die Dienstzugangspunkte für den Nutzer definiert werden. Es bietet sich an, für Modbus-Anfragen pro Funktionscode eine eigene Funktion zu definieren. Diese muss die Meta-Daten (Adresse, Elementzahl) sowie eventuell die Daten (Werte für Bits oder Register etc.) entgegen nehmen, die passende Struktur füllen und das Telegramm absenden.

Bei Empfang einer Antwort wird das Telegramm vom Puffer in die zum Funktionscode passende Struktur kopiert und überprüft, ob es sich um ein korrektes Modbus-Telegramm oder ob der Server einen Fehler zurückgegeben hat (Modbus-Exception). Anschließend kann anhand des Funktionscodes eine Notifikation an den Nutzer erfolgen, eventuell mit Übergabe der empfangenen Daten.

3.3 Lösung ansatzübergreifender Probleme

Unabhängig vom gewählten Lösungsansatz ergeben sich Probleme, die im Folgenden erläutert und deren Lösung skizziert werden soll.

3.3.1 Erkennung von Verbindungsabbrüchen

Bei einem automatisierten Test müssen etwaige Verbindungsprobleme erkannt werden, z.B. um das abgerufene Prozessabbild zu invalidieren und den Test zu unterbrechen, bis die Verbindung wieder hergestellt ist. Andernfalls wäre das Testergebnis nicht verlässlich. Wird Modbus-TCP über TCP verwendet, existieren bereits effektive Mechanismen, um die Verbindung zu überwachen und notfalls Pakete erneut zu senden. Trotzdem müssen abbrechende TCP-Verbindungen behandelt und gemeldet werden. Noch wichtiger ist dies bei Modbus über UDP, da hier keine Verbindung im eigentlichen Sinne aufgebaut wird. Bei der Aufgabenstellung wurde jedoch Wert auf ein physikalisch verlässliches und explizit für Modbus verwendetes Netz gelegt. Das bedeutet, dass Verbindungsprobleme eigentlich nur auftreten können, wenn das betroffene Gerät vom Strom- oder Ethernet-Netz getrennt wurde, da die Netzlast nahezu konstant und ausreichend gering ist, um Kollisionen und Pufferüberläufe zu vermeiden. In solch einem kleinen und verlässlichen Netz ist UDP bei der Erkennung von Timeouts von Vorteil: Da TCP mehrere Versuche unternimmt, das Gegenüber zu kontaktieren, wird ein Abbruch der Verbindung von der Applikation deutlich später erkannt als bei einer UDP-Verbindung. Somit kann mit letzterem bei Tests eine geringere Reaktionszeit eingestellt werden, die Wahrscheinlichkeit für verlorene Telegramme wird jedoch gleichzeitig erhöht.

Zumindest bei UDP ist es demzufolge nötig, Timer zu implementieren, um die Verbindung zu überwachen. Da sich CAPL an C orientiert und Klassen nur in sehr begrenztem Maße möglich sind, kann nicht für jedes Paket eine neue Instanz eines Timers erstellt werden. Darüber hinaus muss ein

Timer einen Namen besitzen, um eine Prozedur definieren zu können, die nach Ablauf ausgeführt wird; er kann also auch nicht Teil eines assoziativen Felds[12] sein. Die übrigen Möglichkeiten sollen nun dargestellt werden. Das Timer-Intervall wird als t_t, die Latenz als l, die Verarbeitungsdauer als $t_p = 3ms$ und die Sendefenstergröße als w_s bezeichnet.

1. Verwendung eines Timers ($t_t = 2l + t_p$), Ablehnen neuer Pakete: $w_s = 1$
   Wird nur ein Timer zur Überwachung aller gesendeten Pakete eingesetzt, muss beim Sendeversuch überprüft werden, ob bereits auf eine Bestätigung eines Pakets gewartet wird. Ist dies der Fall, wird der Sendeversuch abgebrochen.
   Dieser Aufbau entspricht einer Sendefenstergröße von 1. Obwohl dies in manchen Fällen bereits ausreicht, treten doch gewisse Schwierigkeiten auf: Wenn ein Paket gesendet werden soll, obwohl noch ein anderes unterwegs ist, muss dieses abgelehnt werden. Somit muss eine Fehlerbehandlung auf der höheren Schicht, die den Sendeversuch unternimmt, eingeführt werden, was nicht erwünscht ist. Darüber hinaus ist absehbar, dass sowohl Register- als auch Bit-Werte gleichzeitig abgefragt werden sollen.

2. Verwendung eines Timers pro Funktionscode ($t_t = 2l + t_p$), Ablehnen neuer Pakete: $w_s \leq 10$
   Eine zweite Möglichkeit ist die Verwendung mehrerer Timer, wobei jeder einem speziellen Modbus-Funktionscode zugeordnet ist. Die Senderoutine startet den jeweiligen Timer und dessen Prozedur meldet wiederum den Fehler des Requests. Somit können u.a. Register und Bits gleichzeitig gelesen und geschrieben werden, jedoch sind zwei gleichzeitige Register-Abfragen nicht möglich.
   Diese Methode erwies sich ebenfalls als unpraktisch: Einerseits wurde obiges Problem mit der Fehlerbehandlung in der Applikationsschicht nicht gelöst, anderseits soll bei Analyse der Busklemme bestimmte Parameter und somit mehrere Register abgefragt werden (s. Kapitel 3.3.2). Um dies zu implementieren, müsste eine Zustandsmaschine erstellt werden.

3. Verwendung eines Timers ($t_t = 2l + t_p$), Schreiben neuer Pakete in einen Stack: $w_s \leq 2000$
   Sollen Sendevorgänge nicht mehr abgelehnt werden, ist es nötig, einen Stack zu implementieren, in den diese Pakete gespeichert werden können. Glücklicherweise unterstützt CAPL assoziative Arrays, als Index wird die fortlaufende Transaktionsnummer des Modbus-Protokolls eingeführt. Beim Senden wird das Paket zuerst in einen Stack geschrieben. Dieser wird von einem Timer überwacht, der die Pakete sendet und überprüft, ob die bereits gesendeten Pakete beantwortet wurden. Ist dies der Fall, wird das Paket aus dem Stack entfernt.
   Um ein einheitliches Timeout-Intervall zu gewährleisten, müssen die Anfragen in Bursts gesendet werden. Hierdurch verdoppelt sich die maximal mögliche Antwortzeit: Besitzt das Netzwerk eine sehr hohe Latenz von z.B. 20 ms, muss der Timer entsprechend langsam (hier mit 50 ms) getaktet werden. Ein Paket, das nicht beantwortet werden wird, wird im schlechtesten Fall erst nach 50 ms Wartezeit im Stack gesendet und benötigt dann nochmals 50 ms, um als verloren erkannt zu werden.

4. Verwendung eines Timers ($t_t = 1ms$), Schreiben neuer Pakete in einen Stack: $w_s \leq 2000$
   Die Lösung der großen Latenz bietet ein Timer mit dem kleinstmöglichen Intervall von 1 ms. Die neuen Pakete werden hierbei wie bei Punkt 3 in einen Stack geschrieben, jedoch überprüft der Timer jede Millisekunde, ob bereits Pakete beantwortet wurden. Ist dies nicht der Fall,

[12] Assoziatives Feld: Array mit dynamischer Größe und ohne fortlaufenden Index, vergleichbar mit Hashtables

wird für jedes Paket ein Zähler inkrementiert, der die verstrichene Zeit in Millisekunden seit dem Senden darstellt. Jedes Paket erhält somit quasi seinen eigenen Timer.
Mit diesem Ansatz werden neue Pakete im Stack nach spätestens 1 ms gesendet, das Timeout bleibt jedoch variabel. Erkauft wird dieser Gewinn mit einer höheren ($\approx 2 \cdot l$-fachen) Rechenbelastung. Diese kann jedoch reduziert werden, wenn der Timer nur gestartet wird, wenn unbeantwortete Anfragen vorhanden sind, außerdem kann auch ein etwas größeres Intervall (z.B. 2 ms) verwendet werden.

Da das Empfangsfenster der *Wago 750-881* nur fünf und das der *B+R X20BC0087* nur acht Pakete fasst – also nur diese Anzahl an Telegrammen gleichzeitig verarbeitet werden kann – wird das Sendefenster ebenfalls auf diesen Wert begrenzt.

3.3.2 Generieren der Konfiguration

Damit mit dem Prozessabbild in *CANoe* gearbeitet werden kann, muss das Abbild vor dem Messungsstart definiert werden. Um dem Nutzer langwierige Arbeit abzunehmen, sollen die Definitionen generiert werden. Da hierbei spezifische Informationen zu der jeweiligen Busklemme mit einfließen müssen, bietet es sich an, die Busklemmen bei der Generierung der Systemvariablendefinitionen hinsichtlich der angeschlossenen Module zu analysieren (s. Kapitel 3.3.3). Für diese Aufgabe kann auf den implementierten Modbus-Client zurückgegriffen werden.

Bei der Verwendung von Systemvariablen (s. Kapitel 3.2) müssen diese zur Wiederverwendbarkeit des Modbus-Codes in Namensräume nach bestimmten Konventionen sortiert werden (ungefähr `%BUS%::%KLEMME%::Inhalt`). Darüber hinaus ist es nötig, vor der Laufzeit die Größe der Systemvariablen-Arrays zu definieren, welche die Daten der Register und Bits beinhalten. Die endgültige XML-Datei kann dann in *CANoe* eingebunden werden.

Identisch verhält es sich bei der Verwendung des Ethernet-Interaction-Layers mit Signalen (s. Kapitel 3.1). Hier können anhand der von den Busklemmen abgefragten Daten die Definitionen der Nachrichten (mit Message-ID) und der darin enthaltenen Signale generiert werden.

Die nötigen Definitionen können als ASCII in Dateien geschrieben und als solche in *CANoe* eingebunden werden. Das nötige Format kann leicht in den vorhandenen Dateien nachvollzogen werden.

3.3.3 Analyse der Busklemmen

Die Busklemmen der beiden Hersteller stellen – wie in Kapitel 2.3.2 beschrieben – Informationen über sich selbst und die angeschlossenen Module bereit. Dazu gehören Informationen, wie viele komplexe und digitale Ein- und Ausgänge verfügbar sind. Auf diese Informationen wird während der Generierung der Konfiguration für *CANoe* (s. Kapitel 3.3.2) zugegriffen.

Da diese Angaben abhängig vom Hersteller an unterschiedlichen Adressen zur Verfügung gestellt werden, muss diesbezüglich zuerst eine Unterscheidung erfolgen. Die Modbus-Spezifikation definiert zwar den Funktionscode `0x2B` (Subcode `0x0E`) zur Abfrage des Herstellers[13], jedoch wurde dieser von keinem der beiden Hersteller implementiert. Eine zweite Möglichkeit stellen die Konstanten dar, welche *Wago*-Busklemmen bei Abfrage bestimmter Register-Adressen zurückgeben, allerdings sind

[13] Modbus Application Protocol (MAPP), Kapitel 6.21

die Werte an diesen Adressen bei Busklemmen von *B+R* dynamisch und könnten somit ebenfalls den Wert der Konstante annehmen, was eine Fehlentscheidung zur Folge hätte.

Schließlich bleibt nur noch, eine Abfrage zu definieren, welche von der Busklemme eines Herstellers normal beantwortet werden kann, aber bei der anderen Busklemme fehlschlägt und mit einer Modbus-Exception beantwortet wird. In dieser Arbeit wird die MAC-Adresse der *B+R*-Klemme abgefragt (drei Wörter ab Adresse `0x1000`). Diese Abfrage ist bei *Wago*-Klemmen nicht möglich, da diese an `0x1000` die „WatchdogTime" (ein Wort) zur Verfügung stellen und für diese Adresse deshalb nur die Abfrage eines einzelnen Wortes ermöglicht wird.

Anschließend kann die Anzahl der digitalen und komplexen Ein- und Ausgänge abgefragt werden. Eine Einschränkung existiert jedoch bei den Busklemmen von *B+R*: Im Prozessabbild belegt ein Modul immer eine ganze Zahl an kompletten Bytes. Wenn ein Modul nun beispielsweise über zwölf digitale Eingänge verfügt, werden im Prozessabbild vier Bits angefügt, um 16 Bits und somit zwei Bytes belegen zu können. Dies muss später beim Mapping der Variablen (vgl. Kapitel 2.4.2) beachtet werden, da das Programm über keine weiteren Informationen zu den Modulen verfügt und somit nicht wissen kann, dass es tatsächlich nur zwölf und nicht 16 digitale Eingänge sind. Darüber hinaus ist zu beachten, dass das *X20BC0087* die ersten drei komplexen Worte im Prozessabbild selbst belegt. Den Inhalt dieser Register konnte der Support von *B+R* auf Anfrage nicht erklären.

Beide Busklemmen ermöglichen die Abfrage der Produkt-Codes der angeschlossenen Module. Nur *Wago* war jedoch in der Lage, eine Zuordnung von Produkt-Codes zu Art und Größe des Moduls (z.B. DI16, AO2) zur Verfügung zu stellen. Deshalb können dem Nutzer bei Busklemmen von *B+R* keine detaillierten Informationen über die angeschlossenen Module zur Verfügung gestellt werden. Dies ist jedoch nur ein kosmetisches Problem.

3.4 Der zyklisch abfragende Modbus-Client

Nachdem der Modbus-Stack in *CANoe* implementiert wurde, muss eine darauf aufbauende Schicht eingefügt werden, welche die Brücke von den Modbus-Nachrichten zu *CANoe* schlägt.

Ganz gleich, welche der oben aufgeführten Implementierungen vorgenommen wird, müssen die Zustandsdaten aufgrund des Master-Slave-Prinzips von Modbus zyklisch von der Busklemme abgefragt werden. Hierfür genügt es, einen Timer einzurichten, der immer wieder über den Modbus-Stack diese Daten von der Busklemme anfordert.

Es ist zu beachten, dass der in komplexen Busklemmen-Modulen verwendete Analog-Digital-Wandler je nach Ausfertigung bis zu 80 ms benötigen kann, bis ein analoger Zustand in einen digitalen Wert gewandelt wurde und zur Abfrage zur Verfügung steht. Ein geringeres Aktualisierungsintervall ist somit nicht sinnvoll. Für eine genaue Auskunft soll hier auf das jeweilige Datenblatt des Moduls verwiesen werden. Die digitalen Eingänge von *Wago* verfügen gemäß Auskunft über Filter, welche prellende Schalter neutralisieren sollen. Eine Änderung steht hier nach 3 ms zur Verfügung; bei den verwendeten *B+R*-Modulen *X20DI9372* kann dieser Wert zwischen 100µs und 25 ms angepasst werden. Digitale Werte können somit vom Modbus-Client deutlich häufiger aktualisiert werden.

Auf der Empfangsseite des Modbus-Clients werden die Unterschiede größer: Wird der Ethernet-Interaction-Layer mit Signalen verwendet, muss sich diese Schicht nicht um eingehende Pakete kümmern; die Signale setzt *CANoe* selbstständig (s. Kapitel 3.1). Bei einer reinen CAPL-Implementierung dagegen müssen eingehende Telegramme analysiert und die enthaltenen Werte

den richtigen Systemvariablen zugeordnet werden (s. Kapitel 3.2.2). Ein Vorteil hier ist, dass beim Übertragen der empfangenen Daten in die Systemvariablen auf die Start-Adresse und die Anzahl der empfangenen Daten dynamisch reagiert werden kann (vgl. Kapitel 3.1.3). So kann z.B. ein Ausschnitt der Daten häufiger aktualisiert werden als der Rest, indem er zeitversetzt mit gleichem Intervall abgefragt wird.

Darüber hinaus ist es ratsam, einen Indikator zu implementieren, welcher auf veraltete oder fehlerhafte Daten hinweist. Tritt im Modbus-Stack ein Fehler auf (Paket kann nicht gesendet werden, Timeout, Antwort ist eine Exception), wird diese an den Modbus-Client weiter gegeben, welcher daraufhin die Werte in den Systemvariablen invalidiert; dies wird durch den Wert `-1` repräsentiert.

4 Implementierung des Modbus-Stacks

In diesem Abschnitt soll auf Details der Implementierung sowie den angestrebten Workflow eingegangen werden.

4.1 Beschreibung des Codes und der Dienstzugangspunkte

Bei der Betrachtung der Code-Struktur soll bei der niedersten Programm-Schicht begonnen und darauf aufbauend weitere Schichten betrachtet werden. Dies soll vor Allem den Einstieg in den Programmcode erleichtern. Um dem Nutzer des Stacks die Arbeit zu vereinfachen, beginnen im Programmcode alle „privaten" Funktionen, die er nicht direkt verwenden soll, mit einem Unterstrich.

Um fatale Fehler nicht durch sämtliche Schichten weiterleiten zu müssen, wurde die Funktion `OnModbusPanics(enum FatalErrors error)` definiert, an welche diese Fehler direkt gemeldet werden können. In vielen Fällen ist dann ein Abbruch der Messung ratsam.

4.1.1 Netzwerkschicht: TCP & UDP

Wie bereits in Kapitel 3.2.1 erläutert, kann Modbus sowohl über TCP als auch über UDP kommunizieren. Letzteres kann dabei wiederum mit Sockets oder über den Ethernet-Interaction-Layer implementiert werden. Um die Vor- und Nachteile der verschiedenen Protokolle ausloten zu können, wurden kurzerhand alle drei implementiert. Sie befinden sich in den CAPL-Include-Dateien `ModbusEil.cin`, `ModbusTcp.cin` und `ModbusUdp.cin`. Um einen leichten Umstieg zwischen den Protokollen zu ermöglichen, wurden Dienstzugangspunkte definiert, welche die Verbindungsmodule implementieren müssen. Es sind folgende Funktionen:

- `_ModbusConnectTo(char IP[], word Port)`
  Diese Funktion bereitet alles vor, um das Senden über die Netzwerkschicht zu ermöglichen. Dies kann das Erstellen eines Sockets, das Aufbauen einer Verbindung oder das Vorbereiten eines UDP-Pakets sein.
- `_ModbusDisconnect()`
  Diese Funktion führt Aktionen durch, welche die Netzwerkschicht ordentlich beendet.
- `_ModbusSnd(byte Buffer[], word Length)`
  Diese Funktion sendet die übergebenen Daten über das zur Verfügung gestellte Netzwerk.
- `_ModbusRecv()`
  Mit Aufruf dieser Funktion wird signalisiert, dass auf eingehende Pakete gewartet werden soll. Diese werden an dieser Stelle jedoch nicht verarbeitet.
- → `_OnModbusReceive(…, byte Buffer[], dword Length)`
  Da CAPL Event-basiert arbeitet und keine Parallelität erlaubt, würde das simple Warten auf Pakete per `_ModbusRecv()` *CANoe* komplett blockieren. Deshalb werden ankommende Pakete je nach Verbindungsart an bestimmte Funktionen gemeldet (z.B. `OnTcpReceive()` oder `OnEthReceivePacket()`). Diese Daten müssen anschließend an die Modbus-Funktion `_OnModbusReceive()` weitergegeben werden.

4.1.2 Protokollschicht: Modbus-Client

Auf die Netzwerkverbindung kann nun die Modbus-Client-Schicht aufbauen. Sie befindet sich in der CAPL-Include-Datei `ModbusClient.cin`, die Paket-Strukturen und Enumeratoren hingegen in `ModbusStructs.cin`. Eine Modbus-Schicht für den Server sähe vom Ablauf grundlegend anders aus, lediglich die Strukturen haben Client und Server gemeinsam.

Der Modbus-Client stellt als Dienstzugangspunkte jeweils Funktionen für die einzelnen Modbus-Funktionscodes bereit. Hiermit können Anfragen an den Server gestellt werden. Erfolgreiche und fehlgeschlagene Anfragen werden jeweils an eine extra Funktion gemeldet:

- `ModbusInit(char IP[], word Port, word Timeout, byte MaxTransmissions)`
  In dieser Funktion wird `_ModbusConnectTo(IP, Port)` aufgerufen und die Werte für `Timeout` und `MaxTransmissions` gespeichert. Sie werden vom Timer `gtModbusRobin` verwendet.

- `ModbusReadOutBits(word Address, long Count)`
- `ModbusReadInBits(word Address, long Count)`
  Diese Funktionen erstellen eine oder mehrere Anfragen, welche mit dem Funktionscode `0x01` bzw. `0x02` den Status der Bits an der Adresse `Address` anfordern. Dabei wird automatisch entschieden, ob die Anfrage aufgeteilt werden muss. Die empfangenen Daten bzw. ein Timeout oder eine Ausnahme werden an folgende Funktionen gemeldet:
  - `OnModbusReadBitsSuccess(struct ModbusResReceiveBits mbres, byte bitStatus[], struct ModbusReqRead mbreq)`
  - `OnModbusReadBitsFailed(enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`
- `ModbusReadOutRegisters(word Address, long Count)`
- `ModbusReadInRegisters(word Address, long Count)`
  Diese Funktionen erstellen eine oder mehrere Anfragen, welche mit dem Funktionscode `0x03` bzw. `0x04` den Status der Register an der Adresse `Address` anfordern. Die empfangenen Daten bzw. ein Timeout oder eine Ausnahme werden an folgende Funktionen gemeldet:
  - `OnModbusReadRegistersSuccess(struct ModbusResReceiveRegisters mbres, byte bitStatus[], struct ModbusReqRead mbreq)`
  - `OnModbusReadRegistersFailed(enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`
- `ModbusWriteBit(word Address, byte Value)`
  Diese Funktion erstellt eine Anfrage, welche den Wert des Bits an der Adresse `Address` setzt. Eine Erfolgsmeldung bzw. ein Timeout oder eine Ausnahme werden an folgende Funktionen gemeldet:
  - `OnModbusWriteBitSuccess(struct ModbusResConfirmSingle mbres)`
  - `OnModbusWriteBitFailed(enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`
- `ModbusWriteRegister(word Address, byte Value)`
  Diese Funktion erstellt eine Anfrage, welche den Wert des Registers an der Adresse `Address` setzt. Eine Erfolgsmeldung bzw. ein Timeout oder eine Ausnahme werden an folgende Funktionen gemeldet:
  - `OnModbusWriteBitSuccess(struct ModbusResConfirmSingle mbres)`
  - `OnModbusWriteBitFailed(enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`
- `ModbusWriteBits(word Address, long Count, byte Values[])`
- `ModbusWriteBitsB(word Address, long Count, byte Values[])`
  Diese Funktionen erstellen eine oder mehrere Anfragen, welche die Werte der Bits beginnend mit der Adresse `Address` setzen. Die zweite Funktion codiert jeweils acht Bytes aus `Values` in ein Byte, wie es der Modbus-Server erwartet. Ein Array aus Nullen und Einsen sollte also `ModbusWriteBitsB()` übergeben werden. Die Erfolgsmeldung(en) bzw. Timeout(s) oder Ausnahme(n) wird/werden an folgende Funktionen gemeldet:
  - `OnModbusWriteBitsSuccess(struct ModbusResConfirmMultiple mbres)`
  - `OnModbusWriteBitsFailed (enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`

- `ModbusWriteRegisters(word Address, long Count, word Values[])`
  Diese Funktion erstellt eine oder mehrere Anfragen, welche die Werte der Register beginnend mit der Adresse `Address` setzt. Die Erfolgsmeldung(en) bzw. Timeout(s) oder Ausnahme(n) wird/werden an folgende Funktionen gemeldet:
  - `OnModbusWriteRegistersSuccess(struct ModbusResConfirmMultiple mbres)`
  - `OnModbusWriteRegistersFailed(enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`
- `ModbusWriteMasks(word Address, word AND, word OR)`
  Diese Funktion erstellt eine Anfrage, welche den Wert des Registers mit der Adresse `Address` maskiert. Die Erfolgsmeldung(en) bzw. Timeout(s) oder Ausnahme(n) wird/werden an folgende Funktionen gemeldet:
  - `OnModbusWriteMasksSuccess(struct ModbusResConfirmMasks mbres)`
  - `OnModbusWriteMasksFailed(enum ModbusRequestError error, enum ModbusException ex, struct ModbusApHeader mbap)`

Wenn eine solche Anfrage gesendet werden soll, wird sie zuerst von der Funktion `_ModbusSend(byte buffer[], word length, word TxID)` in das assoziative Array `gQueuePending` eingereiht. Hier wird sie anhand ihrer Modbus-TransmitID referenziert. Beim nächsten Durchlauf des Timers `gtModbusRobin` (nach Punkt 4 in Kapitel 3.3.1) kann das Paket gesendet werden, falls das Sendefenster groß genug ist. Ist dies der Fall, wird es in `gQueueSent` verschoben. Der Timer prüft daraufhin jede Millisekunde, ob für dieses Paket bereits ein Timeout aufgetreten ist und meldet dies an die jeweilige `Failed()`-Funktion (s.o.). Wenn dies vom Anwender gewünscht und konfiguriert ist, kann die Anfrage auch nochmals gesendet werden; dies erwirkt jedoch normalerweise keinen Vorteil und kann sich negativ auf die Auslastung des Modbus-Servers und des Netzes auswirken.

Wird ein Paket empfangen und an `_OnModbusReceive()` weitergegeben, prüft diese und die nachfolgenden Funktionen die empfangenen Daten auf etwaige Fehler (darunter Fehler des Netzwerks-Stacks, Plausibilität der angegebenen Telegramm-Länge und Modbus-Exceptions). Darüber hinaus ist es bei der TCP/IP- und UDP/IP-API von *CANoe* möglich, dass mehrere Pakete nacheinander im Empfangspuffer landen. Deshalb müssen hieraus die Modbus-Pakete extrahiert werden, was mit Hilfe der Längenangabe aus dem Modbus-Application-Header gut möglich ist. Es ist jedoch zu beachten, dass ein Nicht-Modbus-Paket, welches somit wahrscheinlich auch keine zwei Byte große Längenangabe im dritten Wort aufweist, alle folgenden Modbus-Pakete im Puffer nicht detektierbar macht. Damit dies auftritt, muss jedoch eine Applikation im Netzwerk solche nicht interpretierbaren Telegramme an den bei jedem Programmstart zufällig gewählten Port des Modbus-Clients senden. Dem Erstellen einer solchen Applikation geht jedoch eine bösartige Absicht voraus, was in einem abgeschlossenen, firmen-internen Modbus-Netz vorerst ausgeschlossen werden kann. Somit wurde bisher auf eine effizientere Fehlerbehandlung (wie z.B. intensives Suchen nach einem vollständigen und validen Modbus-Application-Header) verzichtet.

Wurde ein Modbus-Telegramm korrekt empfangen, wird die dazugehörige Anfrage aus `gQueueSent` nach `gQueueAck` verschoben. Hierdurch ist der Timer nicht mehr für dieses Paket verantwortlich, aber die weiterführenden Funktionen können noch auf die Anfrage zurückgreifen. Dies ist nötig, weil in den Antworten häufig wichtige Angaben wie beispielsweise die Start-Adresse der gesendeten Daten fehlen. Anschließend werden die empfangenen Daten evtl. noch konvertiert und schließlich an die jeweilige `Success()`-Funktion (s.o.) übergeben.

4.1.3 Anwendungsschicht: Zyklisch abfragender Modbus-Client

Die Aufgabenstellung beinhaltet, dass regelmäßig Werte des Modbus-Servers abgefragt werden sollen. Hierfür wurde eine neue Schicht eingeführt, die auch die Systemvariablen mit den Daten aus den Modbus-Telegrammen verknüpft. Diese Schicht muss alle nach oben gerichteten Dienstzugangspunkte (also die `Success()`- und `Failed()`-Funktionen des Modbus-Clients) implementieren (s. Kapitel 4.1.2). Sie befindet sich in der CAPL- Datei `PollingModbusClient.can`.

Zu Beginn werden die Konfigurationsdaten aus den Systemvariablen anhand des Knotennames extrahiert und damit der Modbus-Client konfiguriert. Ebenso werden einmalig die Zustände der Ausgangsbits und –register abgefragt, um diese Systemvariablen korrekt zu initialisieren. Werden diese danach verändert, wird ein Telegramm gesendet, das den Zustand der Ausgänge der Busklemme aktualisiert. Außerdem wird der Timer `gtRobin` gestartet, der regelmäßig die Eingangswerte abfragt. Dessen Intervall kann während der Laufzeit geändert werden.

4.1.4 Gerätespezifische Einstellungen

Sowohl der Modbus-Client (Kapitel 4.1.2) als auch der Polling Modbus-Client (Kapitel 4.1.3) benötigen einige Informationen über den Modbus-Server. Für den Stack ist dies die Information über die maximale Anzahl der Ein- und Ausgänge sowie die Größe des Empfangsfensters des Servers. Für die übergeordnete Schicht hingegen ist wichtig zu wissen, an welcher Adresse die Ein- und Ausgänge im Prozessabbild zur Verfügung stehen. Bei den untersuchten Busklemmen von *Wago* und *B+R* unterscheiden sich lediglich die Adressen der Ausgänge, jedoch wurden aus Gründen der Erweiterbarkeit alle Adressen aufgenommen. Gesetzt werden diese Variablen von `DeviceInit(byte Vendor)`, welche in der CAPL-Include-Datei `DeviceInformation.cin` zu finden ist.

In dieser Datei befinden sich auch weitere Funktionen, die für das Analysieren der Geräte im Rahmen der Generierung der Systemvariablenkonfiguration und der Projekt-Datenbank zuständig sind. Somit wurden alle gerätespezifischen Code-Teile in einer Datei implementiert und können zentral bearbeitet und erweitert werden.

4.1.5 Testen des Codes

Da der zyklisch abfragende Modbus-Client (s. Kapitel 4.1.3) nicht alle Funktionen des Modbus-Clients verwendet, wurde ein sequenzieller Test entworfen, der als Anwendungsschicht alle Dienstzugangspunkte des Modbus-Clients anspricht und das zurückgelieferte Ergebnis überprüft. Hierfür wurde eine einfache Zustandsmaschine entworfen, die nach jedem empfangenen Telegramm eine weitere Aktion vornimmt. Sie ist in der Datei `TestTheStack.can` enthalten.

Da der Test sowohl bei der *Wago 750-881* als auch bei der *B+R X20BC0087* erfolgreich durchlaufen wurde, kann der entwickelte Modbus-Client-Stack als voll funktionsfähig erachtet werden.

4.2 Workflow

Im Folgenden soll der Prozess beschrieben werden, der nötig ist, um mit Hilfe des Modbus-Stacks in *CANoe* mit einem Netzwerk mit Modbus-Busklemmen arbeiten zu können.

4.2.1 Anschließen und Konfiguration der Geräte

Zuerst ist es natürlich nötig, den Computer mit *CANoe* als auch die Busklemmen zu installieren. Auf dem Computer muss vorerst nur eine Ethernet-fähige Instanz von *CANoe* eingerichtet sein.

Anschließend müssen die Busklemmen angeschlossen und ihre IP-Adressen sowie die des Computers fest eingestellt werden. Dies ist unter anderem über die Schalter an den Klemmen selbst möglich, hier muss jedoch aufgrund der vielfältigen Bedeutungen vor Allem bei *B+R* das Handbuch konsultiert werden.

Abbildung 12: Versuchsaufbau mit den beiden Busklemmen und der Energieversorgung

Da *B+R* und *Wago* über die Hardware-Schalter unterschiedliche Netze verwenden (`192.168.100.0/24` bzw. `192.168.1.0/24`), empfiehlt es sich, einen DHCP-Server zu betreiben (DHCP-Client-Einstellung bei *B+R*: `0x80`, bei *Wago*: `0xFF`). Falls noch kein Router oder anderer Rechner mit DHCP-Server-Funktionalität verfügbar ist, kann z.B. der *Open DHCP Server*[14] verwendet werden. Bei dieser Methode ist darauf zu achten, dass die Modbus-Klemmen ihre IP-Adresse dauerhaft behalten sollten, da diese in *CANoe* statisch verwendet werden. Der DHCP-Server sollte also bei Einsatz mehrerer Busklemmen die IP-Adressen anhand der MAC-Adressen verteilen.

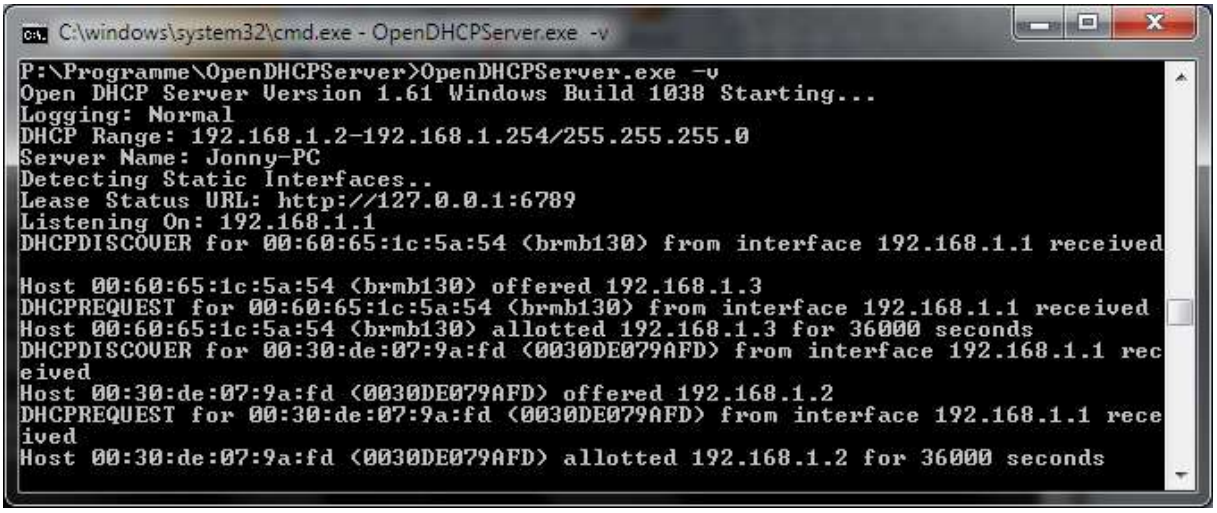


Abbildung 13: Ausgabe des Open DHCP Servers bei Anschluss der Busklemmen

[14] https://sourceforge.net/projects/dhcpserver/ (gestartet als Dienst oder mit `OpenDHCPServer.exe -v`)

4.2.2 Generierung der Systemvariablenkonfiguration und Projekt-Datenbank

Anschließend muss diese Netzwerkkonfiguration in *CANoe* eingebracht werden. Beim Polling-Modbus-Client geschieht dies über die Systemvariablen. Wie in Kapitel 3.3.2 beschrieben, kann hierzu die ebenfalls im Rahmen dieser Arbeit entwickelte *CANoe*-Konfiguration `MakeConfig.cfg` geladen werden. Danach müssen die IP-Adressen der Geräte im CAPL-Code des Netzwerkknotens „MakeConfig“ (`on preStart`) konfiguriert werden. Hierbei ist es möglich, die IP-Adressen der Geräte einzeln einzutragen oder einen IP-Bereich einzustellen, der gescannt werden soll. Es können auch mehrere nebeneinander liegende Netze, die von unterschiedlichen Netzwerkadaptern bedient werden, (z.B. `192.168.1.0/24` und `192.168.2.0/24`) verwendet werden, weshalb momentan die Broadcast-Adresse `.255` übersprungen wird. Sollen mehrere Busklemmen in mehreren Netzen bedient werden, ist es wichtig, das dritte Byte der IP-Adresse ebenfalls in den Namen des Netzwerkknotens einfließen zu lassen, um diese Geräte weiterhin anhand ihres Namens unterscheiden zu können. Beide Optionen können zu Beginn des Codes deaktiviert werden.

Wenn die Messung durchgeführt wird, werden die Systemvariablenkonfiguration und die Projekt-Datenbank an der angegebenen Stelle generiert. Diese Dateien können im endgültigen *CANoe*-Projekt eingebunden werden.

```
Start der Messung 14:13:32
44-0014 Ethernet Treiber Info: [Eth 1] The adapter is not isolated.
Scanning from 192.168.1.2 to 192.168.2.20 with timeout of 10 ms
192.168.1.2  ..........................................................
192.168.2.0  ......................
Analyzing 192.168.1.2: AOs: 0, AIs: 2, DOs: 16, DIs: 2 --> DI2,AI2,DO16
Analyzing 192.168.1.3: AOs: 0, AIs: 7, DOs: 0, DIs: 64 -->
Messungsstopp 14:13:35
```

Abbildung 14: Ausgabe des MakeConfig-Projektes

4.2.3 Konfiguration des Projektes in *CANoe*

Wird das bereits vorhandene Projekt `ModbusNet.cfg` verwendet, werden die Änderungen an den Systemvariablen und der Projekt-Datenbank automatisch übernommen. Anschließend muss pro Busklemme je ein Netzwerkknoten hinzugefügt werden. Diesem muss dann noch der entsprechende Eintrag aus der Datenbank (DB-Knoten) sowie der Pfad zur `PollingModbusClient.can` mitgeteilt werden.

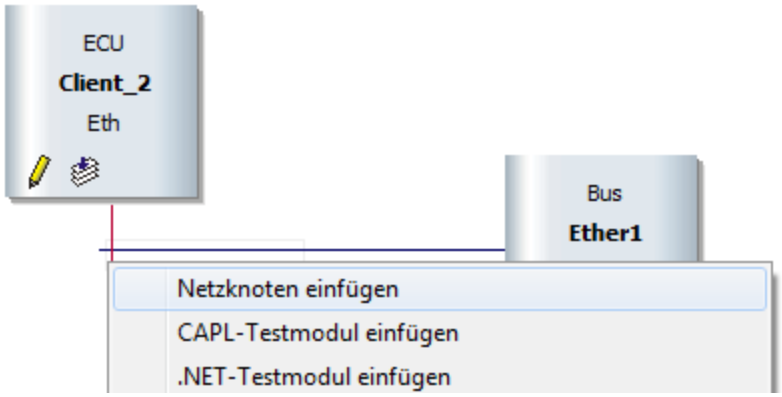


Abbildung 15: Hinzufügen eines Netzwerkknotens im Simulationsaufbau von *CANoe*

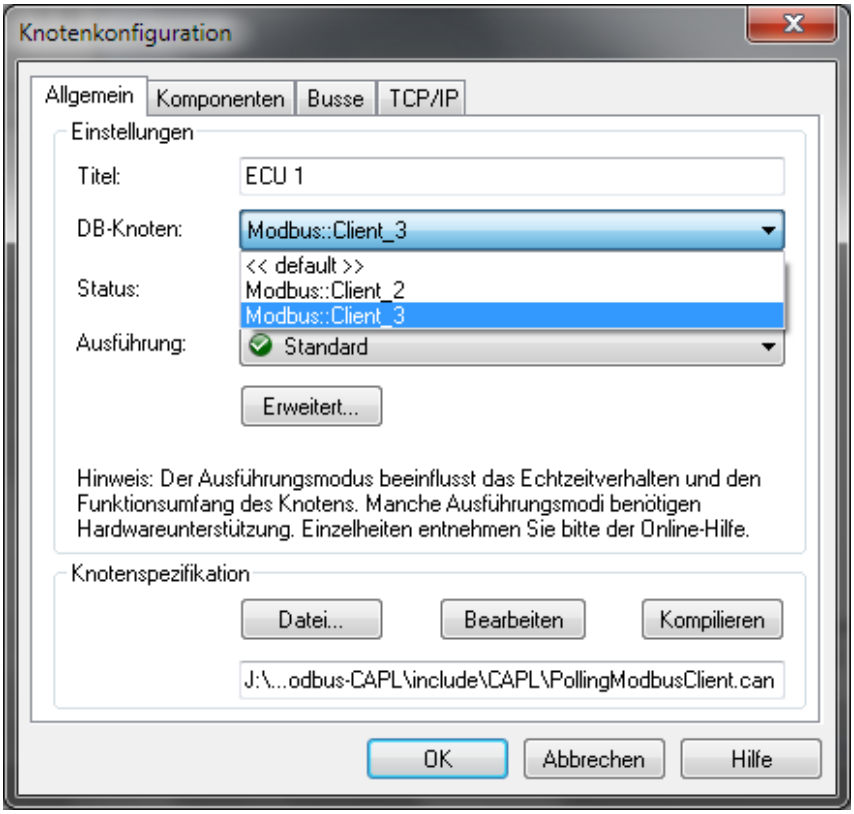


Abbildung 16: Konfiguration des neu erstellten Netzwerkknotens

Der Modbus-Client funktioniert mit allen von *CANoe* angebotenen TCP/IP-Stacks. Normalerweise ist es nicht nötig, für jeden Netzwerkknoten eine eigene Instanz mit eigener IP-Adresse zu erstellen, deshalb können diese den *CANoe* eigenen oder den Betriebssystem-Stack mitbenutzen (s. Abbildung 17). Wird, wie bei Airbus der Fall, der IP-Adressraum `192.168.0/16` verwendet, muss in der aktuellen Version von *CANoe* die Netzmaske angepasst werden. Wird dies versäumt, meldet der verwendete Socket, dass das Zielnetzwerk nicht verfügbar ist.

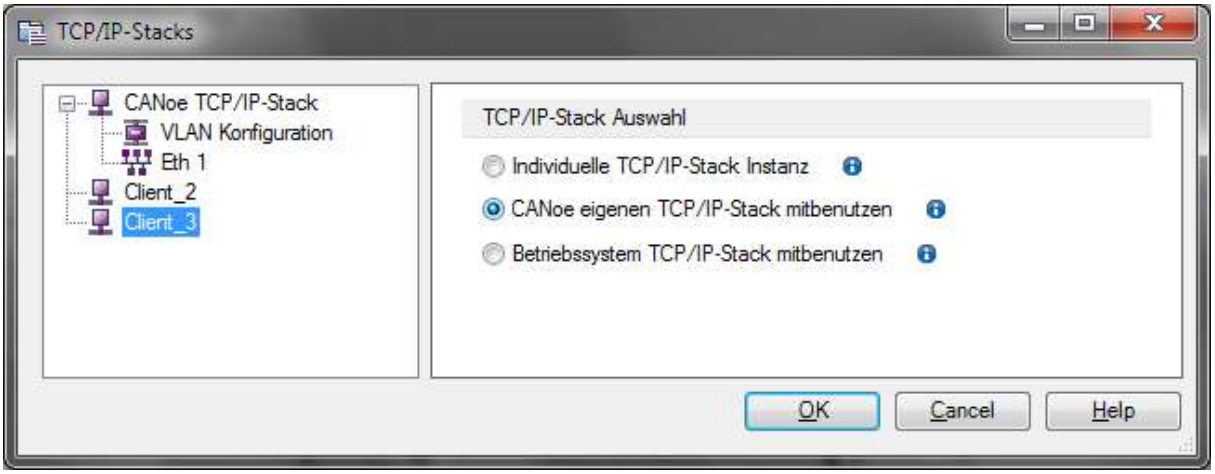


Abbildung 17: Konfiguration der TCP/IP-Stacks in *CANoe*

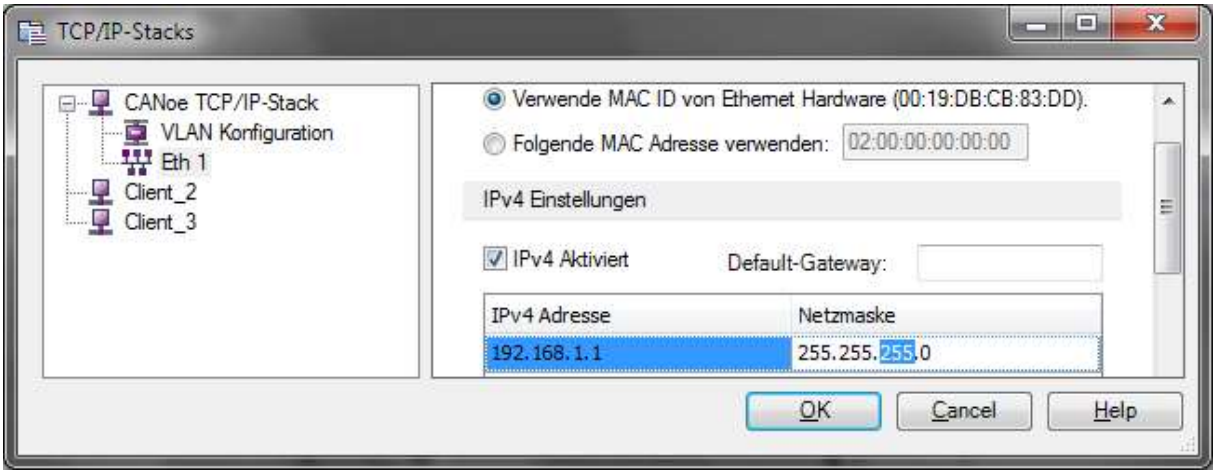


Abbildung 18: Einstellungen des *CANoe* eigenen TCP/IP-Stacks

Die Einstellungen des Modbus-Clients können in der Systemvariablen-Konfiguration verändert werden (s. Abbildung 19). Dort wird global der Timeout eines Modbus-Requests in Millisekunden spezifiziert, genauso können die IP-Adresse und das Intervall, mit dem die Daten abgefragt werden, verändert werden. Die Variablen in den Bereichen „Data“ und „Info“ sollten nicht verändert werden.

Wenn das in Kapitel 2.4.2 beschriebene Mapping von Systemvariablen verwendet werden soll, so müssen noch die Ziel-Systemvariablen (vgl. Abbildung 20) und die Mapping-Anweisungen (vgl. Abbildung 21) erstellt werden. Darauf aufbauend können dann in *CANoe* Panels und CAPL-Programme entworfen werden, welche die grafische Darstellung und Steuerung der Ein- bzw. Ausgänge der Modbus-Klemmen und auch anderer Systeme ermöglichen (vgl. Abbildung 22).

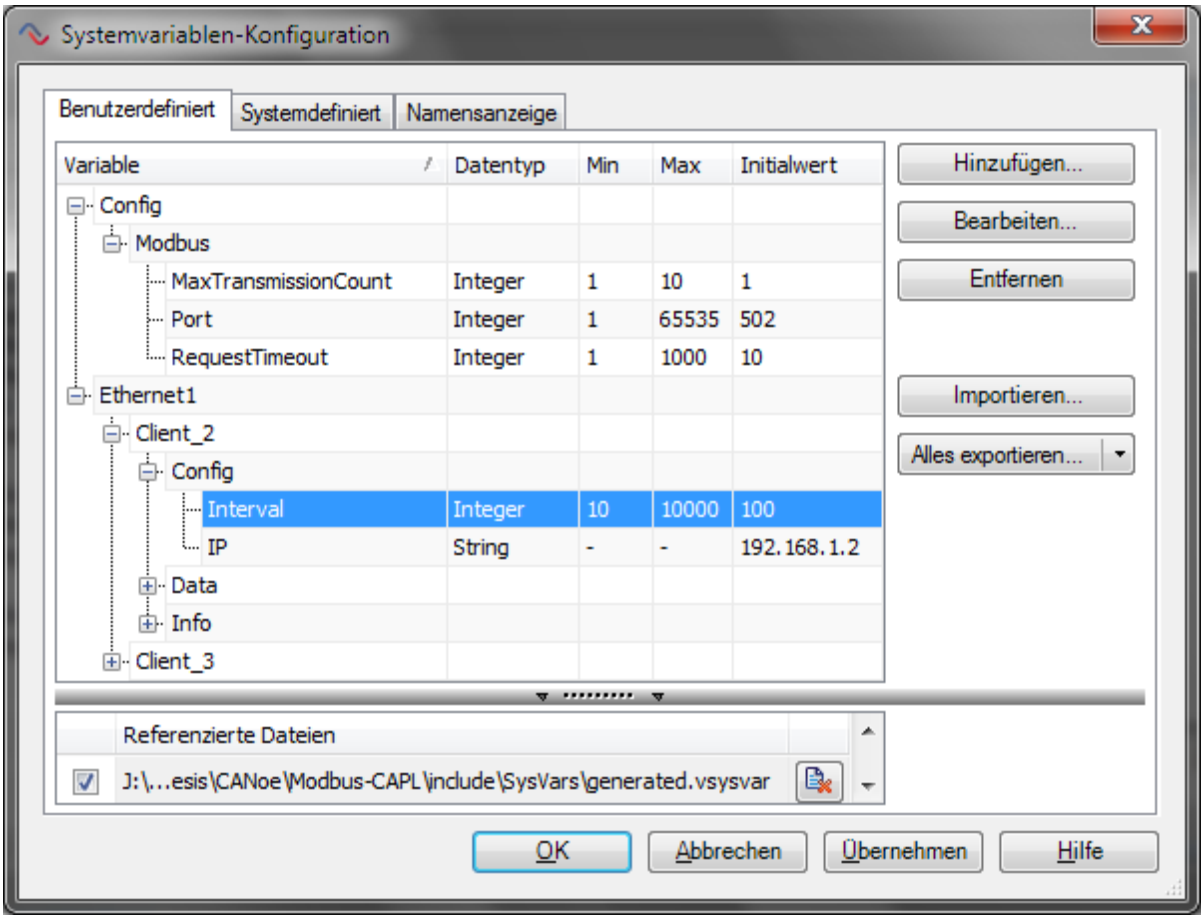


Abbildung 19: Systemvariablen-Konfiguration in *CANoe*

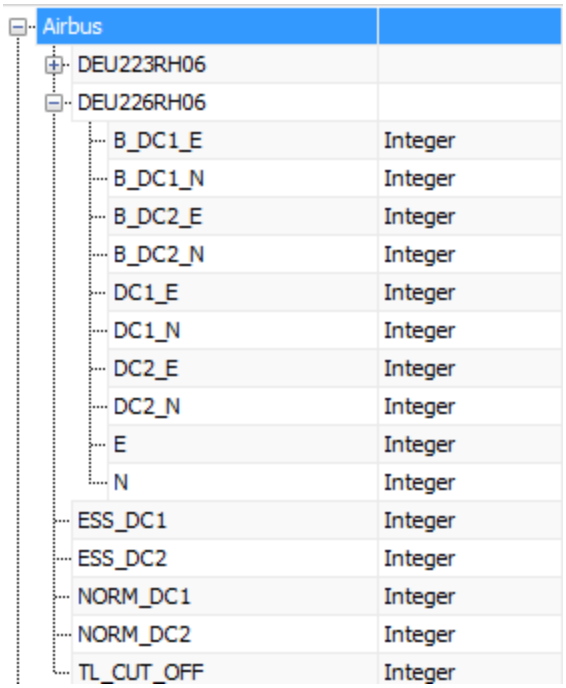


Abbildung 20: Zusätzlich definierte Systemvariablen, welche über sprechendere Namen verfügen

Abbildung 21: Mapping der Modbus-Systemvariablen auf die sprechenderen Variablen

Abbildung 22: Ein für Airbus entworfenes Panel zur Steuerung der Decoding-Encoding-Units

5 Betrachtung der Grenzen

Im Folgenden soll die Obergrenze der entwickelten Lösung ausgelotet werden. Hierfür müssen verschiedene Bereiche beachtet werden.

5.1 Grenzen des Netzwerks

Als Erstes kommt gemeinhin das verwendete Netzwerk als begrenzendes Element in den Sinn. Da Modbus-TCP ebenso wie die davon verwendeten Protokolle (TCP/UDP, IP, Ethernet) in die Datagramme Header einfügen, muss der hierdurch entstandene Overhead[15] vom theoretisch möglichen Durchsatz im Netzwerk abgezogen werden.

Da das zu verwendende 100Mbit/s-Ethernet (100BASE-TX) kein geteiltes Medium verwenden soll und somit kollisionsfrei arbeitet, kann die volkstümliche Weisheit, dass Ethernet maximal zu 50% ausgelastet werden kann, außer Acht gelassen werden. Stattdessen soll hier eine maximale Auslastung von 90% angenommen werden, wobei für diesen geringen Wert jedoch keine belastbare Aussage gefunden werden konnte. Hier mit inbegriffen ist die Betrachtung des Switches. Da dieser bei einer sternförmigen Netzwerktopologie nicht alle Kanäle – sondern nur den zum Master – mit voller Datenrate bedienen muss, muss er nicht besonders hohen Anforderungen genügen. Auch bei der von den Busklemmen unterstützten Linientopologie ist lediglich das letzte Segment zum Master voll ausgelastet, weshalb der in diese Busklemme integrierte Switch entsprechend leistungsfähig sein muss. Insgesamt ergibt sich somit eine maximale Übertragungsrate von

$$r = 0{,}9 \cdot 100 \frac{Mbit}{s} = 90 \frac{Mbit}{s} = 11{,}25 \frac{MByte}{s}.$$

Die Größe des Headers von Modbus-TCP beträgt 62 Byte (vgl. Abbildung 23), die von Modbus-TCP über UDP nur 50 Byte (vgl. Abbildung 24). Da die Größe der Nutzdaten („Payload“) nicht festgelegt ist, sollen hier die Grenzwerte betrachtet werden. Ein Modbus-Datagramm darf wie in Kapitel 2.3.3 beschrieben maximal 260 Byte beinhalten, somit muss die maximale Paketgröße von Ethernet (ca. 1.500 Byte) nicht beachtet werden. Der Overhead eines Pakets berechnet sich wie folgt:

$$o = \frac{Größe\ Header}{Größe\ Paket} = 1 - \frac{Größe\ Nutzdaten}{Größe\ Paket} = 1 - \frac{Größe\ Nutzdaten}{Größe\ Nutzdaten + Größe\ Header}.$$

Da für jedes Paket mit Nutzdaten auch eine Anfrage mit Start-Adresse (2 Byte) und Anzahl der Elemente (2 Byte) gesendet werden muss, ergibt sich der tatsächliche Overhead zu

$$o' = 1 - \frac{Größe\ Nutzdaten}{Größe\ Nutzdaten + 2 \cdot Größe\ Header + 4\ Byte}.$$

Eth			IP											TCP										Modbus					Payload
14			20											20										8					
Dst	Src	Typ	V	H	D	Le	ID	F	C	T	T	CH	Src	Dst	Src	Dst	Seq	Ack	H	F	Wi	CH	Ur	C	Txl	Prc	Le	U	F	
6	6	2				2	2					2	4	4	2	2	4	4			2	2	2		2	2	2			

Abbildung 23: Die Header und ihre Größen der von Modbus-TCP verwendeten Protokolle in maßstabsgetreuer Breite

[15] Quotient aus Größe der hinzugefügten Header und Gesamtgröße des Pakets

Eth			IP										UDP				Modbus					Payload
14			20										8				8					
Dst	Src	Typ	V	H	D	Le	ID	F	C	T	T	CH	Src	Dst	Src	Dst	Le	CH	Txl	Prc	Le	U	F
6	6	2				2	2					2	4	4	2	2	2	2	2	2	2		

Abbildung 24: Die Header und ihre Größen der von Modbus-TCP über UDP verwendeten Protokolle in maßstabsgetreuer Breite

	Abfrage 1 Bit	Abfrage 2000 Bit
TCP	o' = 99,9%	o' = 33,9%
UDP	o' = 99,9%	o' = 29,4%

Tabelle 3: Maximaler und minimaler Overhead des Modbus-Protokolls bei den verwendeten Protokollen TCP und UDP

Setzt man in diese Formel die Werte ein, erhält man als bestes Ergebnis einen Overhead von 29,4% (s. Tabelle 3), welcher durchaus höheren Ansprüchen genügt. Der Download einer 3 kB großen Test-Webseite über http erreichte vergleichbare Werte. Verbessern könnte man diesen Wert hauptsächlich durch Vergrößerung der maximalen Paketlänge. Damit ließe sich, begrenzt durch die maximale Größe der Ethernet-Frames, ein Overhead von lediglich 6,7% erreichen.

Die tatsächlich verfügbare Nutzdatenrate erhält man, indem man den Overhead von der Übertragungsrate abzieht:

$$r' = r - o'[\%] = 90\frac{Mbit}{s} - 29{,}4\% = 63{,}5\frac{Mbit}{s}.$$

Dieser Wert gilt jedoch nur, wenn alle Modbus-Telegramme vollständig ausgenutzt werden; andernfalls fällt er geringer aus. Pro 100 ms können über 100BASE-TX also über 6 Millionen Bits übertragen werden, für deren Verwaltung bei gleichmäßiger Aufteilung auf digitale und analoge Eingänge über 1.500 Busklemmen von *Wago* oder an die 200 Busklemmen von *B+R* nötig wären.

5.2 Antwortzeit der Busklemmen

Für die Verarbeitung der Modbus-Anfragen benötigt ein Server eine gewisse Zeit, welche den maximalen Durchsatz beschränkt. Da beide betrachteten Busklemmen mehrere eingehende Anfragen zwischenspeichern können und diese Anfragen nicht parallel bearbeitet werden, soll im Folgenden die (ohnehin geringe) Übertragungsdauer im Netzwerk außer Acht gelassen werden. Wago gibt in seinem Modbus-Handbuch als maximale Antwortzeit 3 ms an[16]. Mit diesem Wert ergibt sich der Durchsatz einer Busklemme zu

$$r_1 = \frac{Anzahl\ Bits}{Bearbeitungsdauer} = \frac{2000}{3\ ms} = 670\frac{kbit}{s}.$$

Laut Angaben des Wago-Supports können sämtliche Anfragen jedoch innerhalb einer Millisekunde beantwortet werden. Dies verdreifacht den vorher erhaltenen Wert:

$$r_2 = \frac{2000}{1\ ms} = 2\frac{Mbit}{s}.$$

Falls diese Werte auf die Busklemmen von B+R übertragbar sind, können alle Eingänge einer voll besetzten Busklemme (16.384 digitale und 2.048 komplexe Eingänge; insgesamt fast 50.000 Bit) mit einer Periode von 40 ms abgefragt werden.

[16] Wago Modbus Spezifikation (WagoMb), S.18

5.3 Grenzen der Software

Die empfangenen Daten müssen in *CANoe* verarbeitet werden, was somit hohe Anforderungen an die verwendete Hardware stellt. In *CANoe* Version 8.2 können gemäß der *Vector Informatik GmbH* bis eine Million Zustandsaktualisierungen pro Sekunde verarbeitet werden; da diese große Zahl bisher jedoch schlicht nicht benötigt wurde, wurde dies noch nie überprüft.

Ein kleiner Test auf recht schwacher Hardware[17] zeigte, dass bereits 6.000 digitale Eingänge pro Sekunde für den Modbus-Client eine kaum zu bewältigende Aufgabe darstellten. Es muss jedoch gesagt werden, dass der in dieser Arbeit implementierte CAPL-Modbus-Stack (vgl. Kapitel 3.2) nicht auf Geschwindigkeit programmiert wurde. So finden z.B. über 100 Funktionsaufrufe statt, welche den Programmablauf im Log-Fenster in *CANoe* dokumentieren, um mögliche Fehler nachvollziehen zu können. Dabei wird zur Laufzeit entschieden, ob eine Nachricht angezeigt werden soll, was relativ rechenintensiv ist.

Eine Optimierung des CAPL-Codes auf Geschwindigkeit, die Verwendung des Ethernet-Interaction-Layers (vgl. Kapitel 3.1) oder das Erstellen einer C-Bibliothek mit Modbus-Funktionen, welche in CAPL eingebunden werden kann, sind Möglichkeiten, um den Rechenaufwand zu reduzieren. An dieser Stelle muss bei Bedarf als Erstes angesetzt werden, da es die größte Einschränkung darstellt und trotzdem mit relativ geringem Aufwand große Gewinne erzielt werden können.

[17] AMD Athlon™ 64 X2 Dual Core Processor 5600+, 2x 2.800 MHz, 2 GB RAM

6 Zusammenfassung und Entwicklungspotenzial

Wie in und mit dieser Arbeit gezeigt wurde, können die Busklemmen *Wago 750-881* und *B+R X20BC0087* mit Hilfe des Modbus-TCP-Protokolls auf recht einfache Weise in *CANoe* integriert werden. Durch die Simplizität des Protokolls konnte innerhalb weniger Wochen ein funktionsfähiger Modbus-Client implementiert werden, welcher die digitalen und komplexen Ein- und Ausgänge der Busklemmen für den Endanwender nahezu transparent in die Systemvariablen von *CANoe* abbildet. Ebenso gut kann aber auch der entwickelte Modbus-Stack anderen Anwendungen zu Grunde gelegt werden.

Durch die ebenfalls erstellte *CANoe*-Konfiguration zur Generierung der Konfigurationsdateien, inklusive Analyse der im Netzwerk befindlichen Busklemmen, ist der initiale Aufwand zur Verwendung des Modbus-Clients recht gering und in wenigen Minuten erledigt. Bereits vorher können weitere Programme oder Oberflächen implementiert werden, die Variablen mit sprechenden Namen verwenden, auf welche die generischen Modbus-Variablen schließlich gemappt werden (vgl. Kapitel 2.4.2).

Sollten in *CANoe* eine große Anzahl von Variablen über Modbus verwaltet werden, könnte durch eine Einschränkung der Flexibilität von Modbus zur Analyse der empfangenen Telegramme auf den Ethernet-Interaction-Layer von *CANoe* zurückgegriffen werden (vgl. Kapitel 3.1.3). Hiermit wäre laut Aussagen eines Mitarbeiters der *Vector Informatik GmbH* eine deutliche Steigerung der Leistungsfähigkeit möglich. Beinahe ebenso gut können die bisher komplett in CAPL geschriebenen Funktionen in eine C-Bibliothek ausgelagert werden. Hierbei bliebe die volle Flexibilität des Modbus-Protokolls erhalten, jedoch wäre die Analyse der Telegramme nicht ganz so effizient wie mit dem Interaction-Layer. Das Netzwerk stellt bei Verwendung von Switches hinsichtlich der Datenmenge keine Begrenzung dar, auch die Latenz dürfte in kleinen Netzen annehmbar gering sein. Sollte in der Zukunft das Netzwerk doch ausgelastet sein, ist es (zumindest für Busklemmen des Herstellers *Wago*) möglich, Anfragen per Multicast zu empfangen. Somit könnten mehrere (wahrscheinlich identisch konfigurierte) Busklemmen simultan angesprochen werden, was die bei der Generierung der Abfrage entstehende Last des Modbus-Clients verringern würde. Große Gewinne sind hierdurch jedoch nicht zu erwarten.

Eine Modifikation des Anwendungsfelds könnte ebenso ein höheres Sicherheitspotenzial erfordern, z.B. wenn den eingesetzten Busklemmen oder dem Netzwerk nicht vertraut werden kann. Um dieses zu erreichen, sollte vor Allem die Verarbeitung der eingehenden Telegramme geprüft und verbessert werden. In der aktuellen Version des Modbus-Stacks wird beispielsweise – wie in Kapitel 4.1.2 beschrieben – darauf vertraut, dass an den Netzwerk-Port des Modbus-Clients standard-konforme Modbus-Telegramme gesendet werden. Es sind zwar Prüfungen eingebaut, welche beispielsweise inkorrekte Längenangaben im Modbus-Application-Header oder ein Fehlen desselbigen in den meisten Fällen erkennen können, jedoch können falsche Telegramme unter Umständen die Verarbeitung direkt nachfolgender Modbus-Telegramme behindern. Da CAPL Speicherüberläufe abfängt, besteht keine unmittelbare Gefahr für den Rechner, auf dem *CANoe* läuft. Nichtsdestotrotz bestehen hinsichtlich der Sicherheit des Modbus-Clients somit noch Lücken, welche in einem offenen Netz ausgenutzt werden könnten. Da Modbus als Protokoll selbst jedoch keine Sicherheitsaspekte wie beispielsweise eine Authentifizierung des Modbus-Servers gegenüber dem Client beinhaltet,

sollten im Fall eines offenen Netzes ohnehin ergänzende Lösungen (z.B. ein VPN[18]) eingesetzt werden. Die angesprochenen Sicherheitsmängel sind somit als marginal zu betrachten.

Ein Verbesserungs-Potenzial beim Workflow liegt in der Generierung der Mapping-Daten: Da das Mapping der Variablen aktuell noch großen manuellen Aufwand beinhaltet, ist es ratsam, mit Hilfe bereits existierender Verschaltungspläne oder Ähnlichem die Mapping-Tabelle zu generieren. Das Mapping ist jedoch stark anwendungsspezifisch und konnte deshalb in dieser Arbeit nicht näher behandelt werden.

Diverse Konstellationen könnten es nötig machen, dass *CANoe* auch als Modbus-Master agiert; so könnte z.B. ein azyklischer Datenaustausch zwischen zwei Instanzen von *CANoe* über Modbus-TCP realisiert werden. Ebenso könnte ein Gerät mit Modbus-Master-Funktionalität entwickelt, in *CANoe* simuliert und dabei von einem Modbus-Client gesteuert werden. Obwohl sich ein Modbus-Master stark von einem –Client unterscheidet, kann bei dessen Implementierung auf einige Aspekte dieser Arbeit zurückgegriffen werden: Es können ebenfalls entweder Signale oder Systemvariablen sowie die dahinter liegenden Prozesse als Schnittstelle zu *CANoe* verwendet werden (vgl. Kapitel 2.4.2). Ebenso muss je nach verwendeter Netzwerkschicht (s. Kapitel 4.1.1) lediglich eine Socket::Listen()-Funktion hinzugefügt werden. Bei der Implementierung der „mittleren“ Modbus-Server-Schicht können die bereits vorhandenen Strukturen genutzt werden.

Insgesamt ist Modbus-TCP ein einfaches, zuverlässiges Protokoll zum Ethernet-basierten Datenaustausch zwischen einem Master und einem Client, mit dessen Hilfe die verwendeten Busklemmen gut in *CANoe* eingebunden werden können.

[18] VPN: Virtual Private Network, ein privates Rechnernetz, das auf einer öffentlichen Netzwerk-Infrastruktur aufgebaut ist

7 Literaturverzeichnis

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H. (28.04.2014). *Modbus/TCP X20BC0087 Anwenderhandbuch.* Abgerufen am 25.06.2014 von http://www.br-automation.com/download.php?type=download&folder=BRP4440000000000000299389&file=b_ModbusTCP_X20BC0087_GER.pdf (BRMbHr)

HMS Industrial Networks GmbH. (2012). *Definition der Protokollfamilien*. Abgerufen am 19.08.2014 von http://www.feldbusse.de/Normung/protokollfamilien.shtml

HMS Industrial Networks GmbH. (2012). *EtherNet/IP*. Abgerufen am 17.06.2014 von http://www.feldbusse.de/EthernetIP/ethernetip.shtml

Ihle, M. (01.05.2011). *Rapid Prototyping for Embedded Systems - Vorlesungsfolien 09: Modultest, Rückgekoppelte Systeme.* Abgerufen am 24.07.2014 von http://www.home.hs-karlsruhe.de/~ihma0001/download_gesichert/RPES-09_Rev1.00.pdf

Mehta, K. (09.09.2004). *ProfiNet, Modbus/TCP or Ethernet/IP*. Abgerufen am 17.06.2014 von http://www.control.com/thread/1026200383

Modbus Organization. (2014). *About us*. Abgerufen am 17.06.2014 von http://www.modbus.org/about_us.php

Modbus Organization. *Modbus TCP Toolkit.* Abgerufen am 17.06.2014 von http://www.modbus.org/docs/Toolkit_a.pdf

Modbus-IDA. (28.12.2006). *Modbus Application Protocol Specification.* Abgerufen am 25.06.2014 von http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf (MAPP)

VARAN-BUS-Nutzerorganisation. (2014). *VARAN Echtzeit Datennetz für die Automatisierung*. Abgerufen am 16.05.2014 von http://www.varan-bus.net/deutsch.htm

Vector Informatik GmbH. (2013). *Produktinformation CANoe.* Abgerufen am 27.06.2014 von http://www.vector.com/pi_canoe_de (VecPI)

WAGO Kontakttechnik GmbH & Co. KG. (2007). Modbuskommunikation zwischen WAGO Ethernet Kopplern und Controllern. Abgerufen am 26.08.2014 von http://www.wago.com/wagoweb/documentation/app_note/a3000/a300003d.pdf (WagoMb)

Wikipedia. (22.10.2013). *Versatile Automation Random Access Network*. Abgerufen am 16.05.2014 von http://de.wikipedia.org/wiki/VARAN

Wikipedia. (06.02.2014). Abgerufen am 16.06.2014 von http://de.wikipedia.org/wiki/Modbus

Wikipedia. (14.06.2014). *Controller Area Network.* Abgerufen am 02.07.2014 von https://de.wikipedia.org/wiki/Controller_Area_Network

Wikipedia. (29.01.2014). *EtherNet/IP*. Abgerufen am 17.06.2014 von http://de.wikipedia.org/wiki/EtherNet/IP

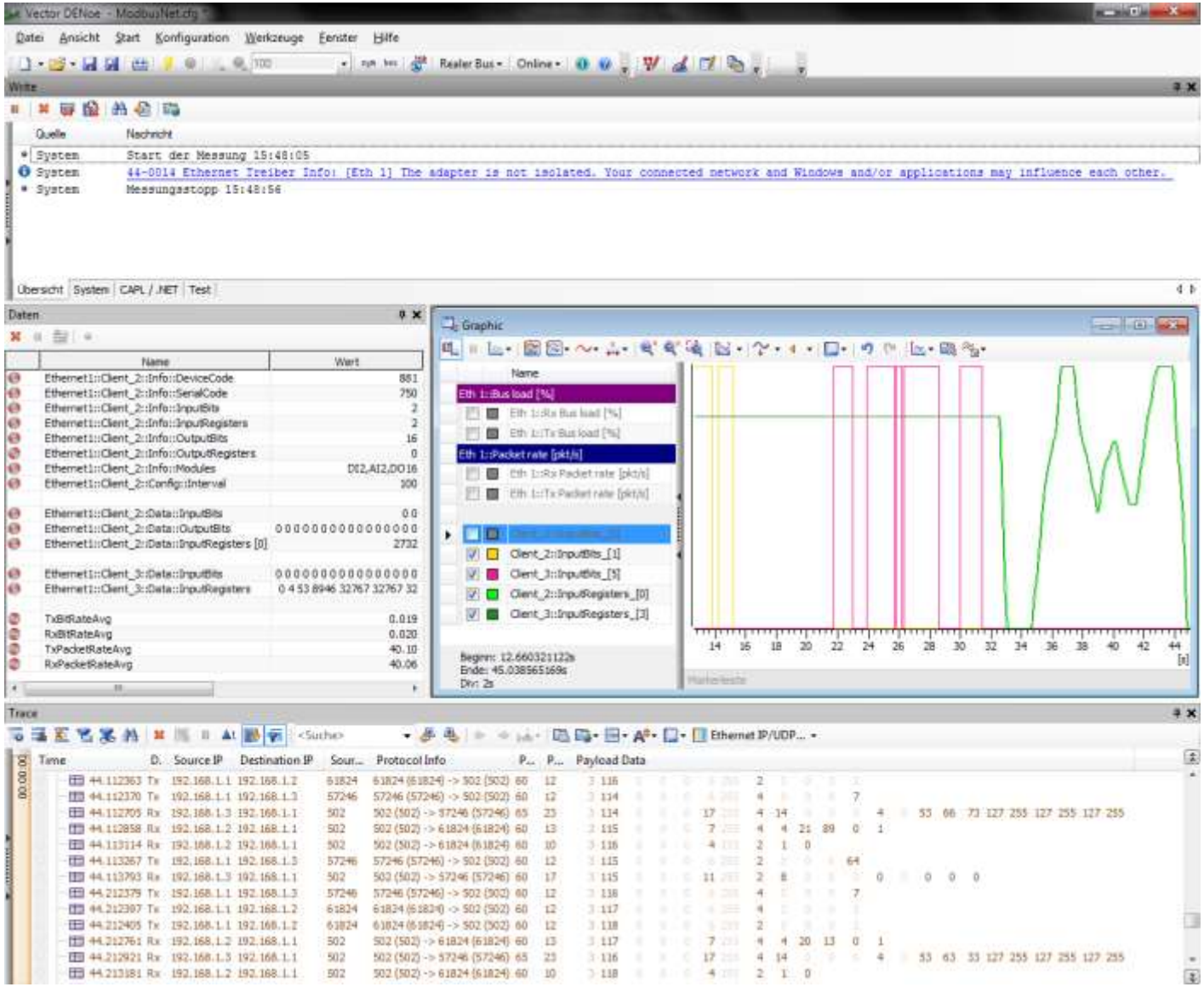
8 Abbildungsverzeichnis

9 Anhang

9.1 Bewertung diverser Bussysteme

	Ethernet	von Klemme unterst.	geringe Lizenzkosten	Simplizität	dig. und analog. I/Os, ~15000
ARCNET					
AS-Interface					
BACnet					
BITBUS					
CAN					
EtherCAT					
Ethernet Powerlink					
EtherNet/IP			Wenn Verkauf: 2.000€	Objekte	hauptsächlich analog
FlexRay-Bus					
Hart					
INTERBUS					
KNX					
LCN					
LIN					
Loconet					
LON					
M-Bus					
Modbus			vollständig offen	Funktionen	analog & digital
MOST-Bus					
P-NET					
PROFIBUS					
PROFINET		Wago 750-370			
SafetyBUS					
Time-Triggered Protocol					
VARAN			nicht offen zugänglich		

9.2 Oberfläche von *CANoe* während einer Messung mit beiden Busklemmen



9.3 Wichtige Punkte bei Verwendung des Modbus-Clients

- Auswahl der Busklemmen
  - Die Wago *750-881* kann 512 digitale Ein- und Ausgänge sowie 256 komplexe Ein- und Ausgänge über Modbus zur Verfügung stellen, während die *B+R X20BC0087* bis zu 16.384 digitale und 2.048 komplexe Ein- und Ausgänge verwalten kann.
  - Die Busklemme von *B+R* kann den Tests in dieser Arbeit zufolge bis zu acht Anfragen gleichzeitig verarbeiten, die Klemme von *Wago* kann fünf Anfragen zwischenspeichern.
  - Die *Wago 750-881* ist eine CPU, auf der auch SPS-Programme laufen können. Ein entsprechende *B+R*-Klemme (z.B. X20CP1483) ist ebenfalls auf dem Markt erhältlich, jedoch muss diese für den Einsatz (auch als einfacher Modbus-Master) programmiert werden.
  - Bei *B+R* wird der negative Pol der Versorgungsspannung als digitale ‚1' kodiert, bei *Wago* hingegen der positive Pol. Es ist möglich, dass hierzu weitere IO-Module existieren, die sich anders verhalten.
- Mapping
  - Die *X20BC0087* von *B+R* stellt drei komplexe Eingänge an den Beginn des Prozessabbilds. Der erste tatsächliche komplexe Eingang steht somit an vierter Stelle (Index 3).
  - Darüber hinaus füllt die *X20BC0087* die benutzten Bytes eines digitalen Moduls voll aus. Somit benötigt ein Modul mit zwölf digitalen Eingängen (DI12) zwei ganze Byte, die Eingänge werden an den (relativen) Adressen 0 bis 11 zur Verfügung gestellt. Die folgenden vier Bit sind nutzlos und der erste digitale Eingang des nächsten Moduls liegt somit an Adresse 16.

9.4 Bekannte Fehler

- CANoe
  - „Das Zielnetzwerk ist nicht verfügbar"
    Wird eine ungewöhnliches IP-Netz verwendet (beispielsweise `192.168.0/16`), so übernimmt *CANoe* in Version 8.2 nicht die korrekte Subnetzmaske. Diese muss deshalb korrigiert werden (s. Abbildung 18)
- Modbus
  - Bei Messungsstart werden die meisten Pakete nicht gesendet, wodurch viele Timeouts gemeldet werden (vor Allem bei `TestTheStack`)
    Ein Bug im TCP-Stack von *CANoe* 8.2 führt dazu, dass, solange der Stack auf eine ARP-Response wartet, neue Sende-Befehle die alten überschreiben. Der Bug wird demnächst behoben.

10 Erklärung

Ich erkläre an Eides Statt, dass ich die hier vorgelegte Bachelor-Thesis selbstständig und ausschließlich unter Verwendung der angegebenen Literatur und sonstigen Hilfsmittel verfasst habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgeleg

_____ _____

11 Programmcode

Das komplette Projekt (inklusive *CANoe*-Konfigurationen und beispielhaften DBC-, SysVar-, Panel-Dateien und CAPL-Code für die Panels) kann von folgendem Git-Server abgerufen werden. Es wird darum gebeten, eventuelle Fehler oder Verbesserungen an den Entwickler per Issue/Pull-Request zu melden.

https://git.nordlichter-brv.de/Jonny007-MKD/Bachelorthesis/

Im Verzeichnis `Modbus-DLL/` im Repository ist der Beginn einer Ethernet-Interaction-Layer-Implementierung (vgl. Kapitel 3.1) zu finden, die Verwendung von Signalen wurde allerdings noch nicht implementiert. Der Polling-Modbus-Client-Stack nach Kapitel 3.2 sowie die zugehörigen Code-Teile `MakeConfig` (s. Kapitel 4.2.2) und TestTheStack (s. Kapitel 4.1.5) sind im Ordner `Modbus-CAPL/include/CAPL/` zu finden und werden in dieser gekürzten Fassung nicht abgedruckt.

12 Danksagungen

Das Verfassen dieser Abschlussarbeit und das Meistern der damit verbundenen Hindernisse hat mir sehr viel Spaß gemacht. Es ist jedoch selbstverständlich, dass eine solche Arbeit nicht ohne Unterstützung anderer entstehen kann. Bei allen, die mir in den letzten Monaten halfen, möchte ich mich an dieser Stelle bedanken.

Zuerst danke ich meiner Betreuerin von der Hochschule Karlsruhe, Prof. Dr. Marianne Katz, die trotz der großen Entfernung sofort bereit war, diese Arbeit zu betreuen und mich zu unterstützen. Weiterhin danke ich meinem Betreuer Jörn Haase von der Vector Informatik GmbH, der mich auf ausgezeichnete Art und Weise anleitete und alle meine Fragen kompetent beantworten konnte. Jörn ermöglichte es, dass das Arbeiten in einer solch positiven Atmosphäre stattfand.

Anschließend möchte ich Daniel Bilir und Andre Schäfer für die gute und freundschaftliche Zusammenarbeit sowie den Mitarbeitern der *WAGO Kontakttechnik GmbH & Co. KG* (*Wago*) sowie der *Bernecker + Rainer Industrie Elektronik Ges.m.b.H.* (*B+R*) für ihre Unterstützung bei Fragen zu den Busklemmen danken.

Schließlich danke ich allen, die diese Arbeit Korrektur lasen und mich mit hilfreichem Input unterstützten, darunter Jelka Lamke, für ihre unverzichtbare Mithilfe.

Am meisten danke ich meiner Frau Laura Wendeborn sowie meiner Tochter Maja Wendeborn, die mir in allen Phasen dieser Arbeit mit viel Geduld, aufmunternden Worten und auch tatkräftiger Hilfe zur Seite standen und das Leben so lebenswert machen.