Universität Rostock
Fakultät für Ingenieurwissenschaften
Fachbereich Elektrotechnik & Informationstechnik
Institut für Automatisierungstechnik

Kleiner Beleg

Inbetriebnahme und Programmierung einer CAN-API

vorgelegt von
Jan Blumenthal

Rostock, 10. Mai 2001

Inhaltsverzeichnis

Verzeichnis der verwendeten Abkürzungen und Symbole

ACK	Acknowledge (Bestätigung eines Paketes)
ACR	Acceptance code register
AMR	Acceptance mask register
API	Application programable interface
C166	16-Bit Mikrocontroller von Siemens
CAN	Feldbus (Controller Area network)
CANopen	Protokoll für CAN-Bus
CCD	Charge-coupled device (Hochauflösender Sensor)
CRC	Cyclic Redundancy Checksum
DeviveNET	Protokoll für CAN-Bus
EoF	End of Frame
ID	Identifier
ISO	International standard organisation
Kbs	Kilobit/s
Mbs	Megabit/s
OSI	Open Systems Interconnection
PC98	Spezifikation von Mikrosoft und Intel über Aufbau von Standard-PC’s
PCI	Peripheral components interconnect
RS232	Serielle Schnittstelle
SJA1000	Controller zum Anschluß an CAN-Feldbus
SoF	Start of Frame
TCP	Transport Control Protokol
VCI	Virtual programable interface
VPN	Virtual private network

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Präzisierung der Aufgabenstellung

Die OPTOLOGIC Mess- und Systemelektronik GmbH entwickelt und fertigt intelligente CCD-Zeilenkameras für die industrielle Prozessautomation. Für komplexe Messaufgaben ist es notwendig, mehrere Kameras miteinander zu vernetzen. Als Implementierungsgrundlage für eine solche Feldbusvernetzung wurde CAN favorisiert.

Um Komplikationen und Zeitaufwand bei der Entwicklung eines proprietären firmeninternen Standards zu vermeiden, hat sich OPTOLOGIC für die Implementierung einer standardisierten Applikationsschicht entschieden. Die Wahl fiel auf CANopen. Die Aufgabe dieser Studienarbeit bestand darin, die Voraussetzungen für eine CANopen Umsetzung zu schaffen.

Abbildung 1: CCD-Kamera von OPTOLOGIC

Die CCD-Zeilenkameras beinhalten neben dem Sensor auch einen modular aufgebauten Mikrocomputer mit eigenem Bussystem und einem Mikrocontroller C166 von Infineon. Dieser Computer dient der digitalen Signalverarbeitung der Meßwerte, die allerdings größtenteils in Hardwaremodulen vorgenommen wird, um höhere Performancegewinne zu erzielen. Je nach Kundenwunsch können weitere spezielle Module hinzugeordert werden, die die Funktionalität der Kamera auf das entsprechende Einsatzgebiet optimieren. In der Vergangenheit wurden die Ergebnisse der Meßwertanalyse über eine serielle Verbindung an einen Leit-PC übertragen. Da diese Lösung für komplexe Aufgaben äußerst unflexibel ist, besteht nun die dringende Aufgabe, ein Netzwerkmodul nach Industriestandard zu fertigen.

Die Wahl der Kommunikationshardware fiel auf den CAN-Bus, da er in weiten Teilen der Industrie eingesetzt wird, z.B. Automobilbau. Dadurch wird die Integration der Kameras in bestehende Systeme deutlich erleichtert. Die zweite wesentliche Eigenschaft des CAN-Busses ist die Echtzeitfähigkeit, die für die CCD-Zeilenkameras von entscheidender Bedeutung ist, um Ergebnisse und vor allem Fehlermeldungen im definierten Zeitfenster erfolgreich zu übertragen. Das CAN-Protokoll weist allerdings auch eine Vielzahl von Schwächen auf, die vor allem in komplexeren Applikationen deutlich werden.

Es ist nach dem ISO-OSI Schichtenmodell ein Layer 2 Protokoll (Data Link Layer) und beinhal-

tet daher keinerlei Mechanismen zur Übertragung von größeren Datenströmen, zur Fehlerkontrolle, Ausfallsicherheit und Synchronisation von Netzwerkteilnehmern. Diese Aufgaben werden laut ISO Spezifikation von höheren Schichten übernommen. Aus diesem Grund entstand in der Vergangenheit eine Vielzahl proprietärer Standards, die die Einbindung von Kamera-Applikationen in bestehende Netzwerke deutlich erschwert.

Aus den genannten Gründen ist es notwendig, eine CAN-Anbindung und zusätzlich ein Protokoll für die höheren Schichten zu implementieren. Für den CAN-Bus existieren derzeit zwei speziell entwickelte Protokolle, die die genannten Anforderungen erfüllen, *DeviceNET* und *CANopen*. Im Rahmen mehrerer Projektsitzungen wurde firmenintern entschieden, CANopen zu verwenden, da es sich um eine europäische Entwicklung handelt und zunehmend als europaweiten Standard etabliert, wobei DeviceNET vorwiegend in den USA vertreten ist.

1.2 Die Einsatzszenarien

Die Einsatzgebiete der Kameras in der Industrie sind vor allem kontinuierliche Überwachungsvorgänge, z. B. Erkennung von Oberflächenstörungen. Jede Kamera kann aufgrund ihrer Fähigkeiten nur einen gewissen Bereich beobachten. Ist dieser Bereich kleiner als das zu kontrollierende Gebiet, müssen mehrere Kameras versetzt die Überwachung übernehmen, wodurch Überlappungseffekte auftreten, die nur durch eine nachgeschaltete digitale Signalverarbeitung bearbeitet werden können. Die Kameras nehmen eine Vorauswertung der Aufnahmen vor und übertragen die Ergebnisse in der Regel zu einem Leitrechner, meistens ein handelsüblicher PC mit CAN-Karte (s. Abb. 2). Denkbar ist auch eine Masterkamera, die eine Endauswertung vornimmt und die Ergebnisse weiterleitet (s. Abb. 3).

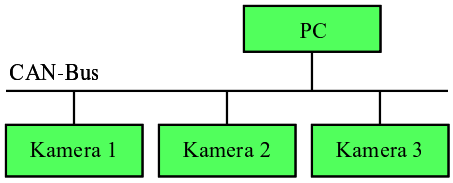

Abbildung 2: Ein Leit-PC steuert 3 Netzwerkteilnehmer(Standardapplikation)

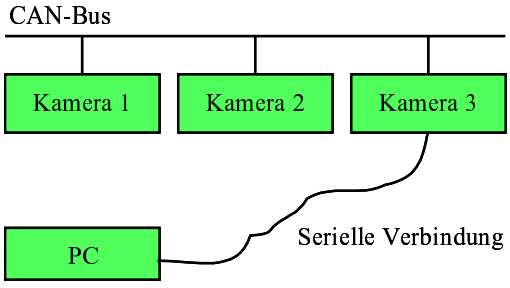

Abbildung 3: Ein Leit-PC steuert ein CAN-Bus Netzwerk und leitet die Ergebnisse weiter

1.3 Die Hardwareplattformen

Für die Erfüllung der Aufgabe muß vorher klar definiert werden, auf welchen Hardwareplattformen die Software laufen soll. Es gilt dabei besonders auf Performance und Speicherverbrauch zu achten.

Die Kamera von OPTOLOGIC arbeitet, wie eingangs erwähnt, mit einem C166 Mikrocontroller, der bist zu 16MB addressieren kann. Allerdings ist aus Kostengründen üblicherweise deutlich weniger bestückt. Da die CPU auch für die Echtzeitverarbeitung der Daten herangezogen wird, darf die CANopen Implementierung nur relativ wenig Rechenzeit an sich binden.

In Abb. 2 ist ein PC gezeigt, der eine CAN-Karte enthält. Eine Betrachtung hinsichtlich des Speicherverbauchs ist bei solchen PC's hinfällig, da RAM-Bausteine in Megabyte-Bereichen deutlich weniger kosten als in Embedded PC's. Die Performancebetrachtung ist ebenfalls ohne Belang, da ein aktueller Mikroprozessor um Größenordnungen schneller, im Vergleich zu einem C166, ist. Um einen Echtzeitbetrieb zu ermöglichen, ist allerdings zu beachten, dass auch ein passendes Betriebssystem gewählt wird, welches u.U. auf virtuellen RAM als unkalkulierbare Performancebremse verzichtet.

Die Implementierung des CAN-Protokolls gestaltet sich in so einem PC etwas anders als in Mikrocontrollern, da jeder Hersteller einer CAN-Karte einen eigenen Betriebssystemtreiber mitliefert, auf den in der Applikation zugegriffen werden muss. Es ist also grundsätzlich davon auszugehen, dass der Zugriff auf die Hardware - je nach Hardwareplattform und Betriebssystem - deutlich unterschiedlich erfolgt.

Für den PC wurde aufgrund der PC98 Spezifikation eine PCI-Karte gewählt. OPTOLOGIC entschied sich für den Erwerb der CAN-PCI Karte von IXXAT, da sie eine umfangreiche Programmierbibliothek (VCI=Virtual programable interface) zur Verfügung stellt und auch in einer Vielzahl weiterer firmeneigener Komponenten verwendet wird.

In der Themenstellung von OPTOLOGIC war eine weitere Hardwareplattform in der mittleren Planung enthalten. Es handelte sich um einen Hyperstone DSP. Dieser Mikrocontroller arbeitet nach dem Harvard-Speichermodell (getrennter Speicher für Programme und Daten). Die beiden ersten Versionen arbeiten nach dem von-Neuman Prinzip (gemeinsamer Speicher für Programme und Daten).

Die Lieferbarkeit des Hyperstone war bis zum Abschluß der Arbeit nicht gegeben, sodass eine testweise Umsetzung nicht möglich war.

1.4 Die Softwareziele

Die Verwendung von Netzwerken in der industriellen Praxis zeigt eine Fülle weiterer Möglichkeiten zur effektiven Nutzung von Resourcen auf.

In Abb. 4 ist ein CAN-Netzwerk inkl. Master-Kamera dargestellt, die die Ergebnisse an einen Leit-PC weiterleitet. Angenommen, in dem Netzwerk tritt plötzlich ein Fehler auf, kann es äußerst kostspielig sein, vor Ort nach dem Fehler zu suchen, wenn dazu erst größere Anfahrtswege nötig sein. Viel sinnvoller ist es, wenn man sich *remote* über ein Virtual Private Network(VPN) in die Anlage einwählen kann und versucht, den Fehler beheben. Um dies zu ermöglichen, sind allerdings weitere Gedanken zur Konzeption der Software nötig. In diesem Fall wird eine Gateway-Funktion in dem Leit-PC erforderlich. Zusätzlich muss beachtet werden, dass die Verbindung zwischen Leit-PC und Remote-PC nicht echtzeitfähig sein kann, was aber für eine erfolgreiche Fehlerkorrektur nicht entscheident ist. In so einem Fall ist es wichtig, das Netzwerk so zu konfigurieren, dass es wieder anläuft. Prinzipiell sind aktuelle nicht-echtzeitfähige Netzwerke, wie Ethernet-LAN's bzw. Standleitungen schnell genug, um Daten im erforderlichen Zeitfenster sicher zu übertragen, eben nur nicht deterministisch.

In Abb. 5 ist ein Leit-PC dargestellt, der gleichzeitig zwei CAN-Netzwerke überwacht. Dies erfordert weitere Überlegungen, denn die Applikation auf dem PC muss mehrere CAN-Instanzen zulassen.

Eine mögliche Anwendung wäre z.B. die Fernwartung mehrerer CAN-Netzwerke von dem LIXUS-i

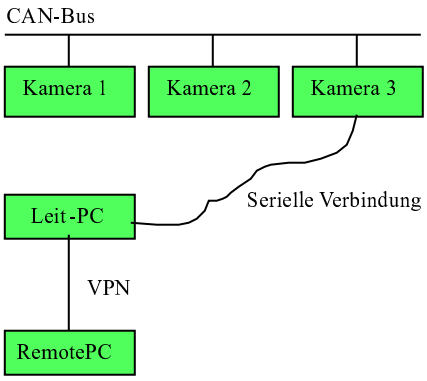

Abbildung 4: Fernwartung eines CAN-Netzwerks per Virtual-Private-Network (VPN)

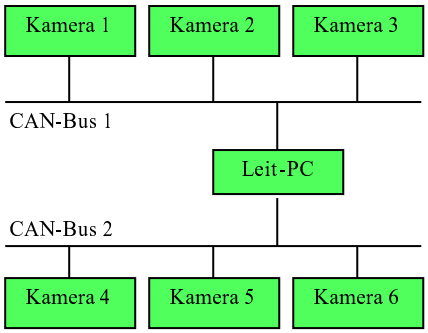

Abbildung 5: Überwachung zweier Netzwerke durch einen Leit-PC

Konfigurationsprogramm aus - einer MFC-Applikation. Dazu werden in jeweils einem Dokument die nötigen Informationen über ein Netzwerk gespeichert. Solch eine Anwendung gestattet es, mehrere Dokumente, wie in MS Word, gleichzeitig zu öffnen. Dieses flexible Design der MFC-Anwendung gestattet es, verschiedene CAN-Netzwerke zu administrieren, ohne die Anwendung selbst zu verändern. Fehlen Überlegungen dieser Art, läuft man Gefahr, für jede CAN-Applikation ein Programm zur Fernwartung zu entwickeln.

Somit ergeben sich folgende Ziele für die zu fertigende Software:

- Unterstützung verschiedener Mikroprozessoren
  - x86
  - C166
  - Hyperstone
- Unterstützung unterschiedlicher CAN-Controller
  - SJA 1000
  - PCI-Karte mit CAN-Controller
  - Simulierte CAN-Controller (TCP)
- Applikationen mit mehreren CAN-Instanzen
- Vorbereitung eines Gateway’s für TCP
- Hohe Performance
- Niedriger Speicherverbrauch

2 Der CAN-Bus

2.1 Einleitung in den CAN-Bus

Der von der Bosch AG entwickelte CAN-Bus diente ursprünglich nur zur Vermeidung von Kabelbäumen in Autos, sollte aber gleichzeitig die Kommunikation verschiedener Komponenten gewährleisten. Um sicherzustellen, dass weniger wichtige Komponenten, wie z.B. Scheibenwischer, die wichtigen, wie Bremsen, durch ihr Kommunikationsbestreben nicht funktionsunfähig machen , wurde in den CAN-Bus ein Prioritätensystem integriert, das hart-echtzeitfähig ist, d.h. Nachrichten werden definitiv in einem Zeitfenster übertragen. Da sich durch diesen Bus mit wenig Aufwand auch größere Anlagen steuern lassen, entwickelte sich der CAN-Bus zunehmend zum Industriestandard.

Beim CAN-Bus sind alle Stationen direkt mit dem Bus - einer Zweidrahtleitung - verbunden. Sollten weitere Teilnehmer im Netzwerk benötigt werden, schaltet man sie einfach parallel hinzu. Der CAN-Bus ist ein Multimasterbus, d.h. mehrere Geräte können eine Übertragung initiieren. Er unterstützt von der Buslänge abhängige programmierbare Bitraten (1Mbs bei 40m, 40Kbs bei 1000m) und besitzt umfangreiche Fehlererkennungsmechanismen wie CRC. Je nach Konfiguration können Nachrichten als Broadcast bzw. Multicast verschickt werden.

2.2 Schichten des CAN-Bus

In der CAN-Spezifikation sind folgende 3 Schichten definiert:

- Physikalische Schicht
- Transport Schicht
- Objekt Schicht

Die Physikalische Schicht ist identisch mit der Bitübertragungsschicht (physical layer) des ISO-OSI Referenzmodells. Diese ist verantwortlich für die Kodierung und Übertragung einzelner Bits. Beim CAN-Bus wird der Non-Return-To-Zero Code mit Bit-Stuffing verwendet. Das Bit-Stuffing dient zur besseren Synchronisation beim Senden von 5 gleichwertigen Bits durch Einfügen eines komplementäres Bits in den Bitstrom.

Die Transport-Schicht hat die Aufgabe, Übertragungsfehler zu erkennen, Bestätigungen (Acknowledgements) zu schicken, Arbitrierungen durchzuführen, Nachrichten in ein Frame zu packen und die richtige Bitrate zu setzen.

Die Objekt-Schicht beinhaltet das Filtern von Nachrichten. Es ist z.B. möglich, einen Filter für ID's zu setzen und so den Aufwand für die Bearbeitung von Nachrichten in Software zu minimieren (s. Kap. 2.4). Weiterhin hat diese Schicht die Aufgabe, Statusinformationen über Fehler und den aktuelle Systemzustand bereit zu stellen.

Die Transport-Schicht und die Objekt-Schicht entsprechen dabei nahezu der Sicherungsschicht (Data Link Layer) des ISO-OSI Referenzmodells. Ein CAN-Baustein wird als kompakter Baustein geliefert, bei dem nur der Zugriff auf die Objekt-Schicht möglich ist.

2.3 Übertragung von Nachrichten

Zur Übertragung von Daten erhält im Gegensatz zu herkömmlichen Netzwerken jede Nachricht eine ID. Diese ID ist 11 Bit breit, was einer Auflösung von $2^{11} = 2048$ Informationen entspricht. Es gibt keine Sende- bzw. Empfangsadressen. Jede Message besteht aus den in Abb. 6 gezeigten Elementen,

wobei die ID den ersten 11 Bit des Arbitrierungsfelds genügt. Das 12. Bit zeigt an, ob es sich um einen *Remote Request* handelt, d.h. es werden Informationen zu dieser ID angefordert. In dem Controlfeld ist angegeben, wie lang das Datenfeld ist. Hierfür sind 4 Bit reserviert. Allerdings ist das Datenfeld max. 8 Byte lang. Somit sind nur, die Werte 0-8 gültig. In dem CRC Feld ist der Rest der Polynomdivision des gesamten Bitstroms dieser Message (von SoF bis Datafield) mit dem Generatorpolynom $p(x) = X^{15} + X^{14} + X^{10} + X^8 + X^7 + X^4 + X^3 + 1$ eingetragen. Dieser Rest wird bei den Empfängern mit dem Ergebnis der erneuten Polynomdivision der empfangenen Daten vergleichen. Bei Ungleichheit wird die Nachricht verworfen (s. Kap. 2.5).

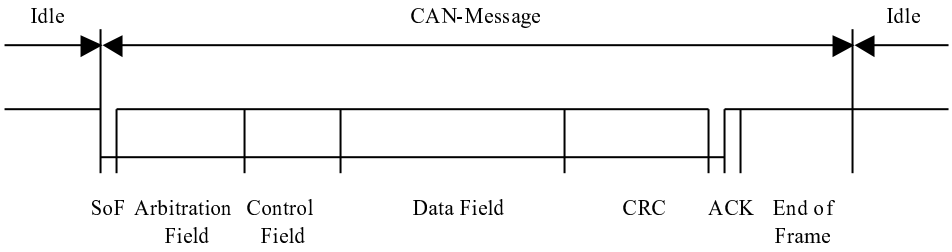

Abbildung 6: Aufbau eines CAN-Frames

Beim CAN-Bus gibt es 2 verschiedene Pegel, den rezessiven H-Pegel und den dominanten L-Pegel. Im Ruhezustand befindet sich der Bus im Idle-Modus (H-Pegel). Möchte eine Station senden, testet sie, ob der Bus im Idle-Modus ist und beginnt mit dem Senden des dominanten Bits Start_of_Frame(SoF), d.h. der Bus hat sofort L-Pegel. Dabei ist es egal, wieviele Stationen ein H-Signal einspeisen. Die anderen Stationen registrieren den L-Pegel auf dem Bus und dürfen nun nicht mehr senden, bis dieser Sendevorgang abgeschlossen ist. Im Anschluß an SoF wird der Identifier Bit für Bit übertragen. Gleichzeitig werden die Pegel vom Bus eingelesen und kontrolliert, ob wirklich der richtige Pegel anlag. War die Prüfung erfolgreich, wird weitergesendet, falls nicht, wird die Übertragung abgebrochen. Die ist dann der Fall, wenn eine weitere Station gleichzeitig mit dem Senden einer Nachricht begonnen hat. Solange alle Bits identisch sind, merkt keine der Stationen, dass in Wirklichkeit zwei senden. In dem Moment, wo sich die Bits im Arbitrierungsfeld unterscheiden, setzt sich automatisch das dominante Bit auf dem Bus durch. Die Station, die das rezessive Bit gesendet hat, unterbricht den Sendevorgang. Auf diese Weise kann durch geschickte Vergabe von ID's eine Priorisierung erreicht werden (Zerstörungsfreie bitweise Arbitrierung). Nach der Arbitrierung ist sichergestellt, dass nur eine Station sendet. Im Anschluß werden die übrigen Felder bis auf das ACK-Feld gesendet. Dieses wird bei fehlerhafter Übertragung von einer mithörenden Station auf den dominanten Pegel gezogen. Daran erkennt jede Station, dass diese Nachricht ungültig ist (s. Kap. 2.5).

Die dargestellte CAN-Message ist in der CAN-Spezifikation V2.0A[3] definiert. Desweiteren gibt es eine CAN-Spezifikation V2.0B, die sich im Wesentlichen durch einen 29Bit Identifier unterscheidet. Ein gleichzeitiger Betrieb beider Messagetypen auf einem Bus ist möglich. Dazu ist im Control-Field das Bit IDE definiert, das in einer 11Bit Message rezessiv und in einer 29Bit Message dominant gesetzt wird.

2.4 Filtern von Nachrichten

Alle CAN-Controller bieten zusätzlich die Möglichkeit, bereits in Hardware eine Vorfilterung von Nachrichten vorzunehmen. Der entscheidende Vorteil besteht in der Verhinderung einer Interruptauslösung durch Nachrichten, die nicht benötigt werden. Auf diese Weise wird verhindert, dass die Performance der gesamten auf dem Gerät laufenden Anwendungen unnötig verringert wird.

Die Konfiguration des Filtermechanismus erfolgt üblicherweise über zwei Register, in denen festgelegt wird, welche Bits des Identifiers relevant sind und welchen Wert diese Bits haben müssen.

Der verwendete CAN-Controller SJA1000 wird über die Register ACR (Acceptance Code Register) und AMR (Acceptance Mask Register) konfiguriert. Die oberen 8 Bits des Identifiers einer Nachricht werden jeweils mit dem korrespondierendem Bit des ACR-Registers XNOR-verknüft, d.h. sie werden auf Gleichheit getestet. Das Ergebnis dieser Operation muß für jedes Bit *wahr* sein, das im AMR als relevant markiert wurde. Nur wenn dies der Fall ist, wird die Nachricht in die Nachrichtenwarteschlange aufgenommen, ansonsten verworfen. Durch diesen Mechanismus kann ein Bereich von ID's zugelassen bzw. gesperrt werden (s. Tab. 1). In dem dargestellten Beispiel sind die Register ACR und ACM so gesetzt, dass alle Nachrichten im Bereich von 1984-2047 akzeptiert werden.

	Beispiel 1	Beispiel 2	Beispiel 3
Empfangener Identifier	11111001xxx	11111011xxx	10011011xxx
Wert des ACR-Register	11111000	11111000	11111000
$\overline{ID \otimes ACR}$	11111110	11111100	10011100
ACM	00000111	00000111	00000111
Relevante Bits	11111xxx	11111xxx	10011xxx
Nachricht akzeptiert	wahr	wahr	falsch

Tabelle 1: Beispiel für Filterung von Nachrichten

2.5 Fehlerbehandlung

Jeder CAN-Controller besitzt einen Sende- und einen Empfangsfehlerzähler. Im Normalfall befindet sich der Controller im Zustand *error active*. In diesem Zustand kann er aktiv am Busgeschehen teilnehmen, d.h. Senden und Empfangen. Tritt ein Fehler beim Empfang einer Nachricht auf, sendet der Controller ein *Active Error Flag* und erhöht seinen Empfangsfehlerzähler. Tritt ein Fehler beim Senden einer Nachricht auf, wird der Sendefehlerzähler erhöht. Überschreitet einer der Zähler die Grenze von 127, geht der Controller automatisch in den Zustand *error passive* über. Im Fehlerfall darf nur noch ein *Passive Error Frame* gesendet werden, wobei der entsprechende Zähler weiter erhöht wird. Bei einer erfolgreichen Übertragung wird der Sendefehlerzähler, falls ungleich 0, um eins erniedrigt.

Erreicht einer der beiden Zähler 255, geht der CAN-Controller automatisch in den *Bus-off* Zustand über. Er ist vollständig vom Bus abgekoppelt und kann üblicherweise nur noch per Software zurückgesetzt werden. Durch diesen Mechanismus wird verhindert, dass fehlerhafte Stationen den kompletten Bus lahm legen, indem sie z.B. ständig alle Messages als fehlerhaft markieren.

2.6 Einleitung in CANopen

Der CAN-Bus beinhaltet, wie im vorangegangenen Kapitel erwähnt, nur die Schichten 1 und 2 des ISO-OSI Modells. Nun stellt sich aber die Frage, wie z.B. über den CAN-Bus längere Datenblöcke ef-

fektiv übertragen werden können? Wie gestaltet sich ein effektives Netzwerkmanagement, z.B. Überwachung aller Stationen, Zeitsynchronisation, Fernwartung? Wie können dem Netzwerk dynamisch Geräte hinzugefügt bzw. entnommen werden?

Die CiA (Can in Automation) hat daraufhin den Standard CANopen[4] definiert, der die geforderten Eigenschaften unterstützt und echtzeitfähig ist. CANopen ist ein Schicht 7 Protokoll. Eine grobe Übersicht über das Referenz Modell zeigt Abb. 7.

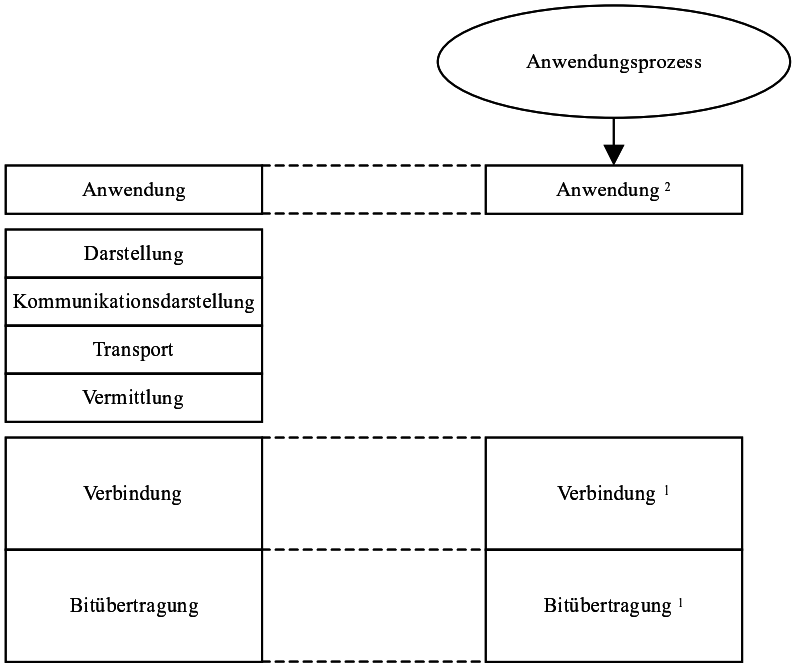

Abbildung 7: Vergleich ISO-OSI-Referenzmodell (links) mit CANopen Modell (rechts)

Die Schichten 3 bis 6 sind in CANopen nicht definiert, da eine Vielzahl von Aufgaben dieser Schichten nicht benötigt werden. Allerdings sind einzelnen Komponenten in der Applikationsschicht umgesetzt, wie z.B.:

- Segmentierung von Daten
- Fehlerkontrolle
- Zuweisung von Adressen

In CANopen ist die Schnittstelle der Anwendungsschicht zur Verbindungsschicht lediglich grundsätzlich beschrieben. Die API (application programable interface) zur eigentlichen Anwendung ist komplett implementationsabhängig.

[1] CAN-Spezifikation
[2] CANopen-Spezifikation

3 Implementierung der Software

3.1 Das Schichtenmodell

Die Umsetzung der in Kapitel 1.4 erwähnten Zielsetzungen erfordert eine Kapselung und Strukturierung der Aufgabe. Wichtig war vor allem, eine Software zur Verfügung zu stellen, die den Ansprüchen gerecht wird und ohne großen Aufwand erweiterbar ist. Aus diesem Grund wurden die einzelnen Funktionalitäten in Module gekapselt, sodass sie ohne Änderungen ausgetauscht werden können. Die einmal entwickelte Applikation läuft ohne Probleme und ohne Änderungen mit einem anderen CAN-Controller zusammen. Es müssen nur die beiden unteren Treiber-Schichten ausgetauscht werden. Alle direkten Hardwarezugriffe auf den CAN-Controller befinden sich im CAN-Treibermodul. Ein direkter Zugriff auf den CAN-Bus von der CANopen-Schicht oder sogar von der Applikationsschicht aus ist nicht möglich. Daraus ergibt sich der Vorteil, dass die CANopen-Schicht komplett hardwareunabhängig ist und der größte Teil der darauf aufsetzenden Applikationen ebenfalls.

Die CAN-Schicht ist ohne weiteres auch für eine 'normale' CAN-Umgebung nutzbar. Sie ist unabhängig von der CANopen-Schicht. Um diese Eigenschaften zu ermöglichen, war es nötig eine standardisierte API zu entwickeln, die die einzelnen Schichten voneinander trennt. Diese API bildet die Voraussetzung zur Erstellung einer einheitlichen CANopen-Schicht für Mikrocontroller und PCs.

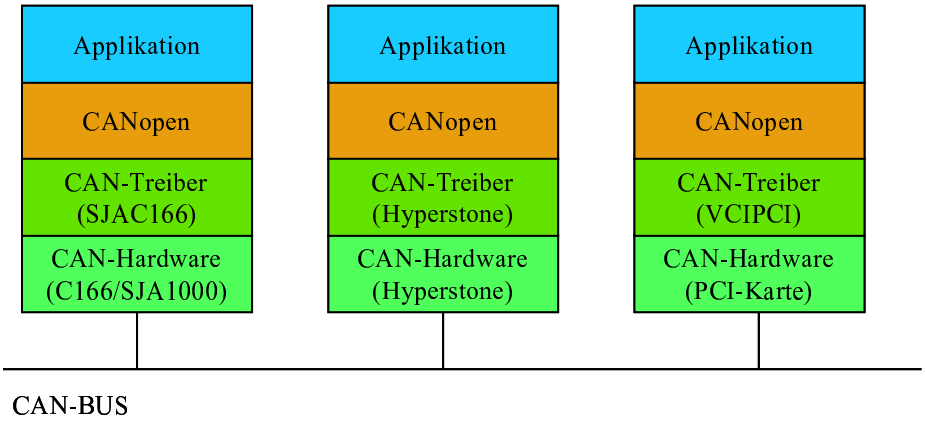

Abbildung 8: Schichtenmodell der Software

3.2 Der CAN-Treiber

Der CAN-Treiber ist ein Softwaremodul, welches die direkte Hardwareansteuerung des CAN-Controllers bewerkstelligt. Der CAN-Treiber entspricht einem virtuellen CAN-Controller. Dies ist der Tatsache geschuldet, dass verschiedene CAN-Controller zu unterstützen sind und die Programmierung stark voneinander abweicht. Einige CAN-Controller werden nur über in den Speicher gemappte Register angesprochen, andere mittels proprietärer API's, wie dem Virtual-CAN-Interface (VCI)[2]. Eine ursprünglich geplante Anpassung auf diese API wurde aus konzeptionellen Gründen, wie etwa unbrauchbarem Fehlermanagement, fehlender Trennung von Applikation und Hardwareeinstellungen sowie fehlender Unterstützung des Hyperstone, verworfen. Eine Alternative wäre auch der direkte Erwerb einer CANopen-Library von IXXAT gewesen, nur läuft diese nicht auf dem Hyperstone und

setzt pikanterweise nicht auf dem firmeneigenen VCI-Standard auf, sodass selbst eine Umsetzung der VCI-API auf dem Hyperstone sinnlos ist.

Die Adressierung der CAN-Hardware erfolgt über die Kapselung in einem *CAN_HANDLE*, einer Struktur, in der Informationen über die zu benutzende Hardware gespeichert sind (s. Abb. 9). Handles sind erforderlich, um mehrere Instanzen der CAN-Schicht voneinander zu trennen, z.B. bei Applikationen, die mehrere CAN-Controller bedienen. Die Unterscheidung, wie welcher CAN-Baustein benutzt wird, fällt in der CAN-Treiberschicht.

```
//////////////////////////////////////////////////////   ////////////////////
// CAN_HANDLE declaration. read only by application !!!
typedef struct
{
       CAN_MESSAG E          receivequeue[MAXMESSAGES];
       BYTE                  readindex;
       BYTE                  writeindex;
       void*                 userdata;     // can be used by application,
                                           // CANopen  uses this ptr to store
                                           // CANOPEN_HANDLE
       //unsigned char       can_mode;     // 0=11Bit Identifier (FUTURE use)
                                           // 1=29Bit  Identifier (FUTURE use)
} CAN_HANDLE;
```

Abbildung 9: Deklaration des CAN_HANDLE

Das CAN_HANDLE wurde in zwei Bereiche gegliedert, ein Basishandle und ein abgeleitetes Handle. Das Basishandle dient in erster Linie zum Speichern der Nachrichten in Warteschlangen und zum Zugriff auf die Callbacks.

Der erste Eintrag *CAN_MESSAGE* ist bewußt am Anfang platziert, da dieses Element das meist frequentierte ist und somit eine potenzielle Addition des Offsets entfällt. Die Nachrichtenwarteschlange beinhaltet alle empfangenen Nachrichten, die noch nicht durch die höheren Schichten abgearbeitet wurden. Sie ist als Ringpuffer konzipiert, d.h. alle neuen Nachrichten werden nacheinander einsortiert - der Write-Index erhöht sich jeweils um 1. Wird von einer höheren Schicht eine Message gelesen, erhöht sich der Read-Index und die Message wird freigegeben. Auf diese Weise können neue Nachrichten auch am Anfang des Array's einsortiert werden, bis der Read-Index erreicht wird. Das Einsortieren und Auslesen der Messagequeue aus den verschiedenen Prioritätsebenen bzw. Threads wird durch Mutex-Funktionen geschützt.

Der void-Pointer *userdata* ist die einzige Variable, auf die die höheren Schichten Zugriff haben. Hier können sie Daten speichern, meistens einen Pointer auf eine eigene Handle-Struktur. Sinnvoll ist dies in Zusammenarbeit mit Callbacks, die aufgerufen werden, wenn eine Nachricht empfangen wurde. Die Callbacks erhalten als Argumente einen Zeiger auf das CAN-Handle und auf die Message. Durch Zugriff auf userdata und geeignetes Casting kann der gespeicherte Zeiger entsprechend umgewandelt werden. Der Vorteil dieser Variante ist, dass durch Austausch der oberen Schichten, z.B. Nutzung von DeviceNET, keine Änderung der CAN-Schicht notwendig ist.

Die konkrete Umsetzung der Hardwarezugriffe erfolgt mittels des abgeleiteten CAN_HANDLE's, z.B dem *VCI_CAN_HANDLE*. Alle abgeleiteten CAN_HANDLE's besitzen als ersten Eintrag in ihrer Struktur eine Instanz der Basisklasse. An die oberen Schichten der Applikation wird genau dieser

Pointer, allerdings als Zeiger vom Typ CAN_HANDLE, übergeben. Der Vorteil besteht darin, dass bei einem Wechsel des CAN-Controllers, das Casting in den oberen Schichten nicht verändert werden muss. Die Änderungen beziehen sich lediglich auf die CAN-Schicht.

```
//////////////////////////////////////////////////////////////////////////////
// VCI-CAN-Handle declaration
typedef struct
{
       CAN_HANDLE                  can_handle;

       // VCI related initializations
       UINT16                      boardhandle;
       UINT16                      transmit_queue_handle;
       UINT16                      receive_queue_handle;
       UINT8                       cannumber;          // Number of used can -
                                                       // controller on bo ard
       CAN_Receivecallback         callback;
       BYTE                        CanQueueLock;       // 0=not locked,
                                                       // 1=locked
} VCI_CAN_HANDLE;
```

Abbildung 10: Deklaration des VCI_CAN_HANDLE

Die Einführung von Handles bedeutet eine geringere Performance und u.U. einen höheren Speicherverbrauch. Dieser ist meistens vernachlässigbar, da z.B. bei Verwendung des SJA1000 Controllers nur ein zusätzlicher Eintrag im *SJA_CAN_HANDLE* gegenüber dem CAN_HANDLE enthalten ist. Alle Einträge in den beiden Strukturen hätten auch auf herkömmlichen Weise, z.B. durch Verwendung statischer Variablen, gespeichert werden müssen, aber durch Handles erreicht man eine deutlich höhere Flexibilität.

In diesem Zusammenhang von Basis-Handles und abgeleiteten Handles zu sprechen ist nur bedingt korrekt, da es sich bei diesen Bezeichnungen um Konzepte aus der objekt-orientierten Programmierung handelt. Da sich der Programmkode aber streng genommen bei einer Umsetzung in C++ ähnlich kodieren würde, werden diese Begriffe hier benutzt. Als Beispiel sei erwähnt, dass in C++ der *this*-Zeiger immer das erste Argument einer Memberfunktion ist, der wie in Abb. 11 dargestellt ist, hier nachempfunden wurde. Der Umsetzung der Software in C++ stand lediglich der fehlende C++-Compiler für die Mikrocontrollersysteme entgegen. Die Funktionen der CAN-Schicht bilden implizit die Memberfunktionen einer Klasse *CAN* . Zu den höheren Schichten stellt jeder CAN-Treiber die in Abb. 11 gezeigten API-Funktionen zur Verfügung.

```
OPTOERROR      CAN_InitHardware(CAN_HANDLE **handle, CAN_Configcallback configcallback );
OPTOERROR      CAN_FreeHardware(CAN_HANDLE **handle);
OPTOERROR      CAN_InitCan(CAN_HANDLE *handle);
OPTOERROR      CAN_TransmitData(CAN_HANDLE *handle, CAN_MESSAGE *message);
OPTOERROR      CAN_SetReceiveInterrupt( CAN_HANDLE *handle, CAN_Receivecallback callback);
OPTOERROR      CAN_WaitMsgTimeout(CAN_HANDLE *h   andle, long milliseconds);
OPTOERROR      CAN_SetFilterMask(CAN_HANDLE *handle, CAN_ID AC_CODE, CAN_ID AC_MASK);
OPTOERROR      CAN_Baudrate(CAN_HANDLE *handle, WORD timing);
```

Abbildung 11: Deklaration der Funktionen der CAN-API

Die Funktion CAN_TransmitData() ist eine Funktion der CAN-API, die von höheren Schichten aufgerufen werden kann. Nun stellt sich das Problem, wie in einer Funktion der CAN-Schicht ohne den Mechanismus der virtuellen Funktionen von C++ eine möglichst effiziente Verzweigung erreicht werden kann, wenn 2 verschiedene CAN-Controller benutzt werden. Der einfachste Fall wäre eine Switch-Case Anweisung, die allerdings absolut unflexibel ist. Mit jedem neuen CAN-Treiber müßte dieses Konstrukt erweitert werden, was zu einer Neucompilierung aller Treiber auf jeder Plattform führt. Da Treiber nur auf der für sie angepassten Hardware lauffähig sind, ist eine erfolgreiche Compilierung mehr als fraglich. Im Anschluß müssten alle Treiber gelinkt werden. Das Programm würde immer einen übermäßigen Overhead enthalten, der nur Performance bzw. Speicher verbraucht und vor allem in Mikrocontrollersystemen zum Problem führen kann.

C++ bietet die Möglichkeit, virtuelle Funktionen zu definieren, was den Vorteil bringt, eine Funktion in der Basisklasse zu deklarieren und in der abgeleiteten Klasse zu definieren, d.h. eine Funktion wie CAN_TransmitData() wird erst im konkreten Hardwaremodul definiert und direkt angepaßt. Nur der Aufruf von außen ist derselbe. In jeder Applikation wären nur die Treiber enthalten, die wirklich gebraucht werden, denn C linkt nur Funktionen, die wirklich benutzt werden.

Von diesem Gedanken ausgehend wurde versucht, einen Mechanismus in C zu etablieren, der eine möglichst effiziente Umsetzung garantiert. In den meisten Anwendungsfällen wird nur ein CAN-Treiber benötigt. Die Funktionen werden über *#define* Anweisungen direkt auf die Hardwaretreiber umgemapped (s. Abb. 12). Nach außen betrachtet, bleibt die erwähnte API dieselbe. Sollte in Zukunft eine Applikation erstellt werden müssen, die mehrere CAN-Controller anspricht, muß ein virtueller Mechanismus implementiert werden. Dies könnte relativ einfach erfolgen, indem die #define Anweisungen entfallen und das CAN_HANDLE um Zeiger zu den Funktionen erweitert wird. Diese Zeiger werden bei der Initialisierung des CAN_HANDLE's im Hardwaretreiber gesetzt. Zusätzlich werden den Modulen *Can.h/Can.c* Funktionen, wie CAN_TransmitData, hinzugefügt, die diesen Zeiger lesen und die entsprechend in die Hardwarefunktion springen.

```
#define CAN_InitHardware            VCI_CAN_InitHardware
#define CAN_FreeHardware            VCI_CAN_FreeHardware
#define CAN_InitCan                 VCI_CAN_InitCan
#define CAN_TransmitData            VCI_CAN_TransmitData
#define CAN_SetReceiveInterrupt     VCI_CAN_SetReceiveInterrupt
#define CAN_WaitMsgTimeout          VCI_CAN_WaitMsgTimeout
#define CAN_SetFilterMask           VCI_CAN_SetFilterMask
#define CAN_SetBaudrate             VCI_CAN_ SetBaudrate
```

Abbildung 12: Mapping der VCI-Funktionen innerhalb der CAN-API

3.3 Das CAN-Environment

Das CANEnvironment ist ein Modul, in dem die Hardwaretreiber ausgewählt und die konkreten Hardwareeinstellungen vorgenommen werden, wie z.B. Nummer der Interruptleitung. Diese Angabe kann für denselben CAN-Treiber bei jeder Kamera verschieden sein. Die Auswahl der CAN-Controller erfolgt durch Übergabe einer Callback-Funktion beim Aufruf von CAN_InitHardware(). Die Callback-Funktionen sind in dem Modul CANEnvironment.c definiert ist. Die Kenntnis dieser Funktion entspricht prinzipiell dem Namen einer Resource, wie z.B. COM1 bzw. LPT1, die man unter Windows öffnet.

In dieser Callback-Funktion werden die grundlegenden Einstellungen, die für die Initialisierung der CAN-Hardware erforderlich sind, vorgenommen, Die Nutzung der PCI-Karte von IXXAT setzt die Kenntnis von Board-Typ, Board-Adr und Board-IRQ voraus, um die Initialisierung der VCI-API korrekt durchzuführen. Es handelt sich im Prinzip um Administratoreinstellungen, wie sie in einer Window-Systemsteuerung bzw. einer üblichen Treiberinstallation vorgenommen werden.

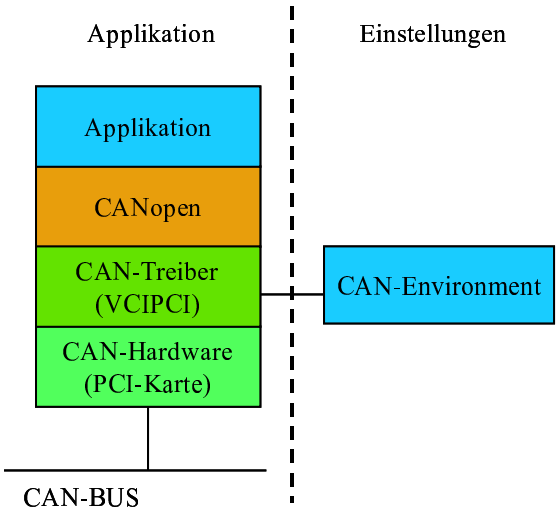

Abbildung 13: Trennung von Applikation und Hardwareeinstellungen

Die Applikation sollte von diesen Einstellungen unberührt bleiben, da sie mit anderen CAN-Controllern ohne Änderungen lauffähig sein soll (s. Abb. 13). Mit der hier vorgestellten Lösung ist bei einem Wechsel des CAN-Controllers lediglich das Modul CANEnvironment auszutauschen.

3.4 Der SJAC166-Treiber

Der SJAC166 Treiber ist der Standardtreiber für die autonomen Kameras. Bei der Erstellung dieses Treibers wurde ganz besonders auf den sparsamen Umgang und die Effizienz des Programmcodes geachtet.

Die Kameras besitzen ein Mikrocontrollermodul mit C166 und einem CAN-Controllermodul mit Philips SJA1000 CAN-Controller. Der Zugriff auf den CAN-Controller erfolgt über 32 Register, die in den Speicherbereich des Mikrocontrollers ab Adresse 0x600000 eingeblendet werden. Diese hohe Adresse erfordert beim C166 *huge*-Ptr. Es handelt sich um einen 32Bit Zeiger, der sich aus einem 16Bit Segmentzeiger und einem 16Bit Zeiger im Segment zusammensetzt. Ein Zugriff auf diesen Speicherbereich dauert verhältnismäßig lange.

Der ursprüngliche Ansatz, mehrere CAN-Controller gleichzeitig zu betreiben, wurde bei diesem Treiber nicht verfolgt, denn das hätte bedeutet, die Startadresse des CAN-Controllers im SJA_CAN_HANDLE zu speichern, sodass bei jedem Hardwarezugriff ein Speicherzugriff zur Ermittlung der Basisadresse und eine Addition des Offsets durchgeführt werden müßte. Falls dies trotzdem erforderlich sein sollte, ist eine Erweiterung ohne weiteres möglich. Sollte sich nur die Startadresse des CAN-Controllers ändern, ist lediglich eine #define Anweisung im Modul CANEnvironment (s. Kap. 3.3) zu korrigieren.

3.5 Der VCIPCI-Treiber

Der VCIPCI-Treiber ist ein CAN-Treiber für die PCI-Karte von IXXAT. Durch Nutzung dieser Karte ist es möglich, ein Konfigurationsprogramm unter Windows zu entwickeln, welches das komplette Netzwerk überwacht bzw. die nötigen Einstellungen für die korrekte Funktionsfähigkeit der einzelnen Geräte vornimmt.

Der Treiber baut auf die VCI-Bibliothek des Herstellers auf, die zur Zeit nur unter Windowssystemen lauffähig ist. Er ist in der Lage, mehrere Instanzen zu erstellen, sobald mehrere CAN-Karten im Rechner verwendet werden.

3.6 Der TCP-Treiber

Der TCP-Treiber ist ein CAN-Treiber, der einen CAN-Bus über ein TCP-Netzwerk simuliert. Die Entwicklung war erforderlich, da über längere Zeiträume keine CAN-Boards für die Kamera zur Verfügung standen. Da die API zu den höheren Schichten konstant ist, war es möglich, bereits Teile des CANopen Standards umzusetzen und die Funktionsweise durch eine Simulation nachzuweisen. Ein weiterer Vorteil ist die wesentlich einfachere Entwicklung mit herkömmlichen Compilern und Debuggern, die unter Windowssystemen verfügbar sind. Aufwendige Downloads des Programms auf den Mikrocontroller entfallen. Alle Clients können gleichzeitig debuggt werden. Es ergibt sich eine deutlich schnellere Implementierung des kompletten Standards.

Dieser Treiber ermöglicht es, die eingangs erwähnte Gateway-Funktion (s. Abb. 4) in einem Leit-PC zu implementieren. Hierzu ist es notwendig, in der entsprechenden Applikation eine CAN-Bus Instanz (PCI-Karte) und eine CANTCP-Instanz zu öffnen. Diese Applikation braucht nur alle Nachrichten, die auf dem CAN-Bus verschickt werden, per TCP an den Remote-PC weiterzuschicken - ideal für eine Fernwartung vom Büro aus. Für den entfernten PC ist dieses Vorgehen völlig transparent. Er öffnet statt des üblichen VCIPCI-Treibers den CANTCP-Treiber und stellt nicht fest, wie weit das eigentliche Netzwerk entfernt ist.

3.7 Die CAN-Message

Da es sich bei der CAN-Treiberschicht um einen virtuellen CAN-Controller handelt, ist es erforderlich, die CAN-Message auch virtuell zu definieren. Die CAN_Message Struktur (s. Abb. 14) beinhaltet alle wichtige Informationen, die nötig sind, um eine Nachricht über den Bus zu senden. Es handelt sich im Prinzip um ein Abbild der nötigen Informationen aus der in Abb. 6 dargestellten CAN-Nachricht.

```
////////////////////////////////////////////////////////////////////////////
// CAN_MESSAGE Declaration (for all CAN  -Controllers)
typedef struct
{
       CAN_ID         id;
       UINT8          data[MAXDATABYTES];  // Array for up to 8 data(UINT8) bytes
       UINT8          data_length:4;       // number of data bytes (0  -8)
       UINT8          remoteframe:1;       // RTR-Bit: 0=Dataframe, 1=Remoteframe
       UINT8          reserved:3;          // not used
       UINT8          status;
       //Future u se
       //UINT32       time_stamp;          // Timestamp for receive queue objects
} CAN_MESSAGE;
```

Abbildung 14: Format der eigenen CAN-Message

Der erste Eintrag bezeichnet den CAN-Identifier. Hierzu wurde ein eigener Datentyp definiert, um eine eventuell spätere Anpassung an den CAN-Standard V2.0b ohne größere Schwierigkeiten zu ermöglichen. Der Identifier beim CAN-Standard V2.0a ist 11Bit breit. Somit ist die CAN_ID als *short* definert. Um Speicherplatz zu sparen, wurden mehrere Attribute (data_length, remoteframe, reserved) zu einem Byte zusammengefaßt. Dadurch wurde gleichzeitig erreicht, dass die Länge der Struktur lediglich 12 Bytes beträgt und in Array's, z.B. der Messagequeue, immer auf geraden Adressen liegt.

3.8 Empfang von Nachrichten

Der virtuelle CAN-Controller erfordert weitere Überlegungen zum Empfangen und Verarbeiten von Nachrichten. Die verfügbaren CAN-Controller besitzen unterschiedliche Möglichkeiten zum Zwischenspeichern von Nachrichten. Vor allem die unterschiedliche Tiefe von Empfangswarteschlangen ist problematisch. Aus diesem Grund wurde eine eigene Empfangswarteschlange für Nachrichten eingerichtet.

Das Behandeln der Nachrichten ist modernen Betriebssystemen nachempfunden. Bei Eintreffen einer Nachricht wird ein Interrupt ausgelöst. Die Nachricht wird dann sofort in die Warteschlange einsortiert. Danach wird der Interrupt verlassen. Der Vorteil dieser Variante ist, dass der CAN-Controller sofort wieder bereit ist, neue Nachrichten aufzunehmen und das aktuelle Programm nur für eine äußerst kurze Zeit unterbrochen wird. Auf diese Weise kann ein aufwendiger Datentransfer mit Hand-Shake Verfahren, wie er in CANopen üblich ist, besser in das Prioritätensystem von Betriebssystemen eingepaßt werden. Ein weiteres Problem bei den Hand-Shake Verfahren ist, dass sie auf Antwortnachrichten warten müssen, was im laufendem Interrupt zum Deadlock führt.

Der Einsatz der Nachrichtenwarteschlangen macht es erforderlich, Funktionen für den gegensei-

tigen Ausschlusses zu integrieren. So ist grundsätzlich zu beachten, dass das Einsortieren der Nachrichten im Interrupt und das gleichzeitige Auslesen von Nachrichten im unterbrochenen Programmteil zu unterbinden ist. Die Implementation ist treiberabhängig, da sich die Tasks, die auf die Nachrichtenwarteschlangen zugreifen, in Bezug auf ihre Priorität betrachtet, nicht immer gleich verhalten. In Mikrocontrollersystemen wird das Einsortieren vorwiegend über einen höherpriorisierten Interrupt vorgenommen, d.h. der Interrupt kann nicht unterbrochen werden. In Multitasktingbetriebssystemen erfolgt die Einsortierung teilweise über gleichpriorisierte Threads (Vergleich Sockets unter Windows). Dort können sich die Tasks gegenseitig unterbrechen. Die API sollte aber nach außen hin konstant sein. Aus diesem Grund wurden 4 Mutex-Funktionen eingeführt. Die Funktionen CAN_Mutex() und CAN_UnMutex() sind für die Tasks vorgesehen, die Nachrichten aus der Warteschlange auslesen und während dieses Zeitraums entweder den Interrupt sperren oder eine Mutex-Variable setzen. Die Methoden CAN_MutexInterrupt() und CAN_UnMutexInterrupt() werden nur in den Programmteilen aufgerufen, die Nachrichten in die Warteschlange einsortieren. Erfolgt dies im Interrupt, ist keine weitere Ausformulierung der Funktionen nötig, da er von konkurrierenden Tasks nicht unterbrochen werden kann. Handelt es sich um einen Thread innerhalb einer Applikation, ist das Verwenden einer Mutex-Variablen nötig.

```
void   CAN_Mutex(CAN_HANDLE *handle);
void   CAN_UnMutex(CAN_HANDLE *handle);
void   CAN_MutexInterrupt(CAN_HANDLE *handle);
void   CAN_UnMutexInterrupt(CAN_HANDLE *handle);
```

Abbildung 15: Deklaration der Mutex-Funktionen

4 Konfiguration der Kamera

Das Konfigurieren der Kameras erfolgt z.Z. mittels lesbarer ASCII-Befehle inkl. variabler Parameter (s. Tab. 2). Die Befehle werden über die serielle Schnittstelle direkt an das entsprechende Zielgerät übertragen - eine in der Industrie gängige Vorgehensweise. Der Vorteil liegt in der einfachen Überwachung und Konfiguration per Terminal durch den Menschen, da er sich jederzeit die Befehle ansehen kann. Dies führt aber zwangsläufig zu Performanceverlust bei der Interpretation der Befehle, abgesehen von den dadurch entstehenden Fehlerquellen. Der vorhandene Overhead durch die Redundanz der Verwendung von lesbaren Zeichen verringert die effektive Datenrate erheblich. Die Zuordnung einer Nummer pro Befehl würde einen ASCII-Befehl von jetzt 3 Bytes auf max. 1 Byte verkürzen. Eine weitere unangenehme Eigenschaft dieser Konfigurationsmethode ist die Belegung einer seriellen Schnittstelle pro Kamera, da es sich bei serieller Übertragung um eine Peer-to-Peer Verbindung und nicht um einen Bus handelt.

Befehl	Parameter	Beschreibung
AVT	-	Automatische Schwellenberechnung einschalten
GVT	-	Pixelabhängige Schwelle(n) einlesen
ODI	<Output>	Definition der Fehlerimpulsausgabe
GRS	-	Ergebnisanforderung (Get Result)
FES	<Width>	Max. zul. Breite in Videosignal definieren
SAL	<Mittlungszahl>	Setzen der Scananzahl und des Toleranzpegels
	<oberer Offset>	
	<unterer Offset>	
SEC	<Output> <Count>	Max. zul. Anzahl von Events setzen

Tabelle 2: Beispielbefehle zur Konfiguration einer Kamera

Eine Teilaufgabe dieser Studienarbeit bestand u.a. darin, die Konfiguration der Kameras über das CAN-Netzwerk zu ermöglichen. Als Beispiel sei hier das Setzen der Scananzahl und des Toleranzpegels erwähnt (Befehl: SAL). Eine Zeichenkette für die serielle Übertragung sieht demnach folgendermaßen aus: „SAL 128 1000 7000“. Die max. Datenlänge in einem CAN-Frame beträgt allerdings 8 Bytes. Auch wenn auf die redundante ASCII-Darstellung verzichtet wird, ist es nicht möglich, diesen Befehl in 8 Bytes zu kodieren, da alleine die beiden Offsets schon long-Werte (je 4 Byte) sind. Eine Segmentierung der zu übertragenden Zeichenketten ist erforderlich, da sogar das Senden der C166-Programme als Übertragung einer einzelnen Zeichenkette realisiert wird.

Die Übertragung von Daten in mehreren Segmenten erfordert Protokolle zur Fehlererkennung und Korrektur und genau dieses Problem löst CANopen, das in einer aufbauenden Studienarbeit implementiert wird. Es macht daher keinerlei Sinn, ein proprietäres Protokoll zur Konfiguration der Kameras zu entwerfen. Aus diesem Grund wurde in Abstimmung mit Herrn. Dr. Richter auf die Erledigung dieser Aufgabe verzichtet.

5 Zusammenfassung

Die konsequente Trennung von Applikations- und Hardwareebene hat die Vorausetzungen für eine modulare CAN-API geschaffen. Diese API ist in der Lage, die verschiedenen Protokolle, wie z.B. CANopen und DeviceNET, die auf den CAN-Bus aufsetzen, zu unterstützen. Durch die einheitliche API wurde erreicht, dass diese Protokolle plattformunabhängig umgesetzt werden können und ohne Probleme auf andere Betriebssysteme und Prozessoren portierbar sind.

Die Kapselung von Applikation und Administration der Hardwareplattform macht es möglich, einen Wechsel des CAN-Controllers vorzunehmen, ohne das eigentliche Programm zu verändern. Es ist lediglich eine Neucompilierung unter Einbindung des neuen Hardwaretreibers erforderlich. Dies ist aus Gründen der Performance und des Speicherverbrauchs in eingebetteten Systemen geboten.

Diese Studienarbeit war die Voraussetzung für die Vernetzung der Kameras der Firma OPTOLOGIC. Die Perspektive liegt in der Erstellung einer plattformunabhängigen Implementierung des CANopen Protokolls.

Literatur

[1] Philips Semiconductors: *SJA1000 Stand-alone CAN controller*,
Preliminary specification, 1997 Nov 04

[2] IXXAT: *Virtual CAN Interface - Benutzerhandbuch*,
V1.09, 05/1998

[3] BOSCH: *CAN Specification*,
V2.0, 1991

[4] CAN in Automation e.V.: *CANopen Application Layer and Communication Profile*,
CiA Draft Standard 301 V4.0, 16.06.1999

[5] Wolfhard Lawrenz: *CAN, Controller Area Network - Grundlagen und Praxis*,
Hüttig Verlag 1997, 2. Auflage, ISBN: 3-7785-2263-9

[6] Konrad Etschberger: *CAN - Grundlagen, Protokolle, Bausteine und Anwendungen*,
Carl-Hanser Verlag München,Wien 1994, ISBN: 3-446-17596-2

[7] Ole Reinartz & Olaf Franke: *Entwicklung einer CAN-Bus Applikation*,
‘Universität Rostock, Institut für Automatisierungstechnik‘

[8] SIEMENS: *C165/C163 16 Bit CMOS Single-Chip Microcontrollers - User Manual*,
V2.0, 06/1996

[9] KEIL Software: *Optimizing 166/167 C Compiler and Library Reference*,
03/1998

[10] KEIL Software: *166/167 Assembler and Utilities*,
07/1996

Erklärung

Ich erkläre, diese Arbeit selbständig angefertigt und die benutzten Unterlagen vollständig angegeben zu haben.

Ort, Datum: Rostock, 21. Juni 2001