Technical Report

Pattern Recognition and Image Processing Group
Institute of Automation
Vienna University of Technology
Treitlstr. 3/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-8161
Fax: +43 (1) 5054668
E-mail: melzer@prip.tuwien.ac.at

PRIP-TR-49 27. Januar 1998

TROL
TU Vienna Robot Control Library

Thomas Melzer

Abstract

TROL is a C-function library which gives the programmer access to virtually all kinematic functions of a CRS A465 industrial robot by providing him with a comfortable, high level Applications Programmers Interface. This document describes how to install, use and extend the library. Furthermore, it gives detailed insight in the design and implementation of TROL.

TROL

TU Vienna **Ro**bot Control **L**ibrary

Version 2.1

Thomas Melzer
Wien, Jänner 1998

Einführung

Seit einiger Zeit verfügt das PRIP über einen Roboter der Firma CRS vom Typ A465. Die vom Hersteller unterstützten Möglichkeiten zur Programmierung dieses Roboters sind jedoch für viele Anwendungen nicht ausreichend. Aus diesem Grund wurde die "C"-Bibliothek **TROL** entwickelt, welche es dem Anwendungsprogrammierer erlaubt, den Roboter bequem aus seiner - auf dem PC laufenden - Applikation heraus anzusteuern.

Die Schnittstellen zum Robotersystem

Alle Anforderungen an den A465 laufen über den C500 Robotercontroller, dessen Herzstück eine Intel 286 CPU ist. Er verfügt unter anderem über 2 serielle Kommunikationskanäle, an denen das sogenannte Teach-Pendant und ein PC angeschlossen sind. Beide Geräte werden vom Controller normalerweise wie serielle Terminals behandelt; befindet sich der Controller allerdings im **ACI**-Modus (**A**dvanced **C**ommunication **I**nterface), so wird für die Kommunikation mit dem PC das proprietäre ACI-Protokoll verwendet, das vor allem für die schnelle Übertragung großer Datenmengen (z.B. Downloaden von RAPL-Programmen in den Controller) ausgelegt ist.

Programmierung des Roboters

Die Programmierung des Roboters erfolgt normalerweise in **RAPL** (**R**obotic **A**utomation **P**rogramming **L**anguage), einer BASIC-ähnlichen Interpretersprache. Es können entweder interaktiv einzelne RAPL-Kommandos abgesetzt - der PC fungiert hierbei als Terminal - oder vollständige RAPL-Programme in den Controller geladen und anschließend ausgeführt werden. Die eigentlichen Systemroutinen, die Aktionen des Roboters auslösen und kontrollieren, werden auf dem Controller - ähnlich wie DOS und BIOS Systemroutinen auf dem PC - über einen Softwareinterrupt aufgerufen; der RAPL-Interpreter z.B. setzt die meisten RAPL-Kommandos direkt in Aufrufe dieses **RAPL-BIOS** um.

Es besteht nun die Möglichkeit, mit einem gewöhnlichen Assembler oder C-Compiler erzeugten 8086-Code in den Programm-Buffer des Roboter-Controllers zu laden und dort auszuführen; diese **PCP**s (**P**rocess **C**ontrol **P**rogram) genannten Programme können sich - so wie der Interpreter - ebenfalls des RAPL-BIOS bedienen und somit prinzipiell jede gewünschte Aktion des Roboters veranlassen. Für die Implementierung komplexerer Anwendungen, insbesondere aus dem Bereich der Bildverarbeitung, sind PCPs jedoch aus folgenden Gründen nicht geeignet:

- Die Größe eines PCP (Code + Daten) darf 64K nicht übersteigen.

- Spezialhardware - wie z.B. das Matrox-Board oder ein LAN-Adapter - kann nicht verwendet werden.

- Die Controller-CPU ist ein 286, PCPs dürfen nur - nicht registeroptimierten!- 8086-Code enthalten. Die unter diesen Voraussetzungen erzielbare Ausführungsgeschwindigkeit ist für viele Anwendungen nicht ausreichend.

Es ist daher sinnvoll, die eigentliche Applikation auf dem PC zu implementieren und die vom Roboter auszuführenden Aktionen in kodierter Form über die serielle Schnittstelle einem PCP zu übermitteln, welches dann die entsprechenden Aufrufe des RAPL-BIOS durchführt.

TROL

TROL ist ein geschichtetes Kommunikationsprotokoll (siehe Kapitel 1) für den Datenaustausch zwischen PC und CRS-Roboter-Controller. Auf dem PC bietet es dem Anwendungsprogrammierer unter DOS oder Windows ein komfortables High-Level **API**[1] mit umfassenden Möglichkeiten zur Steuerung des Roboters. Controllerseitig ist TROL als PCP implementiert und stellt dort außer den Kommunikationsmechanismen noch eine "Serverapplikation" zur Verfügung, die Anforderungen an den Roboter entgegenimmt und an das RAPL-BIOS weiterleitet; für den Anwendungsentwickler besteht somit keine Notwendigkeit, auf PCP-Ebene zu programmieren.

Der Aufruf einer TROL API-Funktion führt zu folgenden Aktionen:

1. Es werden Plausibilitätsüberprüfungen durchgeführt, anschließend werden Funktionscodes und Parameter über die serielle Schnittstelle zum Controller übertragen.

2. Controllerseitig wird das Datenpaket von einem PCP entgegengenommen und dekodiert, ggf. werden auch hier Plausibilitäts- und Sicherheitsüberprüfungen durchgeführt; anschließend wird der entsprechende Aufruf des RAPL-BIOS durchgeführt und der BIOS-Returncode zurück an den PC übermittelt.

3. PC-seitig wird der Returncode entgegengenommen und an die aufrufende Funktion weitergeben.

---

[1] API (**A**pplication **P**rogram **I**nterface): bezeichnet allgemein eine Programmierschnittstelle.

API-Funktionen verhalten sich in dem Sinne synchron, daß sie erst dann wieder zum Aufrufer zurückkehren, wenn der Aufruf der entsprechenden Funktion des RAPL-BIOS erfolgt ist. Dies muß allerdings nicht bedeuten, daß die geforderte Aktion des Roboters (z..B. Bewegung) zu diesem Zeitpunkt bereits abgeschlossen ist!

Treten während der Bearbeitung einer API-Funktion Fehler auf, so wird dies dem Benutzer durch einen entsprechenden Returnwert mitgeteilt; außerdem stellt TROL noch einige Call-Back Adressen zur Verfügung, die es dem Benutzer erlauben, eigene Exception-Handler zu definieren, die das Verhalten der Applikation im Fall des Auftretens von Kommunikationsfehlern festlegen.

Kommunikation

Das TROL Schichtenmodell

Das TROL-Kommunikationsmodell orientiert sich am **OSI**-Schichtenmodell[2], stellt aber keinesfalls den Anspruch, OSI-konform zu sein. Es besteht allerdings eine grobe Zuordnung zwischen verschiedenen Schichten der beiden Modelle:

TROL		OSI	
Bitübertragungs-Schicht	(1)	Physical Layer	(1)
Sicherungs-Schicht	(2)	Data Link Layer	(2)
Transaktions-Schicht	(3)	Session Layer	(5)
Darstellungs/Anwendungs-Schicht	(4)	Presentation Layer &	(6)
		Application Layer	(7)

Schicht 2 und 3 sind, wenngleich auch hinsichtlich des Datenaustauschs zwischen PC und CRS-Controller optimiert, prinzipiell applikationsunabhängig, können also auch in anderen Anwendungen eingesetzt werden, während Schicht 4 applikationsspezifisch ist. Die Aufgaben der 4 Schichten des TROL-Modells sollen im folgenden erläutert werden.

Bitübertragungs-Schicht (Schicht 1)

Diese Schicht ist für die Ansteuerung der seriellen Kommunikationsbausteine zuständig und daher extrem hardwareabhängig; sie stellt sowohl am PC als auch am Controller Routinen zur Übertragung und zum Empfang von Zeichen über die serielle Schnittstelle zur Verfügung. Für DOS-Applikationen wurde eigens ein interruptgetriebener serieller Treiber entwickelt, unter Windows werden die vom Betriebssystem zur Verfügung gestellten Routinen zur seriellen Kommunikation verwendet. In Abhängigkeit vom verwendeten Betriebssystem sind Übertragungsraten bis zu 38400 Baud möglich .

---

[2] OSI (**O**pen **S**ystems **I**nterconnection): von der ISO vorgeschlagenes (logisches) Architekturmodell für Kommunikationsnetzwerke.

Die Schicht 1 setzt voraus, daß PC und Controller über die seriellen Kanäle **COM1** oder **COM2** (PC-seitig) und **DEV1** (controllerseitig) miteinander verbunden sind.

Sicherungs-Schicht (Schicht 2)

Sie ist für die sichere Übertragung von Datenpaketen zwischen PC und Controller zuständig. Zu diesem Zweck wurde von mir ein einfaches, BSC-ähnliches Halbduplex-Protokoll entwickelt, das und unter anderem folgende Merkmale aufweist:

- Kleine Framegröße; der letzte Frame muß nicht vollständig übertragen werden.
- Prüfsummen und Framenumerierung.
- Zusätzliche Sicherheit und Stabilität durch selbstkorrigierende Kontrollcodes.
- Sehr kleine Datenmengen (kleiner als 3 Bytes) können vollständig im Header übertragen werden.
- Es können Datenmengen bis zu 64K übertragen werden (diese Beschränkung ergibt sich aus der segmentierten Speicherarchitektur der Intel-Prozessor-Familie).

Transaktions-Schicht (Schicht 3)

Faßt Sende- und Empfangsanforderungen zu Transaktionen zusammen. Das Protokoll dieser Schicht verlangt, daß jedes Datenpaket des Senders durch ein Datenpaket des Empfängers quittiert wird; es muß also sichergestellt werden, daß bis zum vollständigen Empfang der Quittung keine weiteren (z.B. durch ISRs[3] generierten) Sendeanforderungen bearbeitet werden können. Die Implementierung dieses Features ist allerdings problematisch, da weder RAPL noch DOS die dazu notwendigen IPC-Mechanismen wie Semaphoren oder Priority-Queues zur Verfügung stellen; das Protokoll dieser Schicht ist daher momentan - wie ACI - als Master-Slave Protokoll implementiert, d.h. Transaktionen dürfen ausschließlich vom PC und nur auf synchrone Weise initiiert werden.

TROL-Transaktionen sind allerdings nicht atomar im klassischen Sinn: wurde z.B. am PC ein MOVE-Befehl abgesetzt, der vom Controller zwar korrekt empfangen und an das RAPL-BIOS weitergegeben wurde, dessen Quittung aber nicht erfolgreich an den PC übermittelt werden konnte, so ist es nicht möglich, diesen MOVE-Befehl wieder rückgängig zu machen; es

---

[3] ISR (**I**nterrupt **S**ervice **R**outine); dient der Bearbeitung eines asynchronen, durch einen Interrupt angezeigten Ereignisses.

stehen allerdings Mechanismen zur Verfügung, die es erlauben, nach Wiederherstellen der Verbindung zu überprüfen, ob das zuletzt abgesetzte Kommando korrekt ausgeführt wurde.

Eine weitere Aufgabe dieser Schicht könnte darin bestehen, Datenpakete, die aufgrund ihrer Größe nicht durch ein einziges Paket der Sicherungsschicht übertragen werden können, in mehrere Datenpakete aufzuteilen, diese nacheinander zu übertragen und empfangsseitig wieder zusammenzusetzen. Da die Sicherungsschicht allerdings Pakete bis zu einer Größe von 64K akzeptiert, ist es momentan weder sinnvoll noch notwendig, diese Möglichkeit zu unterstützen.

Darstellungs- und Anwendungs-Schicht (Schicht 4)

Die über die serielle Schnittstelle zu übertragenden Daten sind Instanzen der im „C“-Header-File "commands.h" deklarierten TROL Record-Datentypen; die Darstellungs-Schicht setzt den über die Schnittstelle empfangenen amorphen Byte-Strom in eine Instanz eines solchen Datentyps um. Die Anwendungsschicht hingegen bildet die Schnittstelle zwischen der Kommunikationssoftware und den darauf aufsetzenden Anwendungen. Ihre Aufgabe ist es, dem Anwendungsprogrammierer auf dem PC ein komfortables High-Level API zur Verfügung zu stellen.

Protokolle und Dienste

Für jede der Schichten 2 - 4 existiert ein eigenes **Protokoll**, das festlegt, auf welche Weise Verbindungsaufbau und -abbau sowie Datentransfer zu erfolgen haben; Protokolle regeln also die vertikale Kommunikation zwischen Sender und Empfänger. Zur Ausführung ihrer Aufgaben bedient sich jede Protokollschicht der **Dienste** der ihr - vertikal - direkt untergeordneten Schicht.

Protokoll der Schicht 1

Die Schicht 1 verfügt über kein Protokoll; die von ihr zur Verfügung gestellen Dienste beinhalten neben Routinen zur Initialisierung der seriellen Schnittstelle und zum Setzen/Abfragen der Handshake-Leitungen (letztere werden von den darüberliegenden Schichten allerdings nicht benötigt) die Routinen *SendByte()* und *GetByte()* zur Übertragung resp. zum Empfang einzelner Bytes sowie die Routinen *BufferEmpty()* und *FlushBuffer()* zum Abfragen des Status resp. zum Leeren des Eingangsbuffers. Es wird vorausgesetzt, daß PC und Controller über die seriellen Kanäle COM2 bzw. COM1 und DEV1 miteinander verbunden sind.

Beim Senden von Daten vom PC zum Controller wird zwischen der Übertragung zweier Zeichen ein kurzes Zeitintervall gewartet, um sicherzugehen, daß der Controller genug Zeit hat, die ankommenden Zeichen zu verarbeiten; diese Zeitspanne beträgt standardmäßig 5 ms und kann mit Hilfe der Funktion *SetDelay()* geändert werden.

Protokoll der Schicht 2

Das Protokoll der Schicht 2 ist ein BSC-ähnliches Halbduplex-Protokoll; seine Aufgabe besteht darin, Datenpakte bis zu einer Größe von 64K zuverlässig über die serielle Schnittstelle zu übertragen, wobei "zuverlässig" in diesem Zusammenhang bedeutet, daß Übertragungsfehler erkannt und ggf. auch korrigiert werden. Schicht 2 stellt ihren Benutzern die Dienste *SendData()* und *ReceiveData()* zum Senden bzw. zum Empfang von Datenpaketen zur Verfügung.

Datenübertragung

Der Verbindungsaufbau erfolgt durch Senden von **ENQ** und Bestätigung des Empfängers mittels **ACK**. Wurde die Verbindung erfolgreich aufgebaut, wird zunächst der Header übertragen:

Header

Feld	Größe in Bytes	Beschreibung
SOH	1	Start Of Header - Steuercode
#bytes	2	Anzahl der zu übertragenden Bytes
PCI	2	Protocol Control Information
firstWord	2	Enthält die ersten beiden Datenbytes
BCC Prüfsumme	2	Block Check Characters -
ETX	1	End Of Text - Steuercode
	10	

Der Beginn des Headers wird stets durch das Steuerzeichen **SOH** angezeigt.

Die in **PCI** enthaltene Protokollinformation ist für Schicht 3 (Transaktionsschicht) bestimmt. Idealerweise sollte diese Information nicht gesondert transportiert, sondern - transparent für Schicht 2 - als Nutzdaten übertragen werden; der Grund dafür, die PCI trotzdem im Header zu übertragen, ist, daß dadurch die Verwaltung des dynamischen Speichers stark vereinfacht und das Kopieren von Information zwischen den Schichten vermieden wird: der von Schicht 2 an die Schicht 3 zurückgelieferte Zeiger auf die empfangenen Nutzdaten kann direkt an die Schicht 4 weitergegeben werden. Das PCI-Feld wird in der aktuellen Version von TROL allerdings nicht benötigt und ist für zukünftige Erweiterungen der Schicht 3 vorgesehen.

firstWord enthält die ersten beiden Bytes der zu übertragenden Nutzdaten; Nachrichten, die weniger als 3 Byte enthalten - was in über 50% aller Fälle zutrifft -, werden vollständig im Header übertragen, wodurch der mit dem Senden eines Datenframes verbundene Overhead wegfällt.

Die Prüfsumme des Headers ist in **BCC** enthalten; sie berechnet sich als Summe aller im Header enthaltenen Bytes mit Ausnahme der Steuercodes[4].

Der Header wird schließlich durch den Steuercode **ETX** abgeschlossen.

Nachdem der Empfänger den Erhalt des Headers mittels **ACK** bestätigt hat, können nun die Datenframes übertragen werden. Jeder einzelne Frame muß vom Empfänger ebenfalls mittels **ACK** bestätigt werden, bevor der nächste Frame übertragen werden kann:

Frame

Feld	Größe in Bytes	Beschreibung
STX	1	Start Of Text - Steuercode
data	var	Nutzdaten: 1 .. FRAME_SIZE Bytes
frame-ID	2	Fortlaufende Framenummer
BCC	2	Block Check Characters - Prüfsumme
ETB	1	End Of Transmission Block - Steuercode
	7 .. (FRAME_SIZE + 7)	

Datenframes beginnen stets mit dem Steuerzeichen **STX**.

data enthält die im Frame zu übertragenden Nutzdaten; die Größe dieses Feldes beträgt - mit Ausnahme des letzten Frames - FRAME_SIZE Bytes (die Konstante FRAME_SIZE ist in "protoc.h" definiert); der letzte Frame kann auch kleiner sein, enthält aber zumindest 1 Byte an

[4]Dieses einfache Verfahren zur Berechnung von Prüfsummen wird als LRC (**L**ongitudinal **R**edundancy **C**heck) bezeichnet. Das bekannte CRC (**C**yclical **R**edundancy **C**heck) Verfahren arbeitet hingegen mit Polynomen.

Nutzdaten. Sind - wie im Header spezifiziert- **#bytes** Bytes an Nutzdaten zu übertragen, so berechnet sich die Anzahl der zu übertragenden Frames, #frames, wie folgt:

```
#frames = (#bytes - 2) DIV FRAME_SIZE ;
IF ((#bytes-2) MOD FRAME_SIZE ) > 0 THEN
    #frames := #frames + 1
END;
```

Die Subtraktion von 2 ergibt sich aus der Tatsache, daß die ersten beiden Datenbytes im Header transportiert werden.

Jeder Frame ist durch eine eindeutige **frame-ID** gekennzeichnet. Frame-IDs werden, mit 0 beginnend, aufsteigend vergeben. Sie haben - anders als z.B. bei HDLC[5] - für das Protokoll selbst keine Bedeutung und werden nur für eine zusätzliche Sicherheitsüberprüfung herangezogen: stimmt die ID des zuletzt empfangenen Frames nicht mit der erwarteten frame-ID überein, so gilt dies als fataler Fehler, der zu einem sofortigen Abbruch der Verbindung führt.

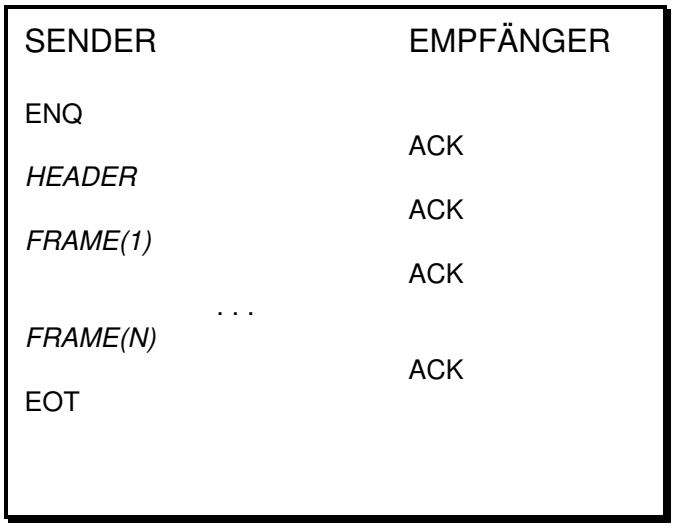
Die in **BCC** enthaltene Prüfsumme des Frames berechnet sich, ebenso wie jene des Headers, mittels des LRC-Verfahrens.

Datenframes werden - mit Ausnahme des letzen Frames, dessen Ende durch **ETX** angezeigt wird - mit dem Steuercode **ETB** abgeschlossen.

Nachdem der Empfänger den Erhalt des letzten Frames bestätigt hat, schließt der Sender die Verbindung durch Übertragung des Steuercodes **EOT**. Das Protokoll der Schicht 2 ist in folgendem Diagramm nochmals zusammengefaßt:

---

[5]HDLC (**H**igh **L**evel **D**ata **L**ink **C**ontrol) ist das im OSI-Modell vorgesehen Protokoll des Data Link Layer. Es ist ein Vollduplexprotokoll, in dem eine bestimmte Anzahl von Frames (das sogenannte Frame-Window) übertragen werden kann, ohne auf eine Bestätigung des Empfängers warten zu müssen; jede Bestätigung muß jedoch die frame-ID des zuletzt empfangenen Frames enthalten.



Steuercodes

Die vom Protokoll der Schicht 2 verwendeten Steuercodes (*control characters*) wurden so gewählt, daß sie sich in zumindest 3 Bits voneinander unterscheiden. Wird nun an einem Punkt des Protokolls ein bestimmter Steuercode erwartet und ein Zeichen empfangen, das sich in weniger als 2 Bits von diesem unterscheidet, so wird das Zeichen akzeptiert. Da der Empfänger *a priori* weiß, an welcher Position das nächste Steuerzeichen zu erwarten ist, benötigt das Protokoll keine *bit stuffing*-Mechanismen, die verhindern, daß ein Datenbyte fälschlicherweise als Steuerzeichen interpretiert wird. Sämtliche Steuercodes sind in der Datei "protoc.h" definiert.

Fehlerbehandlung in der Schicht 2

Als Fehlerursachen kommen Timeouts und Übertragungsfehler in Frage. Erkennt der Empfänger einen Übertragungsfehler, so teilt er dies dem Sender durch ein **NAK**, gefolgt von einem Fehlercode, mit; dies ist allerdings nur an jenen Stellen im Protokoll möglich, an denen die Übertragung eines **ACK** vorgesehen ist, also nach dem vollständigen Empfang des Headers, eines Datenframes oder von **ENQ**.

Wird ein Timeout oder ein fataler Fehler festgestellt, so führt dies zu einem sofortigen Abbruch der Verbindung; ansonsten wird versucht, den fehlerhaften Frame (bzw. den Header) nochmals zu übertragen. Die in "protoc.h" definierte Konstante MAX_RETRIES gibt an, wie oft die Übertragung eines einzelnen Frames maximal wiederholt werden darf; ein Überschreiten dieses Limits führt ebenfalls zum Abbruch der Verbindung. Ist es der Schicht 2 nicht möglich, einen Kommunikationsfehler selbständig zu beheben, treten die Fehlerbehandlungsmechanismen der höheren Schichten in Aktion (siehe Abschnitt 2.2.3. sowie 2.2.4).

Protokoll der Schicht 3

Die Hauptaufgabe der Schicht 3 besteht, wie bereits erwähnt, darin, Sende- und Emfpangsanforderungen an die Schicht 2 zu Transaktionen zusammenzufassen; Sendeanforderungen werden im folgenden als **Kommandos**, Empfangsanforderungen als **Quittungen** bezeichnet. Schicht 3 stellt die beiden Dienste *TransSR()* und *TransRS()* zur Verfügung, welche das Versenden eines Kommandos und den Empfang der Quittung respektive den Empfang eines Kommandos und das Versenden der Quittung übernehmen. Für den Inhalt der Quittung ist jedoch die Schicht 4 und nicht die Schicht 3 verantwortlich!

Die Fehlerbehandlung erfolgt durch **Callback-Routinen**[6], sogenannte **Error-Handler**. Diese Vorgangsweise bietet größtmögliche Flexibilität: so kann kann z.B. der Benutzer im Falle eines fehlgeschlagenen Sendevorganges gefragt werden, ob er diesen wiederholen oder die Transaktion abbrechen möchte. Standardversionen dieser Error-Handler sind in "callback.c" implementiert; in Abschnitt 3.3. wird beschrieben, wie eigene Error-Handler definiert werden können.

Protokoll der Schicht 4

Schicht 4 definiert die Syntax der zu übertragenden Datentypen und legt sogenannte **Transaktionsklassen** fest; die von dieser Schicht angebotenen Dienste sind über das TROL-API zugänglich (siehe Kapitel 5).

Transaktionsklassen fassen Kommandos zusammen, deren Quittungen ein einheitliches Format aufweisen: so gehören z.B. die meisten Bewegungsfehle wie *Move()*, *Align()* etc. zu den sogenannten CAT1-Transaktionen; diese erwarten eine 16-Bit Integer als Quittung, die entweder den Returnwert des RAPL-BIOS (> -32000) oder einen TROL-Fehlercode (-32000) enthält. Jede Transaktion muß genau einer Transaktionsklasse angehören.

Die Fehlerbehandlung erfolgt - wie in Schicht 3 - über eine Callback-Funktion. Diese kann auch als "Filter" eingesetzt werden, der festlegt, ob bestimmte Fehler dem Benutzer über den API-Returnwert mitgeteilt oder auf andere Weise behandelt werden sollen.

Zusammenwirken der Protokollschichten

Nachdem in den vorhergehenden Abschnitten besprochen wurde, welche Aufgaben die einzelnen Protokollschichten wahrnehmen und wie ihre Schnittstellen (Dienste) zu den anderen

---

[6] Unter Callback-Routinen versteht man vom Benutzer bereitzustellende Unterprogramme, welche vom System - in konkreten Fall von TROL - aufgerufen werden; sie stellen also in gewisser Weise das Gegenstück zu den klassischen *system calls* dar.

Schichten aussehen, wird in diesem Abschnitt beschrieben, wie die verschiedenen Protokollebenen bei der Übertragung eines Kommandos an den Roboter zusammenwirken.

Abbildung 1 und Abbildung 2 zeigen den vertikalen und horizontalen Kontrollfluß zwischen den einzelnen Schichten. Durchgehende vertikale Linien symbolisieren den Aufruf, gestrichelte vertikale Linien die Rückkehr (Return) einer Dienstfunktion. Vertikale Linien hingegen symbolisieren die Kommunikation zwischen den zwei Partnerinstanzen einer Protokollschicht am PC und am Controller: wie in Abbildung 1 angedeutet, interpretiert jede der Protokollschichten die zu übertragenden Daten auf andere Weise. Während das zu übertragendes Datenpaket *req* für die Schicht 2 nur eine Folge von Bytes darstellt, wird es von der Schicht 3 als Kommando und von der Schicht 4 als Instanz eines TROL-Datentyps behandelt.

Eine TROL-Transaktion besteht, wie bereits ausgeführt, aus dem Senden einer Anforderung und dem Empfang einer Quittung. Die Übertragung des Anforderungspakets ist in Abbildung 1 dargestellt. Bevor eine Übertragung möglich ist, muß die Schicht 4 am Controller empfangsbereit sein; dies geschieht, indem sie mittels der Dienstfunktion *TransRS()* die Kontrolle an die Schicht 3 übergibt; diese ruft nun ihrerseits die Funktion *ReceiveData(PCI + req)* der Schicht 2 auf, welche daraufhin wartet, bis ein Paket über die serielle Schnittstelle übertragen wird, um dieses nach Emfpang an die rufende Schicht weiterzugeben. Ruft nun der Benutzer am PC eine der TROL-API-Funktionen auf (Schnittstelle zur Schicht 4), so wird die entsprechende Instanz *req* eines TROL-Anforderungs-Datentyps erstellt und der Schicht 3 mittels Aufruf von *TransSR(req, resp)* übergeben; *resp* enthält nach der Rückkehr von *TransSR()* einen Zeiger auf das Quittungsdatenpaket. Die Schicht 3 ergänzt das Anforderungspaket schließlich noch um die sogenannte PCI (Protocol Control Information), die Anweisungen an die Parnterinstanz am Controller enthalten kann, bevor das gesamte Paket mittels Aufruf von *SendData(PCI + req)* der Schicht 2 zur Übertragung an den Controller übergeben wird.

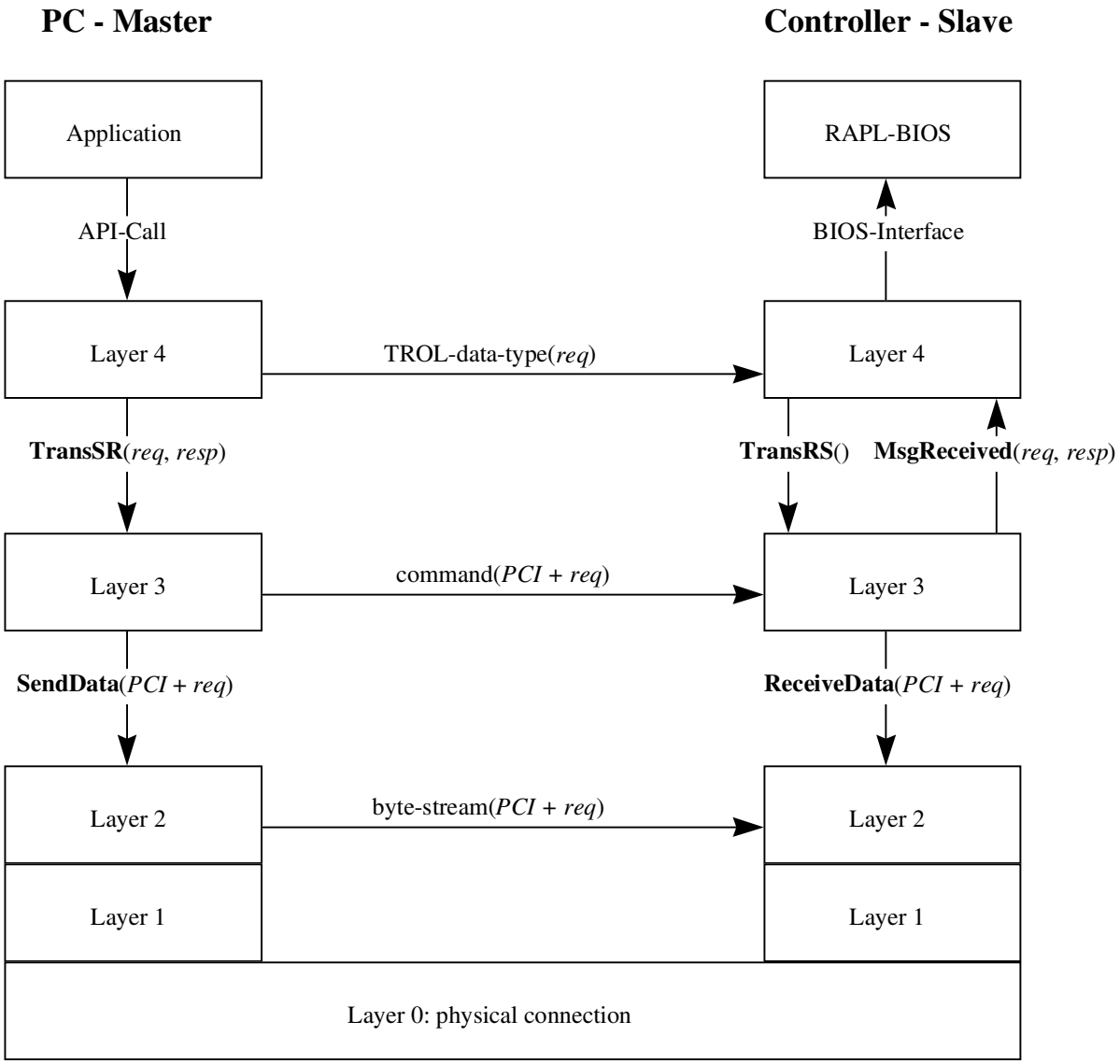


Abbildung 1. Kontrollfluß zwischen den Protokollebenen bei der Übertragung eines Anforderungspakets an den Controller. Durchgezogene Pfeile in vertikaler Richtung symbolisieren den Aufruf von Dienstfunktionen.

Controllerseitig nimmt die Schicht 2 das Anforderungspaket entgegen und gibt es an die Schicht 3 weiter (Rückkehr des Aufrufs*: ReceiveData(PCI + req)*). Diese entfernt das PCI-Feld und übergibt das Anforderungspaket *req* durch den Aufruf von *MsgReceived(req, resp)* an die Schicht 4. Das Anforderungspaket wird nun dekodiert und in den Aufruf einer Funktion des RAPL-BIOS umgesetzt; der Returnwert des BIOS (sowie ggf. andere Informationen) werden in einer Instanz eines TROL-Quittungsdatentyps - *resp* - gespeichert; ein Zeiger auf diese Struktur wird der Schicht 3 - in Form eines Ausgangsparameters der Funktion *MsgReceived(req, resp)* - zur Übertragung an den PC übergeben. Die weitere Übertragung des Quittungspaket erfolgt analog der Übertragung des Anforderungspakets, obwohl die Kommunikation zwischen den Schichten 3 und 4 bzw. der Schicht 4 und der

Anwendungsebene nun nicht durch Aufruf von Dienstfunktionen, sondern durch deren Rückkehr (Return) erfolgt; dies ist in Abbildung 2 dargestellt.

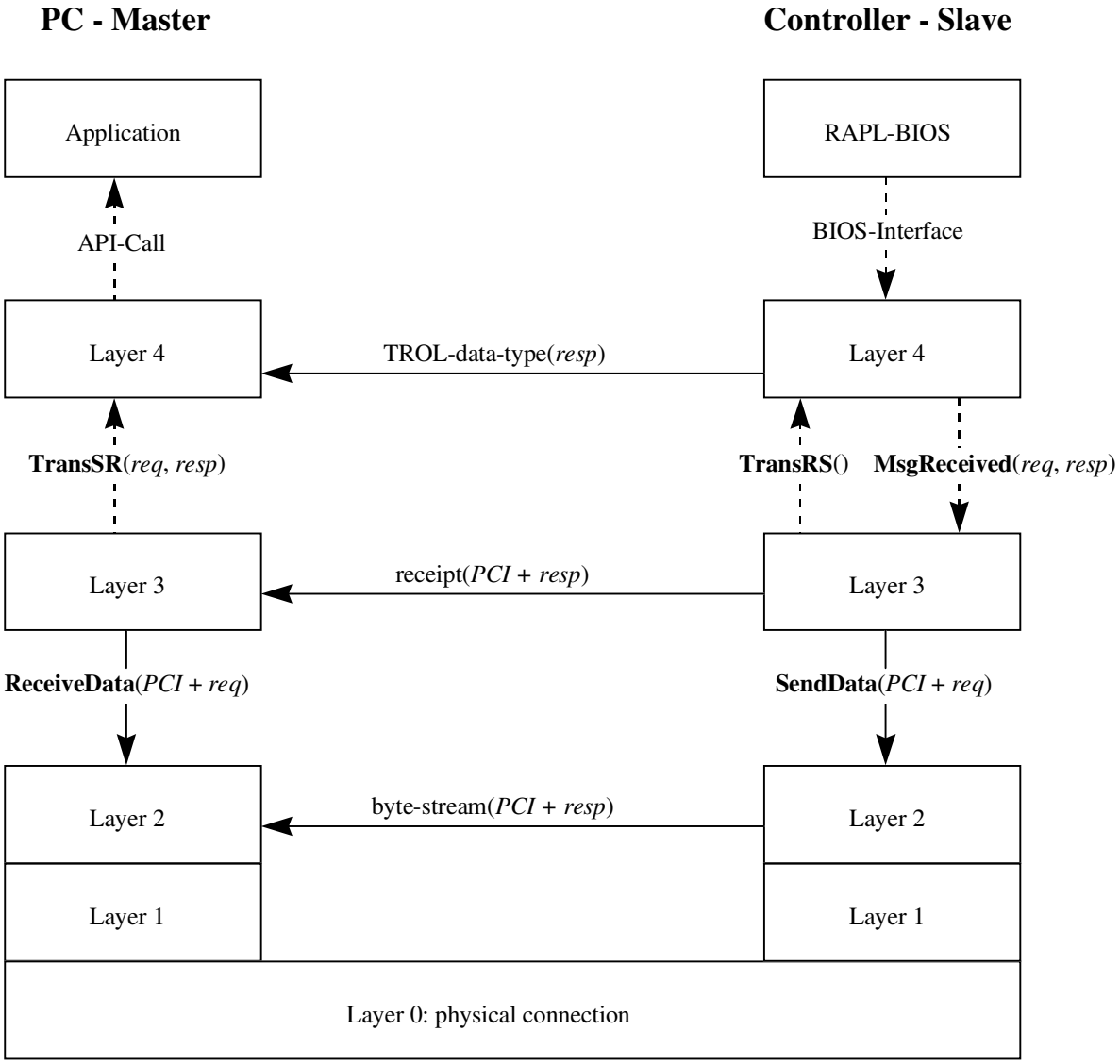


Abbildung 2. Kontrollfluß zwischen den Protokollebenen bei der Übertragung eines Quittungspakets an den PC. Gestrichelte Pfeile symbolisieren die Rückkehr (function return) von Diensfunktionen zur rufendenden Protokollschicht.

Installation von TROL

TROL wurde mit Borland C (BC) entwickelt. Diese Compilerfamilie unterstützt sogenannte "Project-Files", die eine Weiterentwicklung der bekannten Makefiles darstellen und ein komfortables Projekt-Management ermöglichen; sie haben die Dateiendung *.prj (BC 3.1) oder *.ide (BC 4.0+). Die für die Erstellung der DOS- und Windows-Beispielanwendungen verwendeten Project-Files befinden sich im Unterverzeichnis PROJECT; für die Erstellung des PCP-Servers "pcptrol.exe" wurde ein Borland-Makefile verwendet, welches sich im Verzeichnis PROJECT\PCP befindet.

Die Installation von TROL geschieht durch Kopieren bzw. Entpacken von trol.zip an einer beliebigen Stelle des Verzeichnisbaums. Die Projekt- und Make-Files verwenden, soweit möglich, relative Verzeichnisangaben; die Pfade für Compiler-Binaries und -Includes müssen vom Benutzer jedoch entsprechend geändert werden. Im Lieferumfang ist nur ein einziges ausführbares File - nämlich der PCP-Server “pcptrol.exe” -enthalten; alle anderen Programme müssen vom Benutzer mit Hilfe der Project-Files zuerst erstellt werden.

Verzeichnisstruktur

Folgende Unterverzeichnisse sind im Stammverzeichnis TROL enthalten:

SOURCE
: Enthält sämtliche *.c und *.asm Sourcedateien.

INCLUDE
: Enthält sämtliche *.h Dateien.

PROJECT
: Enthält die Unterverzeichnisse INIT (Initialisierung und Cleanup), INTERP (Kommandointerpreter), UTIL (Hilfs- und Testprogramme) sowie PCP. Die Projektdateien „interp.prj“ bzw. „interp.ide“ im Verzeichnis INTERP erstellen als Beispielanwendung einen TROL-Kommandozeilen-Interpreter; sie können auch benutzt werden, um alle TROL-Source-Dateien für die jeweilige Plattform neu zu kompilieren. Im Verzeichnis PCP befindet sich ein Borland-Make-File zur Erzeugung des PCP-Servers „pcptrol.exe“.

LIB
: Enthält die Unterverzeichnisse PCPLIB und TROLLIB. PCPLIB enthält alte Versionen der Borland RTLs (Run Time Libraries) cl.lib, mathl.lib und fp87.lib. Sollte es erforderlich sein, den PCP-Server "pcptrol.exe" neu zu kompilieren, so müssen unbedingt diese - und nicht die mit Borland C 3.1 oder einer späteren Version mitgelieferten - Bibliotheken gelinkt werden. Unter PCPLIB befinden sich die statischen TROL-Bibliotheken “trol.lib“ sowie jeweils eine DOS-Batch-Datei „makelib.bat“, welche diese erstellt.

OBJECT
Ausgabeverzeichnis der Project Files; enthält alle *.exe, *.obj und *.map Files.

DOC
Enthält die Dokumentation (unter anderem diese Datei im Word 7.0-Format).

CONFIG
Enthält die Beipiel-Konfigurationsdatei “trol.cfg”.

Beschreibung der Module

Im folgenden wird eine kurze Übersicht über die im Verzeichnis SOURCE enthaltenen Module gegeben:

ACIOFF.C
Enthält einen rudimentären ACI-Protokolltreiber Von besonderer Wichtigkeit ist die Routine *ACIOff()*, welche zum Ausschalten des ACI-Protokolls dient.

CALLBACK.C
Standardversionen der Error-Handler für die Protokollschichten 3 und 4.

INITTROL.C
Enthält Routinen zum Initialisieren der seriellen Schnittstelle und zum Aktivieren/Deaktivieren des PCP-Servers.

PROTOASM.ASM
Serieller DOS-Treiber für PC und Controller. Die Eingabe erfolgt interruptgesteuert, empfangene Zeichen werden in einem Ringbuffer abgelegt.

PROTOC.C
Implementierung der Schicht 2.

SEND.C
Implementierung der TROL API-Funktionen.

SERCOM.C
Implementiert die Schicht 1; enthält Routinen zur Programmierung der seriellen Bausteine (UARTs).

TRANSACT.C
Implementiert die Schicht 3 (Transaktions-Schicht).

TROLAUX.C
Enthält Timer- und Speichermanagement-Routinen.

INTERP\SCANNER.C
Lexikalischer Scanner für den TROL-Kommando-Interpreter.

INTERP\MAIN.C
Lexikalischer Scanner, Hauptprogramm.

PCP\BIOSCALL.C
Enthält das "C"-Interface für den Aufruf des RAPL-BIOS.

PCP\FLOAT2L.C
Enthält eine Routine zur Umwandlung von *float* in *long integer*-Werte. Sollen die Borland-RTLs nicht mit dem PCP-Server "pcptrol.exe" gelinkt werden, so müssen alle impliziten oder expliziten (type cast) Konvertierungen von *float* nach *integer* durch einen Aufruf dieser Routine ersetzt werden. Diese Funktion wird normalerweise jedoch nicht benötigt.

PCP\PCPTROL.C
Code des PCP-Servers

PCP\START.ASM
Startup-Code für den PCP-Server "pcptrol.exe". Wird für PCPs anstelle des Borland C Startup-Moduls "c0.asm" verwendet.

PCP\STUB.ASM
Dieses Modul muß ebenfalls mit "pcptrol.exe" gelinkt werden. Es enthält die Dummy-Funktion abort*()*, die von den Borland RTLs aufgerufen wird und normalerweise in "c0.asm" enthalten ist. Weiters enthält es die beiden Funktionen *_inportb()* und *_outportb()*; diese werden benötigt, wenn die Borland RTLs nicht gelinkt werden sollen.

TEST*
Routinen zum Testen der verschiedenen Protokollschichten.

OTHER*
Zusätzliche Hilfsroutinen, z.B. für die Utilities „inittrol.exe“ und „cluptrol.exe“.

Bedingte Kompilation

In den meisten Modulen wird bedingte Compilation verwendet, um nicht jeweils mehrere fast identische Module warten zu müssen. Projectfiles - mit ihnen werden die Module für den Einsatz unter DOS oder Windows compiliert und gelinkt - definieren die Konstanten DOS, WIN16 oder WIN32, die für die Erstellung von PCPs zuständigen Makefiles definieren hingegen die Konstante CRS. Es ist darauf zu achten, daß Object-Files gleichen Namens, die für unterschiedliche Targets compiliert wurden, in verschiedenen Verzeichnissen abgelegt werden. Im folgenden sind alle Konstanten, die Einfluß auf Kompilation/Assemblierung haben, aufgelistet:

DOS
Es wird ein DOS-Executable erstellt.

WIN16
Es wird ein Windows 3.1-Executable (16 bit) erstellt.

WIN32
Es wird ein Windows95 oder Windows-NT-Executable (32 bit) erstellt.

CRS
Es wird ein PCP erstellt; wird von allen Makefiles definiert.

NO_RTLS
Muß definiert werden, wenn zum PCP-Server "pcptrol.exe" keine Borland-Laufzeitbibliotheken gelinkt werden sollen (diese Möglichkeit wird aber momentan noch nicht unterstützt).

TROL_ROUTER
Wenn definiert, kann das Programm sowohl als TROL-Master als auch als TROL-Slave fungieren. Wird z.B. für das Utility “prottest.exe” benötigt, sollte im Normalfall jedoch nicht definiert sein.

Konfiguration

Die Konfigurationsdatei “trol.cfg” wird von allen Programmen, welche TROL verwenden, benötigt. Sie enthält 3 Einträge im ASCII-Format:

1. Baudrate des CRS-Controllers.

2. Serieller Port, über den die Kommunikation mit dem CRS-Controller erfolgt.

3. Tool-Transformation. Bei der Initialisierung des PCP-Servers wird die Tool-Transformation automatisch ausgeführt (siehe Kapitel 5, API-Funktion *TROLSetToolTransform*()).

Der Benutzer muß vor dem Aufruf von TROL unbedingt sicherstellen, daß die in der Konfigurationsdatei angegebenen Werte korrekt sind! TROL läuft mit einer Baudrate von 9600 auf allen unterstützen Plattformen stabil, die Verwendung höherer Baudraten kann jedoch - insbesondere unter Windows 3.1. - zu Problemen führen. Im besonderen ist darauf zu achten, daß die angegebene Tooltransformation mit dem tatsächlich verwendeten Tool übereinstimmt; ist das verwendete Tool z.B. kürzer als in der Tooltransformation angegegeben, kann dies bei Ansteuerung einer Position nahe der x-y-Ebene zur Kollision des Greifers mit der Arbeitsplatte führen!

Sollte TROL auf einer Windows-Plattform nicht funktionieren, so hilft es evt., die Kommunikation zum Roboterkontroller über einen DOS-Rechner durchzuschleifen. Dies ist mit Hilfe des Utilities “link.exe” möglich. “link.exe” initialisiert zunächst den Controller (über die in trol.cfg angegebene Schnittstelle) und gibt anschließend einfach alle empfangenen Daten

über die zweite (als Parameter übergebene) Schnittstelle weiter - und umgekehrt. Die Verwendung von “link.exe” geschieht in für den Benutzer transparenter Weise, d.h. ein TROL-Programm kann nicht feststellen, ob es direkt mit dem Controller oder mit “link.exe” kommuniziert.

Programmierung

Initialisierung

Folgende Schritte sind erforderlich, wenn eine Applikation mittels TROL auf den Roboter zugreifen soll:

1. Herstellen der physikalischen Verbindung zwischen PC und Controller. Der Port **COM1** oder **COM2** des PCs muß über ein serielles 0-Modem-Kabel mit dem Port **DEV1** des Controllers verbunden sein.

2. Installation des PCP-Servers. Das PCP-Executable "pcptrol.exe" muß, sofern dies nicht bereits geschehen ist, ins Controller-Memory geladen werden. Dies erfolgt mit Hilfe des UtilityProgramms **ROBCOMM** (Menüpunkt "File/PCP Load"). Es ist nicht notwendig, das PCP nach jedem Hochfahren des Controllers erneut zu laden. Evt. muß der Controllerspeicher jedoch vor dem erstmaligen Laden des PCP-Servers neu partitioniert werden.

3. Aufruf des Utilities „inittrol.exe“. Dieses stellt die serielle Verbindung zum Controller her und aktiviert den PCP-Server (unter Windows 3.1 muß das Ausgabefenster nach erfolgtem Verbindungsaufbau unbedingt wieder geschlossen werden, da sonst kein anderer Prozess auf die serielle Schnittstelle zugreifen kann).

4. Starten und Ausführen der Applikation.

5. Beenden der Applikation.

6. Aufruf des Utilities „cluptrol.exe“. Dieses deaktiviert den PCP-Server im Controller-Memory. Wird dieses Programm nicht aufgerufen, so kann z.B. mittels des **ROBCOMM**-Utilities nicht auf den Roboter zugegriffen werden.

Erstellen eigener Applikationen

TROL wurde mit Borland C (V3.1 - DOS, V4.0 - Win16 und V5.0 - Win32) entwickelt; es wird daher empfohlen, Applikationen, die TROL verwenden, ebenfalls mit diesem Compiler zu erstellen; von der Verwendung der Compilerversion 4.5 wird jedoch abgeraten, da diese schwerwiegende Mängel enthält und sich TROL mit dieser Version nicht compilieren bzw. linken ließ! Für die Erstellung des DOS-Clients und des PCP-Servers wird außerdem ein MASM-kompatibler Assembler benötigt.

Bevor innerhalb der Applikation eine der TROL-API-Funktionen (siehe Kapitel 5) aufgerufen wird, muß die Funktion *InitTROL*(..., 1) aufgerufen werden. Ist der letzte Parameter 1, werden alle anderen Parameter ignoriert; *InitTROL*() stellt in diesem Fall nur die serielle Verbindung

her und überprüft, ob der PCP-Server bereits installiert wurde. DOS-Applikationen müssen mit dem Speichermodell LARGE erstellt werden. Alle TROL-Applikationen benötigen die statische Bibliothek „trol.lib“ (im Verzeichnis LIB\TROLLIB).

Das Verzeichnis PROJECT\INTPERP enthält die Beispielanwendung „interp.exe“ (Kommandointerpreter), die als Ausgangspunkt für eigene Entwicklungen dienen kann. Das zugehörige Projektfile „interp.prj“ bzw. „interp.die“ verwendet nicht die Bibliothek „trol.lib“, sondern bindet alle Source-Dateien einzeln ein, so daß es zur Neukompilation aller TROL-Sourcen verwendet werden kann (z.B. wenn die Bibliothek „trol.lib“ neu erstellt werden soll).

Das TROL-API ist in Kapitel 5 beschrieben, die Returnwerte der API-Funktionen sind in Kapitel 6 dokumentiert.

Callback-Routinen

Um die Fehlerbehandlung möglichst flexibel zu gestalten, ermöglicht es TROL dem Benutzer, sogenannte Error-Handler zu definieren, die im Fall des Auftretens eines Kommunikationsfehlers aufgerufen werden. Die Datei "SOURCE\callback.c" enthält Standardversionen dieser Error-Handler. Sollen diese durch eigene Versionen ersetzt werden, empfiehlt es sich,"callback.c" unter anderem Namen zu sichern und anschließend zu modifizieren: Project- und Makefiles, die "callback.c" verwenden, müssen dann nicht geändert werden.

ErrOnSend() und *ErrOnReceive()* können sowohl am PC als auch am Controller aufgerufen werden (ggf. bedingte Kompilation verwenden!), wohingegen *ErrAppTimeOut()* und *FatalError()* nur PC-seitig benötigt werden.

TUINT *ErrOnSend*(**SInData** **in*, **TUINT** **PCI*)

Wird von *TransSR()* oder *TransRS()* (Schicht 3) aufgerufen, falls das Datenpaket *in* nicht korrekt übertragen werden konnte, der Aufruf *SendData(in, PCI)* also einen Fehlercode zurückliefert. Dieser Fehlercode kann mittels errID = *GetLastCommErr()* abgefragt und zur Auswahl der Fehlerbehandlungsstrategie herangezogen werden (Anhang B enthält sämtliche Fehlercodes und gibt auch Hinweise zur Fehlerbehandlung). Konnte der Kommunikationsfehler behoben werden, ist von *ErrOnSend()* der Wert RECOVERY_SUCCESS zurückzugeben, andernfalls RECOVERY_FAIL.

TUINT *ErrOnReceive*(**SOutData** **out*, **TUINT** **PCI*)

Wird von *TransSR()* oder *TransRS()* aufgerufen, falls das Datenpaket *out* nicht korrekt empfangen werden konnte, der Aufruf *ReceiveData(out, PCI)* also einen Fehlercode zurückliefert. Grundsätzlich gilt auch hier alles, was bereits bei der Beschreibung von *Err-OnSend*() gesagt wurde. Wird RECOVERY_SUCCESS zurückgegeben, so müssen **out* und **PCI* natürlich die empfangene Nachricht bzw. die Protokollinformation für Schicht 3 enthalten.

TUINT *ErrAppTimeOut*(**SInData** **in*)

Wird von *TransSR()* aufgerufen, wenn das übergebene Kommando zwar korrekt an den Controller übertragen werden konnte, aber innerhalb eines festgesetzen Intervalls keine Quittung empfangen wurde. Das erste Byte von **in* codiert das übertragene Kommando, so daß in Abhängigkeit von diesem Wert noch eine zusätzliche Zeitspanne gewartet werden kann. Auf Vorhandensein eines Zeichens im Emfpangsbuffer - in diesem Fall ENQ - kann mittels *BufferEmpty*() geprüft werden; es darf aber auf keinen Fall ein Zeichen ausgelesen werden, da sonst der nachfolgende Aufruf von *ReceiveData()* fehlschlägt!Mögliche Rückgabewerte von *ErrAppTimeOut()* sind RECOVERY_SUCCESS und RECOVERY_FAIL.

TINT *FatalError*(**TUINT** *command*, **TINT** *errorCode*)

Wird von den TROL API-Funktionen im Modul "send.c" aufgerufen, falls *TransSR()* einen Fehlercode zurlückliefert, ein ungültiges Datenformat empfangen wurde oder der empfangene BIOS-Returnwert auf einen Fehler während der Ausführung eines RAPL BIOS-Calls hinweist. Der TROL-Kommandocode der fehlgeschlagenen Transaktion (z.B. TROL_DEPART) wird in *command* übergeben, während *errorCode* einen TROL-(-32000) oder einen BIOS-Fehlercode (> -32000) enthält; siehe dazu auch 2.2.4. sowie Anhang B. Welche Aktionen im Fall eines fehlgeschlagenen BIOS-Aufrufs durchzuführen sind, wird von den Erfordernissen der Applikation und der Art des Fehlers abhängen; es kann unter Umständen durchaus angebracht sein, den Fehler einfach zu ignorieren (in diesem Fall sollte der Returnwert von F*atalError()* 0 sein) oder den Fehlercode *errCode* an den Aufrufer zurückzugeben. TROL-Fehlercodes weisen - mit wenigen Ausnahmen - auf schwere, nicht behebbare Kommunikationsfehler, evt. aber auch auf Programmfehler hin. Der Fehler kann entweder vom PCP-Server (z.B. TROL_ERR_UNKNOWN), von der rufenden API-Funktion selbst (z.B. TROL_ERR_BAD_RSIZE) oder von einer der untergeordneten Protokollschichten (TROL_ERR_COMM) erkannt worden sein. Ist letzteres der Fall, kann davon ausgegangen werden, daß der zuständige Error-Handler der Schicht 3 zwar aufgerufen wurde, dieser aber nicht in der Lage war, den Fehler zu beheben.

Erweiterung von TROL um eigene Kommandos

Wenn die bereits vorhandenen TROL-Kommandos auch für viele Anwendungen ausreichen, so wird sich doch sehr wahrscheinlich einmal die Notwendigkeit ergeben, TROL um neue Kommandos zu erweitern. Der erste Schritt sollte darin bestehen, einen geeigneten Record-Datentyp zu deklarieren, der das Kommando samt Parametern möglichst kompakt kodiert; die über die serielle Schnittstelle übertragenen Nutzdaten sind allesamt Instanzen solcher TROL-Records. Während 3.6.1. und 3.6.2. Hinweise zu diesem Thema enthalten, ist in 3.6.3. eine Anleitung zur Erstellung eigener Kommando-Handler enthalten. 3.6.4. schließlich gibt einen Überblick darüber, welche Änderungen in den verschiedenen Modulen vorgenommen werden müssen, sobald TROL um ein neues Kommando ergänzt werden soll.

Deklaration neuer TROL Datentypen

Das Layout der in "commands.h" deklarierten Datentypen (structs) wurde so gewählt, daß alle "non-char" Strukturkomponenten an Wortgrenzen beginnen (word-alignment). Diese Vorgangsweise wurde aus 2 Gründen gewählt:

- Beschleunigung der Haupspeicherzugriffe: für das Lesen oder Schreiben einer 16-Bit Variable, die nicht word-aligned ist, werden 2 Hauptspeicherzugriffe benötigt.

- Das Alignment der TROL-Datentypen stimmt - unabhängig von den Compileroptionen - auf dem PC und dem Controller immer überein.

Bei der Deklaration neuer TROL-Strukturtypen muß man stets in Erinnerung behalten, daß ihre Instanzen byteweise über die serielle Schnittstelle übertragen und beim Empfänger wieder zusammengesetzt werden müssen. Geht man bei der Deklaration mit entsprechender Sorgfalt vor, so reduziert sich die Arbeit des "Zusammensetzens" auf die Ermittlung des Datentypes (die Typinformation ist stets im ersten Byte enthalten) und die Anwendung des entsprechenden Typecast-Operators auf den Eingabebuffer. Folgende Punkte sollten daher besonders berücksichtigt werden:

- Die einzelnen Strukturkomponenten sollten, wenn notwendig durch Verwendung von Füllbytes, word-aligned sein; dabei ist allerdings die Verwendung von effektiv ungenutzen Füllbytes möglichst zu vermeiden, da diese ebenfalls über die Schnittstelle übertragen werden müssen.

- Das erste Byte eines Anforderungs-Paktets muß immer den Kommandocode enthalten; dieser liefert dem Empfänger die benötigte Typinformation.

- Das erste 16-Bit-Word eines Quittungs-Pakets ist für den Rückgabewert des BIOS bzw. für TROL-Fehlercodes reserviert und wird auf dem PC - als Returnwert der rufenden API-Funktion - dem Benutzer übergeben. TROL- und BIOS-Codes belegen normalerweise verschiedene Wertebereiche; sollte der Rückgabewert einer BIOS-Routine trotzdem in den für TROL-Codes reservierten Bereich (-32000) fallen, so wird dieser durch den TROL-Code TROL_RETVAL_RANGE ersetzt; der Benutzer kann den tatsächlichen BIOS-Returnwert anschließend mittels *GetStatus()* abfragen.

- Der von Arrays benötigte Speicherplatz muß in der Struktur selbst vorhanden sein; ein Array von maximal 6 floats ist daher als **float** floatArray[6] und nicht als **float** *floatArray zu deklarieren.

- Arrays variabler Länge sollten immer als letzte Strukturkomponente abgespeichert werden, damit nicht in jedem Fall alle Array-Elemente übertragen werden müssen.

Als Beispiel für einen TROL-Datentyp soll STROL_DEPART_DATA näher betrachtet werden:

```
typedef unsigned char TUCHAR;

typedef struct
{
   TUCHAR  code;
   TUCHAR  numDims_straight;
   float   vect[6];
}
STROL_DEPART_DATA;
```

Diese Struktur codiert das RAPL-Kommando Depart; dieses verlangt als Parameter einen 6-dimensionalen Displacementvektor (dieser bezieht sich auf das TOOL-Koordinatensystem) *vect* sowie ein zusätzliches Argument *straight*, welches angibt, ob die Bewegung des Roboterarms geradlinig oder im Joint-Interpolated-Modus erfolgen soll (siehe Anhang A). Das erste Byte der Struktur - die Komponente *code* - enthält immer den TROL-Kommandocode TROL_DEPART; dadurch ist der PCP-Server in der Lage, das erhaltene Datenpaket als Struktur vom Typ STROL_DEPART_DATA zu erkennen und in einen Aufruf der entsprechenden RAPL BIOS-Funktion umzusetzen. Obwohl ein vektorielles Displacement erlaubt ist, wird man den Roboterarm in den meisten Fällen nur entlang der Tool-X-Achse (*vect*[0]) verschieben wollen. Es ist daher sinnvoll, nur die tatsächlich benötigten Koordinaten und eine zusätzliche Strukturkomponente *numDims*, welche die Länge von *vect* enthält, zu übertragen. Würde man nun *numDims* als **TUCHAR** deklarieren und vor *vect* einfügen, wäre *vect* nicht mehr word-aligned. *numDims* hinter *vect* einzufügen, würde allerdings bedeuten, daß in jedem Fall alle 6 Koordinaten von *vect* übertragen werden müssen. Glücklicherweise benötigen aber sowohl *numDims* als auch *straight* jeweils nicht mehr als 4 Bit an Speicherplatz, d.h. sie können in einem einzigen Byte - in der obigen Deklaration die Komponente *numDims_straight* - codiert werden.

Kommandocodes und Transaktionsklassen

In "commands.h" muß sowohl ein für die Codierung des Kommandos geeigneter Record-Datentyp deklariert als auch ein eindeutiger Kommandocode vergeben werden (siehe 3.6.1). Jedes Kommando gehört zu genau einer Transaktionsklasse; die Klassenzugehörigkeit ergibt sich implizit aus dem numerischen Wert des Kommandocodes. Die Einführung von Transaktionsklassen dient vor allem der Übersichtlichkeit, macht aber auch einige Code-Optimierungen möglich. Folgende Transaktionsklassen werden unterstützt:

CAT1

Diese Klasse umfaßt Kommandos, die controllerseitig direkt in den Aufruf einer RAPL BIOS-Funktion umgesetzt werden, wie z.B. *Move*() oder *Ready*(). Die Quittungsdatenpakete dieser Klasse enthalten ausschließlich eine 16-Bit Integer. Wertebereich der Kommandocodes: 0 .. LAST_CAT1_CODE.

CAT2

Wie CAT1, mit dem Unterschied, daß die Quittungspakete beliebig lang sein können (z.B. alle Funktionen, die der Berechnung kinematischer Transformationen dienen: diese liefern das Ergebnis der Transformation in einem **float**-Vektor zurück).Wertebereich der Kommandocodes: 64 .. LAST_CAT2_CODE.

CAT3

Hier sind Kommandos enthalten, welche nicht direkt vom RAPL BIOS unterstützt werden (z.B. das im Befehlssatz des RAPL-Interpreters enthaltenen Kommando LOCK) und daher auf andere Weise implementiert werden müssen (z.B. durch Schreiben/Lesen der RAPL PARAMETER TABLE).
Wertebereich der Kommandocodes: 128 .. LAST_CAT3_CODE.

CAT4

Enthält TROL-spezifische Kommandos, wie z.B. *GetStatus()*.
Wertebereich der Kommandocodes: 192 .. LAST_CAT4_CODE.

Kommandocodes dienen controllerseitig zur Indizierung einer Sprungtabelle, welche die Adressen der zugeordneten Kommando-Handler im Controller-Memory enthält. Für jede der vier Transakationsklassen existiert eine eigene Sprungtabelle. Die in "commands.h" deklarierten Konstanten LAST_CAT*X*_CODE geben den größten Kommandocode innerhalb der Transaktionsklasse *X* und somit die Größe der jeweiligen Sprungtabelle minus 1 an. Bei der Deklaration neuer Kommandocodes ist darauf zu achten, daß diese Konstanten entsprechend geändert werden und daß zwischen den Codes einer Transaktionsklasse keine Lücken auftreten!

Erstellen von Kommando-Handlern

Wurde controllerseitig ein Anforderungspaket empfangen, so wird in Abhängigkeit vom Kommandocode eine Routine, die für die Dekodierung der Parameter, die Durchführung von Integritätsüberprüfungen, den Aufruf des BIOS und evt. auch das Erstellen des Quittungsdatenpaketes zuständig ist, aufgerufen. Diese Routinen nennen wir Kommando-Handler; ihre Einsprungadressen sind - wie in 3.6.2. bereits erwähnt - in Sprungtabellen gespeichert, wobei der Kommandocode als Index dient (die Initialisierung der Sprungtabellen erfolgt in *entrypoint()*). Welche Parameter an einen Handler übergeben werden, hängt davon ab, ob er zu CAT1 oder einer anderen Transaktionsklasse gehört:

static void CAT1-Handler(**const SOutData** **out*);
static void *Other-Handler* (**const SOutData** **out*, **SInData** **in*);

Die Länge des empfangenen Anforderungspakets ist in *out->numBytes*, ein Zeiger auf das Paket selbst in *out->data* enthalten; dieses enthält den Kommandocode sowie sämtliche Parameter. *in* hingegen zeigt auf eine Datenstruktur, in die vom Handler die Adresse des an den PC zu übertragende Quittungspakets sowie dessen Länge einzutragen sind. Folgende Punkte sind bei der Implementierung neuer Handler besonders zu beachten:

1. Handler der Transaktionsklassen CAT1 und CAT2 implementieren Kommandos, die direkt in den Aufruf einer einzigen RAPL BIOS-Funktion umgesetzt werden können. Bricht während der Ausführung eines solchen Kommandos die Verbindung zum PC ab, so ist manchmal notwendig, nach dem Wiederherstellen der Verbindung feststellen zu können, ob die BIOS-Funktion tatsächlich aufgerufen wurde und ob der Aufruf erfolgreich war (z.B. bei inkrementellen Roboterbewegungen - *"exactly once semantic*"). Aus diesem Grund müssen Handler dieser Transaktionsklassen Buch über die von ihnen bereits ausgeführten Aktionen führen. Dies erfolgt mit Hilfe zweier globaler Variablen: *BIOSCalled* gibt an, ob das BIOS tatsächlich aufgerufen wurde; ist dies der Fall, so enthält *lastErrorCode* den Returnwert der BIOS-Routine, andernfalls einen TROL-Fehlercode, der angibt, warum der Aufruf des BIOS nicht möglich war. Die Werte dieser Variablen können mit Hilfe der API-Funkion *GetStatus()* abgefragt werden.

2. CAT1-Handler erstellen das Quittungspaket nicht selbst, sondern überlassen dies der rufenden Routine *MsgReceived()*, welche die dazu notwendigen Informationen den globalen Variablen *BIOSCalled* und *lastErrorCode* entnimmt.

3. Handler der Transaktionsklassen CAT2 bis CAT4 erhalten einen zusätzlichen Paramter *in* vom Typ **SInData***; sie müssen in *in->data* einen Zeiger auf das Quittungspaket sowie in *in->numBytes* dessen Länge eintragen. Kleinere Quittungspakete (Länge SEND_BS) sollten nicht auf dem Heap, sondern in dem globalen Array *sendBuf* angelegt werden, um den mit der dynamischen Speicherallozierung verbundenen Overhead zu vermeiden.

4. CAT2-Handler müssen außerden sicherstellen, daß in den ersten 16-Bit des Quittungspakets zu transportierende BIOS-Returncodes (= Rückgabewert der rufenden API-Funktion auf dem PC) nicht imBereich -32000 liegen. Sollte dieser Fall eintreten, so ist, um die Verwechslung mit einem TROL-Fehlercode zu verhindern, in der Quittung statt des BIOS-Returncodes der Wert TROL_RETVAL_RANGE einzutragen (siehe auch 3.6.1.).

5. Nach der Übertragung der Quittung wird von der Schicht 3 die Funktion *TransactionDone(***SInData** **in)* aufgerufen, um der Schicht 4 die Möglichkeit zu geben, für das Quittungs-Datenpaket dynamisch allozierten Speicher wieder freizugeben. *TransactionDone()* erkennt, wenn *in->data* auf das globale Array *sendBuf* verweist, und führt in diesem Fall keine Aktion aus.

Modifikation der betroffenen Source-Dateien

Dieses Unterkapitel beschreibt, welche Veränderungen in den einzelnen Source-Modulen vorgenommen werden müssen, sobald ein neues TROL-Kommando implementiert werden soll:

1. Auswahl der Transaktionsklasse *X* und des Kommandocodes sowie Deklaration eines geeigneten Kommando- und ggf. auch Quittungsdatentyps in "commands.h" . Sollte es notwendig sein, zusätzliche TROL-Fehlercodes einzuführen, so müssen diese ebenfalls in "commands.h" deklariert werden. Die Konstante LAST_CAT*X*_CODE ist entsprechend zu modifizieren (siehe 3.6.2).

2. Definition einer API-Funktion in "send.c" sowie Deklaration des zugehörigen Funktionsprototyps in "send.h". Die Aufgabe der API-Funktionen besteht darin, die übergebenen Parameter zu überprüfen, eine Instanz des Kommando-Datentyps zu erstellen, diese an den Controller zu übertragen und schließlich das Quittungs-Datenpaket entgegenzunehmen und zu dekodieren. CAT1-Transaktionen steht die in "send. .h" definierte Funktion *Cat1Transaction()* zur Verfügung, die große Teile der Fehlererkennung und -behandlung übernimmt.

3. Definition eines Kommando-Handlers in "pcptrol.c". Weiters muß die Adresse des Handlers innerhalb der Funktion *entrypoint()* in die entsprechende Sprungtabelle eingetragen werden:
   dispatchTableX[COMMAND_CODE - *X* *64] = COMMAND_HANDLER.

4. Verwendet das neue Kommando noch nicht implementierte BIOS-Calls, so müssen diese in "bioscall.c" definiert und ihre Prototypen in "bioscall.h" bereitgestellt werden.

5. Die in "callback.c" enthaltenen Error-Handler, insbesondere *FatalError()*, sind, wenn durch die Ausführung des Kommandos neue, bisher noch nicht abgedeckte Fehlerbedingungen auftreten können, ebenfalls zu modifizieren.

Portierung auf andere Plattformen

TROL wurde für Intel-basierte 16-bit-Systeme entwickelt. Bei der Portierung auf andere Plattformen sind folgende Punkte zu beachten.

- Sollte das Zielsystem eine Wordgröße von 32 bit oder größer verwenden, so müssen die Integer-Typdefinitionen in "troltype.h" dergestalt ergänzt werden (bedingte Kompilation), daß sich der Speicherbedarf für Instanzen der dort defnierten Integer-Datentypen nicht ändert. Die Anforderungen sind in der folgenden Übersicht zusammengefaßt:

TLONG	...	signed,	32 bit
TULONG	...	unsigned,	32 bit
TINT	...	signed,	16 bit
TUINT	...	unsigned,	16 bit
TBOOL	...	unsigned,	16 bit
TCHAR	...	signed,	8 bit
TUCHAR	...	unsigned,	8 bit

- Weiters muß bei den Compileroptionen das Strukturalignment entweder ausgeschaltet oder auf 16 bit - keinesfalls aber auf einen größeren Wert! - gesetzt werden.

- Der Zielprozessor muß über eine "Little Endian"-Architektur verfügen (z.B. Intel, VAX), d.h. die Adresse des niederwertigsten Bytes eines Speicherwortes ist gleich der Adresse des Speicherwortes.

Grundlagen der Roboterprogrammierung

Das Roboter-Koordinatensystem

Die Position des Roboterarms innerhalb des Arbeitsbereichs läßt sich in **kartesischen Koordinaten** durch ein 6-Tupel <X, Y, Z, YAW, PITCH, ROLL> beschreiben. Die ersten 3 Koordinaten <X, Y, Z> legen die Position des sogenannten **TCP** (*tool centre point:* Effektorposition) im Arbeitsbereich des Roboters fest, wohingegen die drei Winkel <YAW, PITCH, ROLL> die Orientierung des Tools (z.B. eines Greifers) angeben; das zugrundeliegende **Welt-Koordinatensystem** hat seinen Ursprung im Sockel des Roboters (siehe Abbildung 4).

Von diesem zu unterscheiden ist das **Tool-Koordinatensystem**, welches seinen Ursprung im **TCP** hat und sich mit diesem bewegt. Der **TCP** befindet sich normalerweise im Zentrum der Flansch am Ende des letzten Armsegments, wobei die Tool-X-Achse orthogonal zur Flansch - also in Richtung des letzten Armsegments - verläuft (siehe Abbildung 3). Nach der Ausführung eines absoluten Bewegungskommandos stimmt die Orientierung des **Tool-Koordinatensystems** mit der Orientierung der anzufahrenden **Location** - welche durch <YAW, PITCH, ROLL> festgelegt ist - überein.

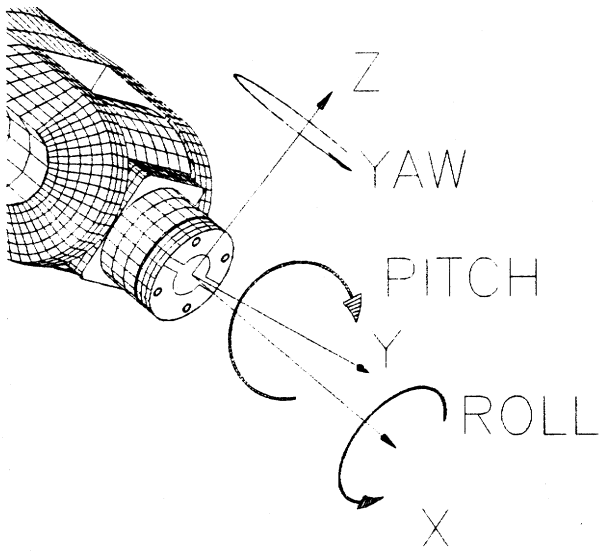


Abbildung 3. Tool-Koordinatensystem des A465.

Locations

Locations sind Variablen, welche die Koordinaten eines Punktes im Arbeitsbereich des Roboters enthalten. Sie sind als Vektoren von 8 **float** oder **long** Werten realisiert, wobei die letzen beiden Vektorelemente allerdings nur benötigt werden, wenn der A465 als Portalroboter[7] eingesetzt werden soll. **Locations** können in einem von 3 Koordinatensystemen vorliegen:

- **Kartesische Koordinaten**
  Diese werden als 6 dimensionaler **float**-Vektor gespeichert. Die ersten 3 Elemente werden in mm, die restlichen 3 im Radialmaß angegeben.

- **Precision Point-** oder **Motor-Koordinaten**
  An jeder der 6-Achsen des A465 ist ein inkrementeller optischer Enkoder angebracht, mit dessen Hilfe die Auslenkung jeder Achse aus ihrer Null-Position ermittelt wird. Die Anzahl der Enkoder-Impulse für alle 6 Achsen wird als 6-dimensionaler Vektor vom Typ **TLONG** (4 byte integer) abgespeichert.

- **Joint- oder Gelenkachsen- Koordinaten**
  Diese geben die Auslenkung der einzelnen Gelenksachsen im Radialmaß an. Betrachtet man z.B. Achse 3 in Abbildung 5, so sieht man, daß die Auslenkung (der Gelenkwinkel) dieser Achse 45 Grad beträgt (da die beiden durch diese Achse verbundenen Armsegmente einen Winkel von 45 Grad einschließen). **Joint-Koordinaten** werden als 6-dimensionaler **float**-Vektor gespeichert.

Locations können entweder am PC oder im Speicher des Robotercontrollers (in der sogenannten **Location Table**) abgelegt werden. Das **TROL**-API enthält keine Funktion, die es gestattet, die aktuelle Position des Greifers im Controller Memory abzuspeichern; allerdings können im Controller Memory bereits vorhandene **Locations** von den beiden API-Funktionen *TROLMove()* und *TROLAppro()* verwendet werden, sofern deren Namen bekannt sind.

Um **Locations** im Controller Memory abzuspeichern, kann z.B. das **RAPL**-Kommando "HERE" verwendet werden**;** dieses Kommando ermittelt die aktuelle Position und Orientierung des Greifers und trägt sie unter einem vom Benutzer anzugebenden Namen in die **Location Table** ein. Dabei ist zu beachten, daß **Locations**, deren Name mit einem Buchstaben beginnt, **kartesische Koordinaten** und **Locations**, deren Name mit dem Sonderzeichen "**#**" beginnt, **Precision Points** enthalten. Es ist nicht möglich, im Controller Memory **Locations** im **Joint-Koordinatensystem** abzuspeichern.

---

[7]Der Sockel eines Portalroboters ist - ähnlich einem Plotter - auf einer Schiene angebracht, wodurch er sich in der XY-Ebene bewegen läßt.

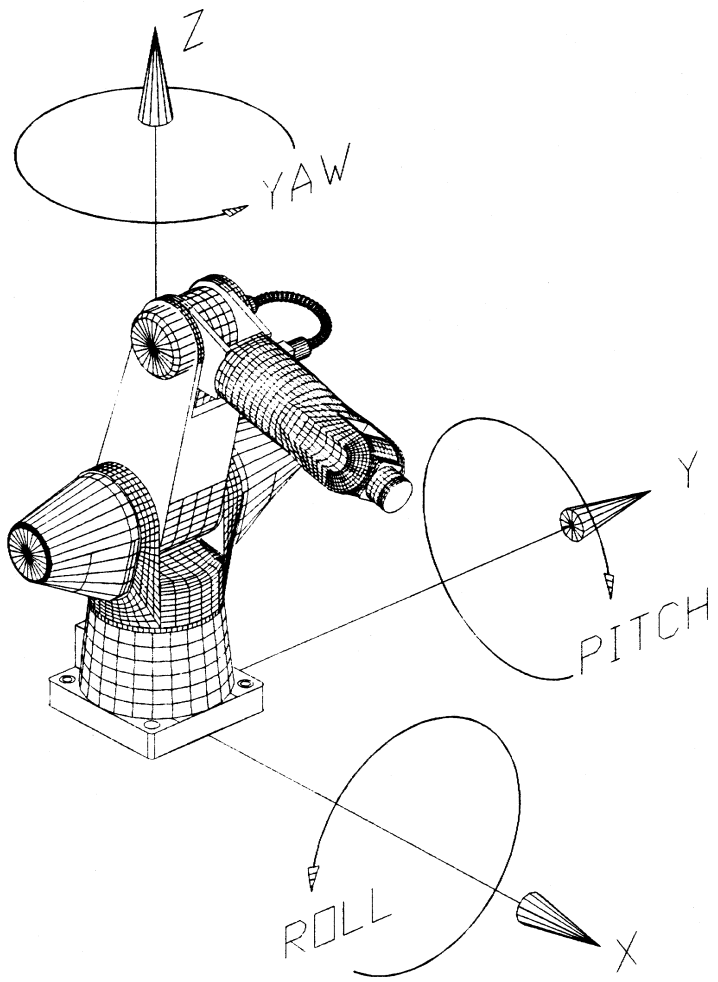


Abbildung 4. Welt-Koordinatensystem des A465.

Location-Namen enthalten bis zu 8 Zeichen, wobei das erste Zeichen ein Buchstabe oder ein "#" sein muß; die nachfolgenden Zeichen können Buchstaben, Ziffern oder Underscores "_" sein; TROL API-Funktionen, die **Location-Namen** als Parameter verlangen, erwarten ein

char-Array der Länge 8 (muß nicht NULL-terminiert sein), so daß bei kürzeren Namen entsprechend mit Leerzeichen aufgefüllt werden muß.

TROL stellt zur Ermittlung der aktuellen Position und Orientierung des Greifers die Funktion *TROLGetPosition()* zur Verfügung; es bleibt dem Benutzer überlassen, die solcherart gewonnenen **Location**-Daten - in Abhängigkeit von den Erfordernissen seiner Applikation - entsprechend zu organisieren.

Bewegungskommandos

Die meisten Bewegungen können wahlweise im **Joint-Interpolated-Modus** (die Bewegungen der einzelnen Gelenke beginnen und enden gleichzeitig) oder im **Straight-Line-Modus** (der **TCP** bewegt sich auf einer Geraden an die angegebene Position) ausgeführt werden. Zu beachten ist, daß Bewegungskommandos asynchron abgesetzt werden, d.h. nach der Rückkehr einer API-Funktion, die eine Bewegung des Roboters initiiert hat, kann nicht davon ausgegangen werden, daß die Bewegung auch tatsächlich abgeschlossen wurde. Synchrones Verhalten kann jedoch mit Hilfe der Funktion *TROLFinish()* erzwungen werden.

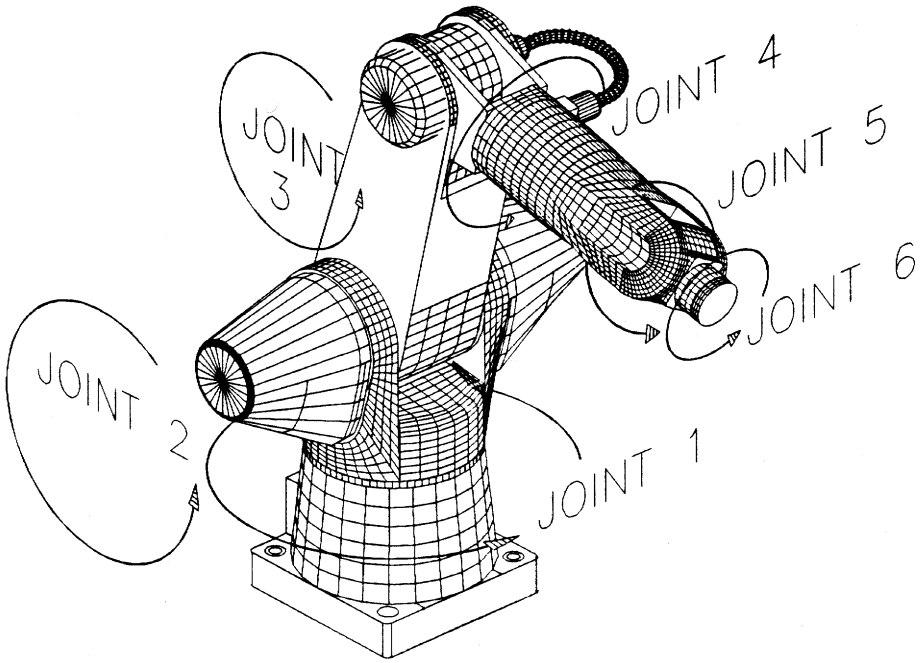


Abbildung 5. Die 6 Gelenke (joints) des A465.

TROL API

Zunächst werden zunächst einige im Zusammenhang mit API-Funktionen wichtige Datentypen vorgestellt und erklärt. In den darauffolgenden Abschnitten werden die bisher implementierten API-Funktionen - nach Transaktionsklassen unterteilt - beschrieben.

Verwendete Datentypen

Im folgenden wird die Datei "troltype.h", in der für die Parameterübergabe an API-Funktionen benötigte Datentypen deklariert werden, teilweise wiedergegeben.

```
#define POS_PREC    0           /* precision point (motor) location    */
#define POS_JOINT   1           /* joint (angular) location            */
#define POS_CART    2           /* cartesian location                  */

typedef struct
{
        TUCHAR      tag;        /* POS_PREC, POS_JOINT or POS_CART */
        TUCHAR      numDims;    /* number of relevant coordinates */
        union
        {       TLONG       prec[8];        /* motor location          */
                float       joint[8];       /* joint location          */
                float       cart[8];        /* cartesian location      */
        } data;
}
SLocData;

typedef TUCHAR TLocName[8];
```

Die meisten Bewegungskommandos benötigen eine Instanz des Typs **SLocData** als Eingangsparameter. Die Komponente *data* enthält - je nach dem Wert von *tag* - einen Satz von **Motor-**,**Joint**- oder **kartesischen Koordinaten**. In vielen Fällen wird man nicht alle Koordinaten benötigen; man kann daher in *numDims* angeben, wie viele Koordinaten tatsächlich an den Controller übertragen werden sollen. Soll z.B. das Kommando *Depart()* mit dem Vektor <10, 0, 0, 1.5, 0, 0> ausgeführt werden, so müssen nur die ersten vier Werte in *data.cart* eingetragen werden. Wird in *numDims* der Wert 4 übergeben, so werden die restlichen beiden Koordinaten am Controller automatisch auf 0.0 gesetzt.

Manche Bewegungskommandos akzeptieren einen Parameter *straight*: hat dieser einen Wert ungleich 0, so erfolgt die Bewegung im **Straight-Line-Modus**, ansonsten im **Joint-Interpolated-Modus**.

API der Transaktionsklasse CAT1

Allgemeine Rückgabewerte

Die möglichen Rückgabewerte dieser Funktionen umfassen die allgemeinen Fehlercodes der Protokollschicht 4, die speziellen Fehlercodes für CAT1- und CAT2-Transaktionen sowie Returncodes des RAPL-BIOS (siehe Kapitel 6). Rückgabewerte > -32000 sind BIOS-Returncodes, alle kleineren Werte TROL-Fehlercodes. Der Rückgabewert 0 zeigt in jedem Fall eine fehlerfreie Ausführung des Kommandos an.

TINT *TROLAppro*(**TLocName** *locName*, **SLocData** **pDisp,* **TUINT** *straight*)

Bewegt den TCP an einen Punkt, der bezüglich des Tool-Koordinatensystems um den Vektor *pDisp->data.cart* von der Location *locName* entfernt ist. Existiert die angegebene Location nicht, so wird der Fehlercode TROL_ERR_LOCATION zurückgegeben. *pDisp* muß auf eine kartesiche Location verweisen (*pDisp→tag* == POS_CART). Das folgende Beispiel zeigt, wie ein Objekt in einem beengten Arbeitsbereich, der eine Annäherung des Arms aus Richtung der Tool-X-Achse erforderlich macht, unter Vermeidung von Kollisionen aufgenommen werden kann:

```
SLocData loc;

loc.tag = POS_CART;loc.data.cart[0] = -90.0;
loc.numDims = 1;
TROLAppro("test_001", &loc, 1);
        /* Annäherung entlang der Tool- X Achse bis auf  9 cm */
        /* geradlinige Bewegung */
TROLMove("test_001", 1);
        /* Bewege den TCP geradlinig an die Objektposition */
TROLFinish(FINISH_WAIT);
        /* Warte, bis Bewegung abgeschlossen ist */
TROLClose(60);
        /* Greife Objekt */
delay(2000);
        /* Warte, bis Greifer geschlossen ist */
TROLDepart(&loc);
        /* Bewegung auf der Tool-X-Achse um 9 cm zurück */
```

TINT *TROLApproII*(**SLocData** *pTarget, **SLocData** **pDisp,* **TUINT** *straight*)

Wie *TROLAppro()*, allerdings wird hier die anzufahrende Location direkt in *pTarget* übergeben. *pTarget* muß auf eine kartesische oder eine Motor-Location verweisen.

TINT *TROLClose*(**TUINT** *perForce*)

Schließt den Greifer mit *perForce%* der Maximalkraft; gültige Werte für *perForce* liegen im Bereich 1 .. 100. Bevor der Greifer geschlossen wird, sollte sichergestellt sein, daß sich der Roboter tatsächlich an der gewünschten Position befindet (siehe *Finish()*).

TINT *TROLDepart*(**SLocData** **pDisp*, **TUINT** straight)

Bewegt den TCP an einen Punkt, der bezüglich des Tool-Koordinatensystems um den Vektor *pDisp->data.cart* von der aktuellen Location entfernt ist. *pDisp* muß auf eine kartesische Location verweisen. Beispiel: siehe *Appro()*.

TINT *TROLFinish*(**TUINT** *mode*)

Wird in *mode* die Konstante FINSISH_STATUS übergeben, so liefert diese Funktion 0 zurück, wenn alle Bewegungen des Roboterarms bereits abgeschlossen wurden, sonst eine 1. Wird allerdings die Konstante FINISH_WAIT übergeben, so wartet diese Funktion, bis alle Bewegungen abgeschlossen wurden; dabei ist zu beachten, daß, wenn die Bewegung sehr lange dauert - also über längere Zeit keine Rückmeldung vom Controller erfolgt-, in *TransSR()* (Schicht 3) ein Timeout ausgelöst wird. Um dies zu verhindern, muß der Error-Handler *ErrAppTimeout()* entsprechend modifiziert werden. Normalerweise führt der Controller zwischen zwei aufeinanderfolgenden Bewegungskommandos ein implizites *wait* aus; der explizite Aufruf von *Finish*(FINISH_WAIT) ist nur bei Verwendung der Befehle zur Steuerung des Greifers sowie des *Motor()*-Kommandos notwendig.

Beispiel: siehe *TROLAppro()*.

TINT *TROLGrip*(**TUINT** *dist*)

Setzt die Distanz zwischen den beiden Fingern des Greifers auf *dist* mm. Diese Kommando arbeitet immer mit 100%iger Kraft und sollte deshalb ausschließlich zur Positionierung der Finger, nicht aber zum Greifen eines Objekts verwendet werden! Gültige Werte für *dist* liegen im Bereich 0 .. 50 mm (2 inches).

TINT *TROLHalt*(**void**)

Bricht alle laufenden Bewegungsvorgänge ab.

TINT *TROLHome*(**TUINT** *axis*)

Homen des Roboters. In *axis* kann entweder die Nummer der zu "homenden" Achse (1..6) oder 0 übergeben werden, in welchem Fall alle Achsen "gehomt" werden. Beim "Homen" fährt der Roboterarm die sogenannten **home switches** an um seine Initialposition zu finden, relativ zu welcher intern alle absoluten Positionen (locations) berechnet werden. Diese Funktion muß nach Einschalten des Controllers aufgerufen werden, bevor ein Bewegungskommando abgesetzt wird!

TINT *TROLJog*(**SLocData** **pD*isp, **TUINT** straight)

Inkrementelle Bewegung des TCP bezüglich des Weltkoordinatensystems; die Orientierung der Tool-Achse wird dabei wenn möglich nicht verändert. Die ersten 3 Elemente von *pDisp→data.vect* geben die Deltas in X-, Y- und Z-Richtung an, die restlichen Elemente werden nicht verwendet. *pDisp* muß auf eine kartesiche Location verweisen, gültige Werte für *pDisp→numDims* liegen im Bereich 1 .. 3.

TINT *TROLJoint*(**TUINT** *jointNum*, **float** *radians*)

Inkrementelle Bewegung der Achse *jointNum* um *radians* Einheiten im Radialmaß. Folgende Achsen werden unterstützt: 1. Waist, 2. Shoulder, 3. Elbow, 4. (Euler-)Wrist, 5. Wrist Pitch, 6. Wrist Roll; siehe hierzu auch

TINT *TROLMA*(**SLocData** **pTarget*)

Bewegt den TCP an jenen Punkt, der durch die in *pTarget->data.joint* übergebenen absoluten Gelenkwinkel festgelegt ist. *pTarget→joints*[i-1] enthält den Wert für die Achse i. *pTarget* muß auf eine Joint Location verweisen.

TINT *TROLMI*(**SLocData** **pDisp*)

Inkrementelle Bewegung aller Achsen um die in *pDisp→data.joint* übergebenen Winkel. *pDisp→data.joint*[i-1] enthält das Inkrement für die Achse i. *pDisp* muß auf eine Joint Location verweisen.

TINT *TROLMove*(**TLocName** *locName*, **TUINT** *straight*)

Bewegt den TCP an die Position *locName*. Existiert diese Location nicht, so wird der Fehlercode TROL_ERR_LOCATION zurückgegeben.

TINT *TROLMoveII*(**SLocData** **pTarget*, **TUINT** *straight*)

Wie *Move()*, allerdings wird die anzufahrende Location direkt in *pTarget* übergeben. *pTarget* muß auf eine Motor- oder eine kartesische Location verweisen.

TINT *TROLOpen*(**TUINT** *perForce*)

Öffnet den Greifer mit *perForce%* der Maximalkraft; gültige Werte für *perForce* liegen im Bereich 1 .. 100. Bevor der Greifer geöffnet wird, sollte sichergestellt sein, daß sich der Roboter tatsächlich an der gewünschten Position befindet (siehe *Finish()*).

Beispiel: siehe *TROLAppro()*.

TINT *TROLReady*(**void**)

Bewegt den Roboter in die Ready-Position.

TINT *TROLSetToolTransform*(**SLocData** **md*)

Dieses Kommando teilt dem Controller mit, wo, relativ zum Ursprung des Tool-Koordinatensystems, sich die Toolspitze (z.B. Greiferspitze) befindet. Alle absoluten Bewegungskommandos beziehen sich nach Aufruf dieses Kommandos auf den neuen, also durch *md*, festgelegten TCP! *md* muß auf eine (6DOF) kartesiche Locatation verweisen.

TINT *TROLSpeed*(**TUINT** *perMaxSpeed*)

Setzt die Geschwindigkeit für alle nachfolgenden Bewegungskommandos auf *perMaxSpeed%* der Maximalgeschwindigkeit; gültige Werte für *perMaxSpeed* liegen im Bereich 1 .. 150.

API der Transaktionsklasse CAT2

Diese Transaktionsklasse enthält im Moment noch keine Kommandos.

API der Transaktionsklasse CAT3

Allgemeine Rückgabewerte

Die möglichen Rückgabewerte dieser Funktionen umfassen die allgemeinen Fehlercodes der Protokollschicht 4**.**

TINT *TROLGetPosition*(**SLocData** **pos*, **TINT** *mode*)

Mit diesem Kommando kann die momentane Position und Orientierung des Greifers abgefragt und in einem beliebigen der drei zu Verfügung stehenden Koordinatensysteme zurückgeliefert werden. Das gewünschte Koordinatensystem muß vor dem Aufruf in *pos->tag* eingetragen werden (POS_JOINT, POS_CART oder POS_PREC). Das Flag *mode* gibt an, ob die von der Firmware berechnete und vorgegebene Position (POS_COMMANDED) oder die tatsächlich erreichte Position (POS_ACT) zurückgeliefert werden soll; diese beiden Werte sind niemals identisch (da der Controller permanent versucht, die aktuelle auf die vorgegebene Position einzuregeln, ist der Arm ständig - wenn auch für den Benutzer nicht immer wahrnehmbar - in Bewegung), unterscheiden sich jedoch kaum, solange nicht gerade ein Bewegungskommando abgearbeitet wird.

API der Transaktionsklasse CAT4

Allgemeine Rückgabewerte

Die möglichen Rückgabewerte dieser Funktionen umfassen die allgemeinen Fehlercodes der Protokollschicht 4**.**

TINT *TROLTerminatePCP*(**void**)

Terminiert den PCP-Server im Controller. Diese Funktion gibt, sofern kein Übertragungsfehler auftritt, 0 zurück.

TINT *TROLGetStatus*(**int** **lastBIOSRetVal*)

Mit Hilfe dieser Funktion läßt sich festellen, ob der letzte Aufruf einer CAT1- oder CAT2-Transaktion tatsächlich zum Aufruf der entsprechenden Funktion des RAPL-BIOS geführt hat; in diesem Fall ist der Returnwert des BIOS in *lastBIOSRetVal* enthalten. *GetStatus()* kann auch nach Abbruch der Verbindung zum PCP-Server und neuerlicher Aktivierung des PCPs aufgerufen werden; sollte während der Ausführung einer CAT1- oder CAT2 Transaktion mit "*exactly once semantic*" (z.B. relative Bewegungskommandos wie *Joint()*) die Verbindung zum PCP-Server abbrechen, so kann nach Wiederherstellen der Verbindung anhand der von *GetStatus()* gelieferten Information entschieden werden, ob das Kommando nochmals ausgeführt werden soll oder nicht.

Returncode	lastBiosRetVal
0 (BIOS wurde nicht aufgerufen)	TROL-Fehlercode, der angibt, warum der Aufruf nicht möglich war.
1 (BIOS wurde aufgerufen)	BIOS-Returncode.

Andere Funktionen

In diesem Abschnitt werden Funktionen, die zwar nicht Bestandteil des TROL-API sind, aber für die Initialisierung und Steuerung der seriellen Kommunikation benötigt werden, behandelt.

TUINT *InitTROL*(**TUINT** *serOnly*)

Diese Funktion initialisiert die in "trol.cfg" angegebene serielle Schnittstelle und überprüft, ob das serielle Kommunikationsprotokoll erfolgreich installiert werden konnte. Hat der Parameter *serOnly* einen Wert ungleich 0, so wird außerdem das CRS-proprietäre ACI-Protokoll am Controller ausgeschaltet und der PCP-Server im Controller-Memory aktiviert; dies sollte jedoch nicht innerhalb des Anwendungsprogrammes selbst geschehen, sondern mittels des Utilites "inittrol.exe". Anwendungsprogramme sollten *InitTROL*(1) zu Beginn, d.h. vor jeder anderen TROL-API-Funktion aufrufen.

TUINT *CleanupTROL*(**void**)

Diese Funktion deaktiviert den PCP-Server (mittels *TerminatePCP()*). Solange der PCP-Server nicht terminiert wurde, ist es nicht möglich, mit dem Robotercontroller im ACI- oder Terminal-Modus zu kommunizieren! Mögliche Returnwerte sind INITTROL_SUCCESS und INITTROL_FAILURE. Diese Funktion sollte nicht direkt, sondern über das Utility "cluptrol.exe" aufgerufen werden.

void SetL3_TO_Intervall(**TUINT** TO_Intervall)

Mit dieser Funktion kann die Zeitspanne in 1/10 s festgelegt werden, welche die Schicht 3 nach der Übertragung eines Kommandos an den Controller auf das Quittungspaket wartet, bevor ein Timeout erkannt und der Error-Handler *ErrAppTimeout()* aufgerufen wird.

Rückgabewerte und Fehlercodes

Fehlercodes der Schicht 4

Diese Codes werden von den TROL API-Funktionen zurückgegeben. Liefern einer der Transaktionsklassen CAT1 oder CAT2 zugeordnete API-Funktionen einen Wert > -32000, so handelt es sich dabei um den Rückgabewert der am Controller aufgerufenen BIOS-Routine; BIOS-Retuncodes ungleich 0 zeigen dabei im allgemeinen ein Fehler an. Die genaue Bedeutung der BIOS-Codes kann dem Appendix I der technischen Dokumentation des A465 Roboters entnommen werden.

Welche Werte tatsächlich zurückgegeben werden können, hängt außer von der Transaktionsklasse auch noch vom Error-Handler *FatalError()* ab, der gewisse TROL- oder BIOS-Fehlercodes "herausfiltern" kann.

Allgemeine Fehlercodes

Codes können in jeder der vier Transaktionsklassen auftreten.

TROL_ERR_COMM -32001
Ein Kommunikationsfehler ist aufgetreten; der entsprechende Error-Handler wurde zwar aufgerufen, konnte den Fehler aber nicht beheben.

TROL_ERR_BAD_RSIZE -32002
Das vom PC empfangene Quittungspaket hat nicht die erwartete Größe.

TROL_ERR_ARG_RANGE -32003
Beim Aufruf einer API-Funktion wurden ein odere mehrere ungültige Parameter übergeben.

TROL_ERR_UNKNOWN -32051
Der vom Controller emfpangene Kommandocode ist nicht bekannt.

TROL_ERR_BAD_CSIZE -32052
Das vom Controller empfangene Anforderungspaket hat nicht die erwartete Größe.

Fehlercodes für Transaktionen der Kategorien CAT1 und CAT2

TROL_RETVAL_RANGE -32101
Der Returnwert der aufgerufenen BIOS-Funktion liegt im für TROL-Codes reservierten Bereich <= -32000; der tatsächliche BIOS-Returncode kann anschließend mit Hilfe der API-Funktion *GetStatus()* abgefragt werden.

TROL_ERR_LOCATION -32102
Der beim Aufruf des Kommandos *Move()* oder *Approach()* übergebene Location-Name existiert nicht.

Fehlercodes der Schicht 3

Dies sind die Rückgabewerte von *TransSR()* und *TransRS()*.

TRANS_SUCCESS 0x00
Die Transaktion wurde erfolgreich abgeschlossen.

TRANS_ERR_SEND 0x01
Während der Ausführung von *SendData()* ist ein Fehler aufgetreten; der Error-Handler *ErrOnSend()* wurde aufgerufen, konnte das Problem aber nicht beseitigen.

TRANS_ERR_RECEIVE 0x02
Während der Ausführung von *ReceiveData()* ist ein Fehler aufgetreten; der Error-Handler *ErrOnReceive()* wurde aufgerufen, konnte das Problem aber nicht beseitigen.

TRANS_TIMEOUT 0x03
Wird von *TransSR()* zurückgegeben, wenn das Quittungspaket des Controllers nicht innterhalb von L3_TO_Intervall 1/10 s empfangen wurde und der Error-Hander *ErrAppTimeout()* das Problem nicht beseitigen konnte. Die Standardversion von *ErrAppTimeout()* wartet beim Kommando *Home()* zusätzlich 90 s und beim Kommando *Finish*(FINISH_WAIT) unendlich lange auf das Quittungspaket. Der Wert von L3_TO_Intervall kann mit Hilfe der Funktion *SetL3_TO_Intvervall()* auf einen neuen Wert gesetzt werden; die Voreinstellung beträgt 15 s.

Fehlercodes der Schicht 2

Die Fehlercodes der Sicherungsschicht untereilen sich in 3 Gruppen: NAK-Codes, Rückgabewerte von *SendData()* und Rückgabewerte von *ReceiveData()*.

NAK-Codes

Fehler, die zu einem NAK führen, werden ausschließlich vom Empfänger erkannt. NAK-Codes unterscheiden sich in zumindest 2 Bits voneinander, Codes verschiedener Untergruppen in zumindest 4 Bits.

Untergruppe 1 (Fatale Fehler)

Das Auftreten eines fatalen Fehlers führt zum Abbruch der Verbindung durch den Empfänger.

ERR_FID 0x01
Die empfangene Frame-ID stimmt nicht mit der erwarteten überein.

ERR_ETB 0x02
Es wurde ETB anstelle von ETX bzw. ETX anstelle von ETB empfangen.

ERR_TOO_LARGE 0x04
Es ist nicht genügend Speicherplatz am Heap vorhanden, um die zu übermittelnden Daten aufzunehmen.

Untergruppe 2

Erhält der Sender ein NAK gefolgt von einem dieser Codes, so wird er versuchen, den letzen Frame (bzw. den Header) nochmals zu übertragen; konnte die Übertragung nach MAX_RETRIES Versuchen nicht erfolgreich durchgeführt werden, wird die Verbindung abgebrochen, und sowohl *SendData()* als auch *ReceiveData()* geben den Fehlercode ERR_LIMIT_EXCEEDED zurück.

ERR_BCC 0xf1
Prüfsumme (Block Check Characters) ist falsch.

ERR_STX 0xf2
STX wurde erwartet, aber ein anderes Zeichen wurde empfangen.

ERR_ETX 0xf4
ETX wurde erwartet, aber ein anderes Zeichen wurde empfangen.

ERR_SOH 0xf8
SOH wurde erwartet, aber ein anderes Zeichen wurde empfangen.

Rückgabewerte

Die Rückgabewerte *von SendData()* und *ReceiveData()* sind vom Typ **TUINT** (unsigened int); bei deren Auswertung ist es notwendig, sie mit 0xff zu maskieren, da das Hi-Byte im Fall ERR_NAK und ERR_ABORT zusätzlich den Fehlercode enthält. Ein Rückgabewert von 0 zeigt eine erfolgreiche Ausführung der Übertragung an.

Konnte eine Übertragung nicht erfolgreich durchgeführt werden, so sollte eine Timeoutperiode (Empfänger) bzw. zwei Timeoutperioden (Sender) gewartet und anschließend der Empfangsbuffer geleert werden; dies entspricht einem Reset. Die Übertragung kann anschließend nochmals versucht werden.

Treten häufig Übtragungunsfehler auf, so sollte eine niedrigere Baudrate bzw. ein größeres Timeout-Intervall gewählt werden.

Rückgabewerte von SendData

ERR_WRONG_CCHAR 0x01
ACK oder NAK wurde erwartet, es wurde aber ein anderes Zeichen empfangen.

ERR_NAK 0x02
Abbruch der Verbindung durch den Empfänger; das Hi-Byte enthält den Fehlercode.

ERR_TIMEOUT_ENQ 0x03
Der Empfänger reagiert nicht auf die Sendeanfrage.

ERR_TIMEOUT_NAK 0x04
Das Timeout-Intervall zwischen dem Empfang von NAK und dem Empfang des Fehlercodes wurde überschritten.

ERR_TIMEOUT_ETX 0x05
Das Timeout-Intervall zwischen dem Senden von ETX / ETB und dem Empfang von ACK wurde überschritten.

ERR_LIMIT_EXCEEDED 0x07
Während der Übertragung eines einzelnen Frames sind wiederholt Fehler aufgetreten.

Rückgabewerte von ReceiveData

ERR_TIMEOUT_SOH 0x81
Das Timeout-Intervall zwischen dem Senden von ACK und dem Empfang von SOH wurde überschritten.

ERR_TIMEOUT_STX 0x82
Das Timeout-Intervall zwischen dem Senden von ACK und dem Empfang von STX wurde überschritten.

ERR_TIMEOUT_TERMC 0x83
Das Timeout-Intervall zwischen dem Empfang von SOH / STX und dem Empfang von ETB \ ETX wurde überschritten.

ERR_TIMEOUT_EOT 0x84
Das Timeout-Intervall zwischen dem Empfang von ETX und EOT wurde überschritten.

ERR_BAD_ENQ 0x85
ENQ wurde erwartet, es wurde aber ein anderes Zeichen empfangen.

ERR_BAD_EOT 0x86
EOT wurde erwartet, es wurde aber ein anderes Zeichen empfangen.

ERR_ABORT 0x80
Ein fataler Fehler ist aufgetreten; das Hi-Byte enthält den dem Sender übermittelten NAK-Code; der Sender terminiert nach dem Empfang dieses NAK-Codes mit dem Rückgabewert ERR_NAK.

ERR_LIMIT_EXCEEDED 0x07
Während der Übertragung eines einzelnen Frames sind wiederholt Fehler aufgetreten

Änderungen gegenüber frühren Versionen

Version 2.0

- Es können nun auch unter Windows 3.1 lauffähige TROL-Applikationen erstellt werden. Bei der Erstellung von Windows-Applikationen müssen im Project-File die beiden Konstanten WIN16 und DOS definiert werden. Weiters ist darauf zu achten, daß die Applikation für das Speichermodell "Large" erstellt und mit der Bibliothek "LIB\TROLLIB\WIN16\trol.lib" gelinkt wird.

- Alle Integer-Datentypen wurden durch mittels typedef definierte Datentypen ersetzt, z.B
  typedef unsigned long **TULONG**.
  Diese Deklarationen finden sind in der Datei "troltype.h"; ihr Zweck ist es, eine Portierung von TROL auf 32- oder 64-bit Plattformen zu erleichtern (siehe 34).

- Die Datentypen **SMoveData** und **SJointData**, die der Übergabe von Locations an verschiedene API-Funktionen dienten, wurden, um das API möglichst homogen und übersichtlich zu gestalten, durch den Datentyp **SLocData** ersetzt (siehe 50).

- Die API-Funktion *GetPosition()*, welche die aktuelle Position und Orientierung des Greifers - wahlweise in Motor-, Joint- oder kartesischen Koordinaten - ermittelt und in einer Variable vom Type **SLocData** zurückgibt, wurde hinzugefügt.

- Die beiden API-Funktionen *ApproII()* und *MoveII()*, die als Eingabe einen Zeiger auf eine Variable vom Type **SLocData** anstatt eines Location-Namens akzeptieren, wurden hinzugefügt. Diese beiden Funktionen machen es - im Verbund mit *GetPosition()* - möglich, gänzlich auf die im Controller Memory untergebrachte Location Table zu verzichten.

Version 2.1

- Es können nun auch 32-bit Windows (Windows 95, Windows NT)-Applikationen erstellt werden.

- Allen API-Funktionen wurde das Präfix TROL vorangestellt (z.B. *TROLMove*() statt *Move*()).

- Alle Codeteile, die bedingt kompiliert werden, wurden überarbeitet. In den Projekt-Files muß (und darf!) nur mehr ein Plattform-Bezeichner definiert werden, also entweder DOS oder WIN16 oder WIN32.