Einbindung des Motorcontrollers C-842 in das *XCTL*-System

Studienarbeit

David Damm
Im.-Nr. 157424

Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Klaus Bothe

9. November 2006

Inhaltsverzeichnis

1 Einleitung **7**

1.1 Einführung in das *XCTL*-System 7

1.2 Aufgabenstellung 7

2 Einbinden des Motorcontrollers C-842 **8**

2.1 Überblick 8

2.1.1 Motorcontroller 8

2.1.2 Treiber 8

2.2 Analyse und Design 9

2.2.1 Klassenstruktur 9

2.2.1.1 `TMotor` 9

2.2.1.2 `TMSettings` 9

2.2.1.3 `TMList` 10

2.2.1.4 `Controller` 10

2.2.1.5 `ControllerList` 10

2.2.2 Funktionalität der Treiber 11

2.2.3 Strategie 11

2.2.4 Änderungen an Dateien 11

2.2.4.1 `HWIO.H` 12

2.2.4.2 `HWIO.CPP` 12

2.2.4.3 `hwguids.h` 12

2.2.4.4 `M_LAYER.H` 12

2.2.4.5 `M_MOTHW.H` 13

2.2.4.6 `motorcontroller.h` 13

2.2.4.7 `MOTORS.CPP` 13

2.2.4.8 `MSIMSTAT.CPP` 14

2.2.4.9 `M_LAYER.CPP` 14

2.2.5 Vereinfachung der Strategie 14

3 Installation und Test **15**

3.1 Inbetriebnahme . . . 15

3.1.1 Voraussetzungen . . . 15

3.1.2 Installation unter *Windows NT/2000* . . . 15

3.1.3 Speicherbereich einstellen . . . 16

3.2 Entwicklungsrechner . . . 16

3.2.1 Entwicklungsumgebung . . . 16

3.2.2 Test . . . 18

3.2.2.1 Testen während der Entwicklung . . . 18

3.2.2.2 Regressionstest mit *ATOS* . . . 19

3.3 Arbeitsrechner . . . 21

4 Fehlersuche und Problembehandlung **22**

4.1 Fehlverhalten . . . 22

4.2 Behebung von Fehlern . . . 22

5 Zusammenfassung **24**

5.1 Fazit . . . 24

5.2 Ausblick . . . 24

A Motor-Controller C-842/C-844 **25**

A.1 Motor-Controller C-842 . . . 25

A.1.1 Board Register . . . 25

A.1.1.1 Register Tabelle . . . 25

A.1.1.2 Status Register . . . 25

A.1.1.3 Function Register . . . 26

A.1.2 Output-Limit . . . 26

A.1.3 Trajectory Profile Generation . . . 26

A.1.3.1 S-curve Point to Point . . . 27

A.1.3.2 Trapezoidal Point to Point . . . 27

A.1.3.3 Velocity Contouring . . . 28

A.1.3.4 Electronic Gearing . . . 29

A.1.4 Digital Servo Filtering . . . 29

A.1.4.1 PID . . . 29

A.1.4.2 PIVff . . . 30

A.2 Motor-Controller C-844 . . . 31

A.3 Vergleich C-842 und C-844 . . . 31

A.4 Unterstützung für Windows 2000 . . . 31

B Vergleich der Funktion von C-832 und C-842 **32**

B.1 Motorfunktionen mit Hardwarezugriffen 32

B.2 Mutator/Accessor-Methoden 40

B.3 Funktionen zur Verwaltung und sonstige 41

B.4 Verwendung der Funktionen aus TC_832Controller 43

C LM628 User Command Set **46**

C.1 Initialisierung 46

C.1.1 RESET 46

C.1.2 DFH 46

C.2 Interrupts 47

C.2.1 MSKI 47

C.2.2 RSTI 47

C.2.3 SIP 48

C.3 Filter 48

C.3.1 LFIL 48

C.3.2 UDF 48

C.4 Trajektorie 48

C.4.1 LTRJ 48

C.4.2 STT 49

C.5 Report 50

C.5.1 RDSTAT 50

C.5.2 RDSIGS 50

C.5.3 RDIP 50

C.5.4 RDRP 51

C.5.5 RDDP 51

D Dokumentation des Quellcodes **52**

E Quellcode **68**

F Literatur **77**

Abbildungsverzeichnis

2.1 Klassendiagramm `TMotor` 9
2.2 Kollaborationsdiagramm `TMList` 10
2.3 Klassendiagramm `Controller` 10
2.4 Klassendiagramm `TC_832` und `TC_842` 12

3.1 Speicherbereich einstellen (DIP) 16
3.2 Entwicklungsrechner 17
3.3 Controller und Motoren 17
3.4 Entwicklungsumgebung (schematisch) 18
3.5 Controllerkarte C-842 19

A.1 S-curve Profil 27
A.2 Trapezoidal Profil 28
A.3 Velocity Contouring Profil 29

Tabellenverzeichnis

3.1 Beispiele für Schalterstellungen (DIP) 16

5.1 Analyse der Quellen/Code 24

A.1 Register Tabelle 25
A.2 Status Register 26
A.3 Function Register 26
A.4 Bewegungsprofile 27
A.5 Parameter zu S-curve Point to Point 27
A.6 Parameter zu Trapezoidal Point to Point 28
A.7 Parameter zu Velocity Contouring 28
A.8 Parameter zu Electronic Gearing 29
A.9 Parameter zu **PID** 29
A.10 Parameter zu **PIVff** 30
A.11 Übersicht C-842/C-844 31

B.1 `TC_832::ActivateDrive(void)` 32
B.2 `TC_832::ActivateFilterParameters(void)` 33
B.3 `TC_832::CheckBoardOk(void)` 33
B.4 `TC_832::ExecuteCmd(LPSTR)` 34
B.5 `TC_832::GetStatus(void)` 34
B.6 `TC_832::_GetFailure(long *failure)` 34
B.7 `TC_832::_GetPosition(long *position)` 35
B.8 `TC_832::IsIndexArrived(void)` 35
B.9 `TC_832::IsLimitHit(void)` 35
B.10 `TC_832::IsMoveFinish(void)` 36
B.11 `TC_832::MoveAbsolute(long pos)` 36
B.12 `TC_832::MoveRelative(long position)` 37
B.13 `TC_832::Reset(void)` 37

B.14 `TC_832::SetAcceleration(DWORD acceleration)` 38
B.15 `TC_832::SetVelocity(DWORD velocity)` 38
B.16 `TC_832::SetHome(void)` 38
B.17 `TC_832::StartToIndex(long &oldPos)` 39
B.18 `TC_832::StopDrive(BOOL bOffOn)` 39
B.19 `TC_832::Drive628(BYTE cmd, WORD ctrl_word, long param)` 40
B.20 `TC_832::GetAcceleration(void)` 40
B.21 `TC_832::GetVelocity(void)` 41
B.22 `TC_832::TC_832(void)` 41
B.23 `TC_832::SetLimit(DWORD limit)` 41
B.24 `TC_832::StartCheckScan(void)` 42
B.25 `TC_832::StartLimitWatch(void)` 42
B.26 `TC_832::StopLimitWatch(void)` 42
B.27 `TC_832::OptimizingDlg(void)` 42
B.28 `TC_832::SaveSettings(int LastSave)` 43

C.1 Status Register (Bedeutung der einzelnen Bits) 50
C.2 Signal Register (Bedeutung der einzelnen Bits) 51

Kapitel 1

Einleitung

1.1 Einführung in das *XCTL*-System

Das *XCTL*-Projekt ist eine Kooperation zwischen dem Institut für Physik und dem Institut für Informatik an der Humboldt-Universität zu Berlin. Am Institut für Physik sind verschiedene Labor-Messplätze zur Untersuchung von Halbleiterstrukturen installiert.
Ein solcher Messplatz wird durch das *XCT*L-Programm gesteuert. Dabei werden Messdaten erfasst und anschließend ausgewertet. Die zu untersuchenden Halbleiterstrukturen werden mit Röntgenstrahlen bestrahlt, die an den Kristallstrukturen von Halbleitern gestreut werden. Diese Strahlung kann mit unterschiedlichen Sensoren gemessen und zur Analyse der Kristalle verwendet werden.
Der Name *XCTL* leitet sich von der Röntgenstrahlung (engl. *X*-ray) und der Steuerung (engl. *C*on*T*ro*L*) ab.

Das Institut für Physik verfügt über mehrere Messplätze. Jeder dieser Messplätze ist mit einem eigenen PC ausgerüstet, auf dem die *XCTL*-Software installiert ist, wobei auf allen Rechnern die gleiche Software arbeitet. Die spezifischen Anwendungen (z.B. Topographie, Diffraktometrie) werden innerhalb des Programms realisert. Die Arbeitsumgebung wird durch entsprechende Modifikationen von Konfigurations- und Makrodateien an den jeweiligen Messplatz angepasst.

Das *XCTL*-Programm wird mittels *Visual C++* programmiert und läuft in der Arbeitsumgebung von *Windows NT/2000*.

1.2 Aufgabenstellung

Bisher unterstützt die Software zwei verschiedene Motorcontrollerkarten (C-812 und C-832), an die die Motoren zur Positionierung der Probe auf dem Messteller angeschlossen werden können. Da diese Karten aber schon relativ alt sind, mit der Zeit nicht mehr funktionieren, und es weder Unterstützung seitens der Firma *PhysikInstrumente* noch Ersatz für diese Karten gibt, ist es unbedingt notwendig auf einen neueren Typ umzusteigen.
Das Institut für Physik hat deswegen schon vor einigen Jahren neue Controllerkarten vom Typ C-842 angeschafft. Inzwischen gibt es auch dafür kaum noch eine Unterstützung seitens *Physikinstrumente*. Die mitgelieferte Firmware wird nicht mehr durch Updates aktualisiert.

Das Ziel dieser Studienarbeit ist es, durch eine definierte Vorgehensweise, die Unterstützung der Controllerkarte vom Typ C-842 in der *XCTL*-Software zu realisieren. Dabei soll ein Konzept entstehen, dass auch für weitere Einbindungen anderer Controllerkarten genutzt werden kann.

Die Software soll nach dem Einbinden der Motorcontrollerkarte C-842 die gleiche Funktionalität bieten wie mit den älteren Modellen (C-812 bzw. C-832).

Kapitel 2

Einbinden des Motorcontrollers C-842

2.1 Überblick

Zum gegenwärtigen Stand werden von der *XCTL*-Software die Motorcontroller C-812 und C-832 unterstützt. Für diese wurden jeweils eigene Treiber implementiert, da die Unterstützung von *PhysikInstrumente* mit der Portierung des Systems nach 32 Bit und nach *Windows NT/2000* nicht mehr gegeben war.

Die Zielstellung ist das Einbinden des Motorcontrollers C-842 und das Erstellen eines Schemas, um das Einbinden weiterer Motorcontroller zu ermöglichen (z.B. C-844).

2.1.1 Motorcontroller

Motorcontroller sind Karten, die in den PC gesteckt werden und der Ansteuerung von Motoren dienen. Dabei unterscheidet man je nach Typ der Karte, ob zwei oder sogar vier Motoren von einer Karte angesteuert und verwaltet werden können.

Die Software sendet Befehle an den Controller, diese werden in seinem Mikroprozessor verarbeitet und anschließend an die gewünschten Motoren weitergeleitet, so dass die entsprechende Bewegung von den Motoren ausgeführt wird.

Der Motorcontroller ist außerdem dafür zuständig, die angeschlossenen Motoren zu überwachen und deren Status an die Software zu übermitteln.

2.1.2 Treiber

Um die Kommunikation zwischen Controllerkarte und Software herzustellen, werden Treiber benötigt. Eine Möglichkeit ist, diese selbst zu erstellen (so wie bei C-812 und C-832). Dies hätte den Vorteil, dass man die Treiber den eigenen Wünschen (also z.B. dem Betriebssystem, der Entwicklungsumgebung) anpassen kann. Außerdem können die Treiber dann jederzeit erweitert werden und ermöglichen so stets ein aktuelles System. Nachteilig ist der relativ hohe Aufwand und die Einarbeitungszeit in die Programmierung von Treibern unter *Windows*. Um einen guten Einblick zu bekommen, ist die Arbeit von Harder/Paschold (siehe Literatur) empfehlenswert.

Alternativ bietet es sich im Falle des Motorcontrollers C-842 an, auf die mitgelieferten Treiber von *PhysikInstrumente* zurückzugreifen. Vorteilhaft ist zudem die Bereitstellung einer DLL mit über 100 Funktionen. Die Einarbeitungszeit fällt hier nahezu weg, da alle Funktionen der DLL (siehe Literatur: *PhysikInstrumente*) dokumentiert sind.

Der Zugriff auf die Bibliothek ermöglicht eine komfortable Arbeit mit dem System und vereinfacht die Integration des Motorcontrollers. Aus diesem Grund wird versucht, genau diesen Ansatz im Weiteren zu verfolgen.

2.2 Analyse und Design

2.2.1 Klassenstruktur

Im Folgenden soll die Struktur der schon vorhandenen Klassen daraufhin untersucht werden, ob Änderungen bzw. Neuimplementationen von Klassen benötigt werden, um den C-842 einzubinden. Dabei gilt es festzustellen, welche Abhängigkeiten der Klassen untereinander vorhanden sind, in die der Motorcontroller C-842 eingepasst werden muss. Schließlich soll der Programmier- und Wartungsaufwand so gering wie möglich gehalten werden.

2.2.1.1 `TMotor`

Alle Klassen, die Antriebe realisieren, erben von der Klasse `TMotor`. Sie bildet das Grundgerüst und stellt eine Vielzahl an Funktionen bereit. Es werden zahlreiche Mutator/-Acessor-Methoden implementiert, die zum Modifizieren und Zugreifen auf typische gekapselte Motorparameter (z.B. `AngleMin`, `AngleMax`, `MinSpeed`, `MaxSpeed`) verwendet werden. Deren Werte sind in der Regel in der `Hardware.ini` gespeichert (Einlesen beim Programmstart, Abspeichern beim Programmende).
Außerdem stehen Funktionen zum Anfahren einer neuen Motorposition zur Verfügung (z.B. `MoveByAngle`, `MoveByPosition`, `MoveToAngle`, `MoveToPosition`).
Zusätzlich gibt es virtuelle Funktionen (z.B. `IsMoveFinish`, `StopDrive`), die dann erst später innerhalb der speziellen Motorklasse implementiert werden, um den jeweiligen Unterschieden der Motoren (bzw. Controllern) zu genügen.

Die Klasse `TDC_Drive` ist wiederum eine Spezialisierung von der Klasse `TMotor` und enthält Funktionen, die für DC-Antriebe benötigt werden oder sinnvoll erscheinen.

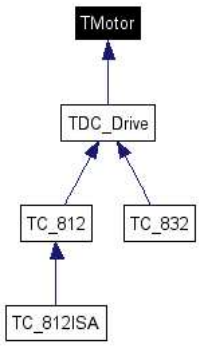


Abbildung 2.1: Klassendiagramm `TMotor`

2.2.1.2 `TMSettings`

Die Klasse `TMSettings` ist eine Struktur, die die Information von gewissen Größen für einen Antrieb festlegt. Zu diesen Größen zählen:

- `AngleMin` (minimale Winkelposition),
- `AngleMax` (maximale Winkelposition),
- `Angle` (aktuelle Winkelposition),
- `AngleWidth` (Schrittweite),
- `Speed` (Geschwindigkeit).

2.2.1.3 `TMList`

Die Klasse `TMList` dient der Verwaltung aller angeschlossenen Motoren. Es existiert immer ein aktueller Antrieb, auf den man mit `MP(void)` zugreifen kann.
Die Funktion `SetAxis(int n)` wählt den Motor unter der angegebenen Identifikation `n` aus und setzt diesen als neuen aktuellen Antrieb (die ID wird gespeichert in `nActiveDrive`).
Mit Hilfe von `GetAxis(void)` bekommt man den aktuellen Antrieb, `GetAxisNumber(void)` liefert die Anzahl der verfügbaren Motoren.

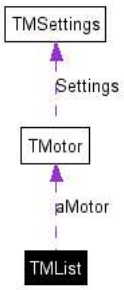


Abbildung 2.2: Kollaborationsdiagramm `TMList`

2.2.1.4 `Controller`

Die Klasse `Controller` bewerkstelligt die Kommunikation mit der Hardware. Dafür werden unterschiedliche Funktionen zum Lesen (`Read`) und Schreiben (`Write`) bereitgestellt.
Außerdem kann ein Controller mehrere Clients verwalten, d.h. es können verschiedene Geräte angeschlossen sein, deren Anzahl durch den Wert in `MaxClients` beschränkt ist.

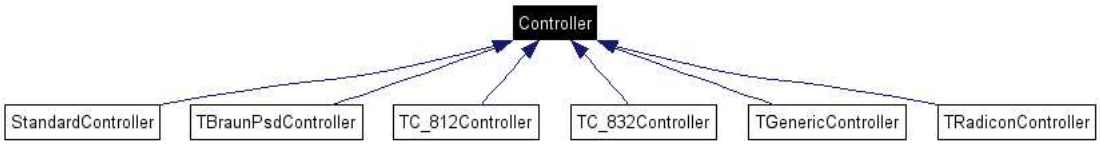


Abbildung 2.3: Klassendiagramm `Controller`

2.2.1.5 `ControllerList`

Die Klasse `ControllerList` realisiert eine Liste, innerhalb derer alle verfügbaren und angeschlossenen Controller verwaltet werden. Es gibt Methoden zum Hinzufügen eines neuen Controllers, zum Zugreifen auf einen Controller und zum Löschen aller Einträge in dieser Liste.

2.2.2 Funktionalität der Treiber

Die Treiber von *PhysikInstrumente* für den Motorcontroller C-842 ließen sich problemlos unter *Windows NT/2000* installieren (siehe Kapitel 3 Installation und Test).

Im nächsten Schritt wurde ein Testmotor an den Controller angeschlossen und mit Hilfe der *PI*-Software *WinMove* getestet. Der Testmotor führte alle gesendeten Befehle entsprechend und ohne Probleme aus. Es ließen sich nur die Bewegungen des Motors in unterschiedliche Richtungen und mit differenzierten Geschwindigkeiten überprüfen.

Im Weiteren wurde nun zusätzlich die *XCTL*-Software gestartet, um Herauszufinden, ob es eventuelle Komplikationen zwischen den Systemen gibt. In der *XCTL*-Software wurde ein Motor, der über einen Controller vom Typ C-812 bzw. C-832 angesprochen wurde, bewegt, während gleichzeitig über *WinMove* der Testmotor angesteuert wurde.
Da die verschiedenen Karten auf unterschiedlichen Speicherbereichen agieren, gab es auch hierbei keine Konflikte, so dass der Integration mittels *PI*-Treiber nichts im Wege steht. Zudem wird dadurch auch die Benutzung der DLL-Bibliothek möglich.

2.2.3 Strategie

Das Hinzufügen eines neuen Controllers erfordert zwangsläufig auch das Hinzufügen eines neuen Motors. Das bedeutet somit einen verstärkten Aufwand beim Erweitern durch einen neuen Motorcontroller, da Motor und Motorcontroller zu sehr miteinander verstrickt sind, und es keine klare Trennung zwischen Hardware-Steuerung (d.h. dem Motorcontroller, der einen realen Motor ansteuert) und der Software-Steuerung (d.h. den Motorfunktionen, die rein durch Software implementiert/emuliert werden) gibt.

Theoretisch wäre eine Umstrukturierung der Klassen und deren Hierarchie denkbar, um diesem Problem zu begegnen. Da dieses Unterfangen aber sehr aufwendig und ebenso fehleranfällig wäre, erscheint das folgende Vorgehen sinnvoll.

Wie wir gesehen haben, bestehen zwischen C-832 und C-842 enorme Gemeinsamkeiten betreffend ihrer Funktionalität und Funktionsweise. Der Motorcontroller C-842 bietet teilweise sogar einen erweiterten Funktionsumfang (z.B. bei der Bereitstellung mehrerer Trajektorienprofile), dem wir jedoch weniger Aufmerksamkeit widmen wollen, da das Ziel der Aufgabenstellung die Realisierung der gleichen Funktionalität wie mit dem Motorcontroller C-832 ist.
Deshalb soll der Motorcontroller C-842 und dessen zugehöriger Motor parallel zum C-832 in die Klassenhierarchie eingefügt werden, wie es die Abbildung 2.4 zeigt. Die neue Klasse `TC_842Controller` wird neben dem `TC_832Controller` hinzugefügt (siehe Abbildung 2.3).

Um auch die Kompatibilität zum vorhandenen System zu gewährleisten, werden alle Funktionen des C-832 für den C-842 kopiert und entsprechend neu mit Unterstützung der gelieferten DLL-Bibliothek implementiert.

Im nächsten Teilpunkt wird die detaillierte Vorgehensweise anhand von Auszügen des Quelltextes beschrieben. Dabei wird nur auf die Änderungen außerhalb der neuen Klasse `TC_842` eingegangen. Die Änderungen innerhalb dieser Klasse sind entweder selbstsprechend oder werden durch Kommentare im Quelltext beschrieben.

Nach Fertigstellung der Programmierung erfolgt das Testen des neuen Motorcontrollers in der Software.

Ist der Softwaretest erfolgreich verlaufen, so kann die Installation am realen Hardwareplatz mit erneutem Testen unter realen Bedingungen vorgenommen werden.

2.2.4 Änderungen an Dateien

Hier erfolgt eine detaillierte Auflistung aller Dateien, an denen Änderungen vorgenommen wurden.

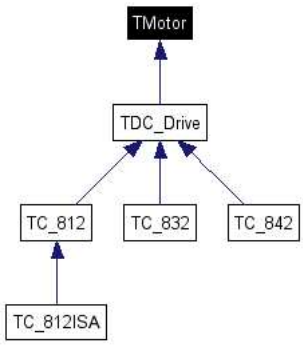


Abbildung 2.4: Klassendiagramm `TC_832` und `TC_842`

2.2.4.1 HWIO.H

Der Controller C-842 muss als neuer Gerätetyp hinzugefügt werden.

```
enum DEVICETYPE {C812 = 1, C832, C842, RADICON, BRAUN, GENERIC, STOE};
```

2.2.4.2 HWIO.CPP

In der Methode `DummyIo` erfolgt ein switch, der nach dem Geräte-Typ unterscheidet und dem Gerät einen Namen gibt.

```
case C842:
strcpy(DeviceName, "C-842");
break;
```

2.2.4.3 hwguids.h

Definieren einer `GraphicUserID`, die für die grafische Oberfläche des Optimierungsdialogs benötigt wird. Es ist aber unklar, woher diese hexadezimalen Werte stammen, bzw. wie sie generiert werden können. Deshalb befinden sich im Quellcode die kopierten Werte vom C-832.

```
#ifndef GUID_C842
#ifdef MOTORS_TESTDRIVER
DEFINE_GUID(GUID_C842, 0x2d510007, ...);
#else
DEFINE_GUID(GUID_C842, 0xb719ee70, ...);
#endif
#endif
```

2.2.4.4 M_LAYER.H

Hinzufügen neuer Callback-Funktionen.

```
typedef int (WINAPI *T842_GET_CALLBACK) (unsigned port, int hw_value);
typedef void (WINAPI *T842_PUT_CALLBACK) (unsigned port, int put);
```

Funktionen zum Registrieren oben beschriebener Callback-Funktionen.

```
void _MOTORCLASS WINAPI msRegister_C842_Get(T842_GET_CALLBACK);
void _MOTORCLASS WINAPI msRegister_C842_Put(T842_PUT_CALLBACK);
```

2.2.4.5 M_MOTHW.H

Einbinden der Headerdatei der neuen Klasse `TC_842`, die den neuen Motor zum Motorcontroller C-842 realisert.

```
#include "motrstrg\TC_842.h"
```

Erstellen einer neuen Klasse, die den Optimierungsdialog für die Motorparameter des C-842 bereitstellt.

```
class TOptimizeDC_842Dlg : public TModalDlg
{
public:
TOptimizeDC_842Dlg(void);
private:
BOOL CanClose(void);
void Dlg_OnCommand(HWND, int, HWND, UINT);
...
};
```

2.2.4.6 motorcontroller.h

Den Motorcontroller für den C-842 definieren.

```
class _MOTORCLASS TC_842Controller : public Controller
{
public:
TC_842Controller(DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList* Devices);
...
public:
BYTE activeConfig;
WORD activeDrive;
...
}
```

2.2.4.7 MOTORS.CPP

Definition der Callbacks für die Motorensimulation und Hardwarezugriffe.

```
static T842_GET_CALLBACK c842_get_callback;
static T842_PUT_CALLBACK c842_put_callback;
```

Implementation der Registrierungsfunktionen.

```
void _MOTORCLASS WINAPI msRegister_C842_Get(T842_GET_CALLBACK cb)
{
c842_get_callback = cb;
}
void _MOTORCLASS WINAPI msRegister_C842_Put(T842_PUT_CALLBACK cb)
{
c842_put_callback = cb;
}
```

Beim Initialisieren in `InitializeModule(void)` den entsprechenden Konstruktor für den C-842 aufrufen.

```
if (strcmpi(ctype, "C-842") == 0)
{
aMotor[id] = (TC_842 *) new TC_842();
continue;
}
```

Methoden zur Verwendung im Optimierungsdialog implementieren.

```
TOptimizeDC_842Dlg::TOptimizeDC_842Dlg(void) : TModalDlg(...) { ... };
BOOL TOptimizeDC_842Dlg::Dlg_OnInit(...) { ... };
void TOptimizeDC_842Dlg::Dlg_OnCommand(...){ ... };
```

```
BOOL TOptimizeDC_842Dlg::CanClose(void){ ... };
void TOptimizeDC_842Dlg::LeaveDialog(void){ ... };
```

Und als wichtigstes die Klasse `TC_842` mit der gesamten Funktionlität unter Verwendung der DLL-Bibliothek implementieren.

2.2.4.8 MSIMSTAT.CPP

Verarbeitung der Callbacks.

```
switch (sim_type)
{
case no_simulation:
msRegister_C842_Get((T842_GET_CALLBACK) GetProcAddress(hLib, "VERBOSE_C842_GET_CALLBACK_NS"));
msRegister_C842_Put((T842_PUT_CALLBACK) GetProcAddress(hLib, "VERBOSE_C842_PUT_CALLBACK") );
...
case test_simulation:
msRegister_C842_Get((T842_GET_CALLBACK) GetProcAddress(hLib, "VERBOSE_C842_GET_CALLBACK_TS"));
msRegister_C842_Put((T842_PUT_CALLBACK) GetProcAddress(hLib, "VERBOSE_C842_PUT_CALLBACK"));
...
case simulation_only:
msRegister_C842_Get((T842_GET_CALLBACK) GetProcAddress(hLib, "VERBOSE_C842_GET_CALLBACK_SO"));
msRegister_C842_Put((T842_PUT_CALLBACK) GetProcAddress(hLib, "VERBOSE_C842_PUT_CALLBACK"));
}
```

2.2.4.9 M_LAYER.CPP

Starten eines neuen Time-Events beim Ausführen eines Scans.

```
void TC_842::StartCheckScan( void ) { ... };
```

Registrieren des Treibers für die grafische Oberfläche.

```
BOOL WINAPI DllMain(...)
{
...
RegisterDrivers(&MotorDrivers,GUID_C842,C842);
...
};
```

Ermitteln und Starten des zugehörigen Controllers zum C-842.

```
Controller* GetController(...)
{
...
switch (DeviceID)
{
...
case C842:
strcpy(szDeviceName, "C-842");
pCon = new TC_842Controller(DeviceID, HardwareID, Drivers);
break;
...
}
...
}
```

2.2.5 Vereinfachung der Strategie

Während des Programmierens kristallisierte sich allmählich heraus, dass die Klassenstruktur des Controller `TC_842Controller` überflüssig wurde. Sie diente beim C-832 zur Realisierung der Hardwarekommunikation. Da diese aber beim C-842 nun vollständig durch die DLL-Bibliothek übenommen wird, blieb nur die Funktion `LimitWatch` in dieser Klasse übrig. Deswegen wurde diese Funktion mit in die Klasse `TC_842` übernommen. Damit entfällt die Controllerklasse für den C-842 vollständig.

Kapitel 3

Installation und Test

3.1 Inbetriebnahme

3.1.1 Voraussetzungen

Um einen C-842-Controller in Betrieb nehmen zu können, müssen die folgenden Voraussetzungen am Zielrechner erfüllt sein:

- ein freier ISA-Slot im PC, in den die Controllerkarte C-842 eingeschoben werden kann
- einen oder mehrere Antriebe mit inkrementellen Encodern oder anderen kompatiblen Positionierungseinheiten (wie die M500 Serie, Rotary Positioner oder DC-Mike)
- Kabel, um die Motoren mit dem Controller zu verbinden.

Außerdem muss auf dem Zielrechner das Betriebssystem *Windows NT/2000* installiert sein, damit die XCTL-Software zur Ansteuerung verwendet werden kann.

3.1.2 Installation unter *Windows NT/2000*

Das gesamte XCTL-System läuft unter der Umgebung von *Windows NT/2000*. Aufgrund von Restriktionen im Hardwarezugriff erlaubt aber *Windows NT/2000* keine direkte Bus-Kommunikation mit dem Motorcontroller C-842. Deshalb funktioniert in diesem Zusammenhang auch nicht die *Windows95* DLL.

Um auf die C-842-Karte unter *Windows NT/2000* zuzugreifen, muss das Programm oder die DLL Zugriff auf einen Kernel-null Treiber haben, der den aktuellen Hardwarezugriff auf das Board ermöglicht.
Bevor man also mit dem Motorcontroller arbeiten kann, muss der entsprechende *NT*-Treiber von *Physik-Instrumente* installiert werden.

Das C-842 *NT* Anwendungspaket erlaubt es, den C-842 Motorcontroller unter *Windows NT/2000* zu verwenden, und enthält alle benötigten Dateien zum Installieren und Registrieren des Treibers in der *NT*-Umgebung.

Für eine erste Installation unter *Windows NT/2000* muss die Datei `Setup.exe` ausgeführt werden. Diese Setup-Routine installiert den Treiber `UIO.SYS`, modifiziert die Registry und kopiert die DLL mit einer Beispielanwendung in das Zielverzeichnis.

3.1.3 Speicherbereich einstellen

Um die Kommunikation zwischen der Controller-Karte und dem *XCTL*-Programm zu ermöglichen, muß ein Speicherbereich, über den die Daten ausgetauscht werden können, festgelegt werden.

Diese Zuweisung erfolgt an der Controller-Karte, indem man den DIP-Schalter, bestehend aus 8 einzelnen Schaltern, die jeweils auf 0 oder 1 gesetzt werden können, entsprechend einstellt. Diese sogenannte Basisadresse besteht aus 12 Bit, wobei die Bits 11 und 12 stets auf 0 und das 10. Bit stets auf 1 gesetzt sind. Die Bits 1 und 2 sind nicht einstellbar und sind stets auf 0 gesetzt. Die Bits 3 bis 9 können frei gewählt werden. Damit ergibt sich der nutzbare Bereich für die Basisadresse zwischen `0x200` und `0x3FC`.

Wird ein Schalter auf `ON` gestellt, so wird das zugehörige Bit auf 0 gesetzt. Ist der Schalter dementsprechend auf `OFF` gestellt, so wird das Bit auf 1 gesetzt.

Der Schalter für das erste Bit, muß im Regelbetrieb auf 0 gesetzt werden. Der Wert 1 ist nur für Testzwecke vorbehalten.

Die häufig verwendeten Adressen sind in der folgenden Tabelle 3.1 (siehe auch Abbildung 3.1) aufgelistet und geben die Stellung der Schalter an.

Adresse	Beschaltung
0x210	00001000
0x214	00001010
0x218	00001100
0x300	10000000
0x310	10001000

Tabelle 3.1: Beispiele für Schalterstellungen (DIP)

Bit	12	11	10	9	8	7	6	5	4	3	2	1
Belegung	0	0	1	x	x	x	x	x	x	x	0	0
Schalter				SW8	SW7	SW6	SW5	SW4	SW3	SW2	SW1	
0x210	0	0	1	0	0	0	0	1	0	0	0	0
0x214	0	0	1	0	0	0	0	1	0	1	0	0
0x218	0	0	1	0	0	0	0	1	1	0	0	0

Abbildung 3.1: Speicherbereich einstellen (DIP)

3.2 Entwicklungsrechner

3.2.1 Entwicklungsumgebung

Der Entwicklungsrechner ist ein PC mit AMD Athlon Prozessor und 512 MB RAM. Auf dem Rechner ist als Betriebssystem *Windows 2000 Service Pack 4* installiert. Die Entwicklung der *XCTL*-Applikation erfolgt mit dem Tool *Visual C++ 6.0*. Zusätzlich wird *Doxygen* zur Dokumentation des Quellcodes und *CVS* zum sichern der Daten auf einem Server benutzt (siehe auch Abbildung 3.4).

Um die DLL für den Motorcontroller C-842 in der *XCTL*-Anwendung zu nutzen, muss man folgende Dinge umsetzen:

Abbildung 3.2: Entwicklungsrechner

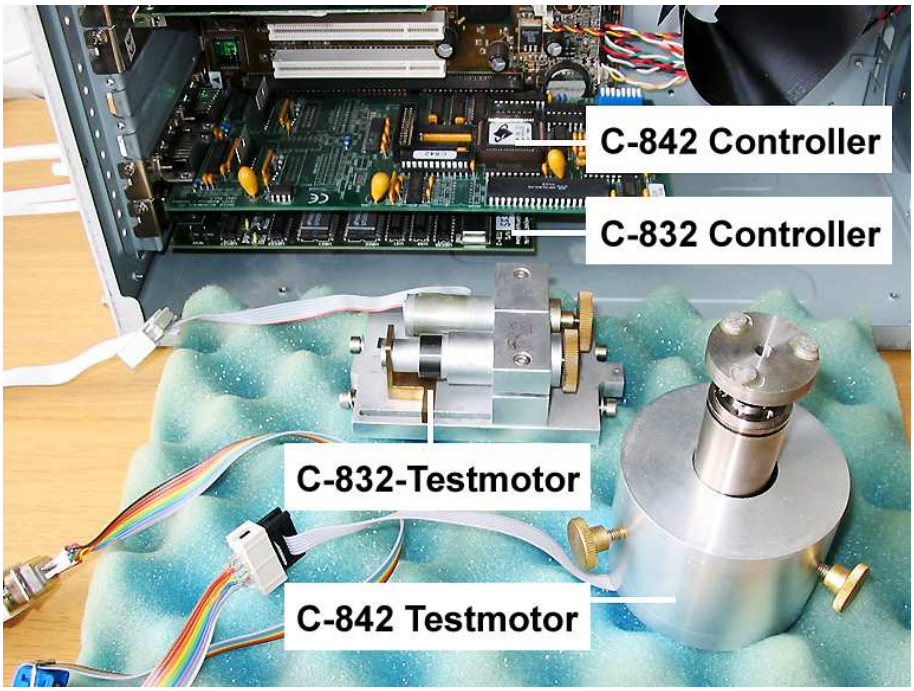


Abbildung 3.3: Controller und Motoren

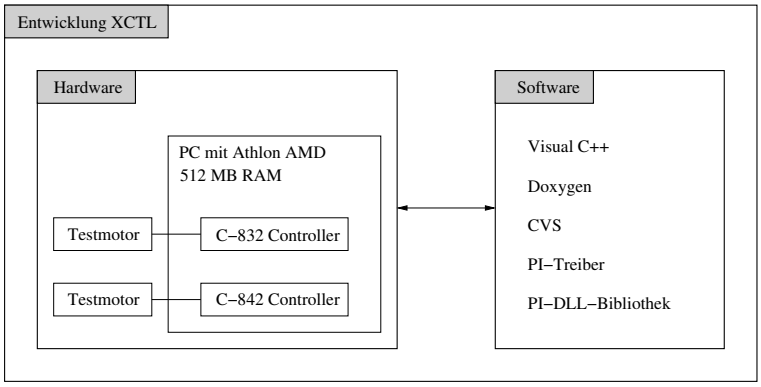


Abbildung 3.4: Entwicklungsumgebung (schematisch)

- Inkludieren der Header-Datei `C842CMD26.H` in die Anwendung. (Bei PhysikInstrumente gab es zwei Headerdateien, wobei die eine nur von der anderen verwendet wurde. Deswegen wurden im *XCTL*-System (und somit auch im CVS) beide zu einer zusammengefügt, indem der Code von `C842NT34.H` und `C842CMD26.H` gemeinsam übernommen wurde. Außerdem mußte aufgrund eines Konfliktes zwischen den Bezeichnern im Befehlssatz (`DF` = Define Home) und den Bezeichnungen für Antriebe (`DF` = Beugung fein) eine Umbenennung zu `DF0` vorgenommen werden (`DF0={0x2B, WR_NOFRAC, "DF:\0" }`).)
- Linken oder Importieren der Bibliothek `C842NT34.LIB` zum Projekt.
- Um den Zugriff mittels `translate` Funktion auf den Controller C-842 zu ermöglichen, muss die Initialisierungsfunktion `c842_open(void)` für den Treiber aufgerufen werden. Beim Verlassen der Anwendung wird der Treiber durch den Aufruf von `c842_close(void)` geschlossen. Diese Funktionen erzeugen die Verbindung zum `UIO.SYS` Treiber.

3.2.2 Test

3.2.2.1 Testen während der Entwicklung

Zunächst erfolgte kontinuierlich ein Testen am Entwicklungsrechner während der Implementation. Dazu wurde, nachdem Änderungen im Quelltext vorgenommen worden waren, das *XCTL*-Programm gestartet und unter Verwendung eines Testmotors überprüft, ob das Programm den minimalen Anforderungen entspricht. Damit ist gemeint, dass sich das Programm überhaupt erstmal kompilieren und fehlerfrei Starten ließ. War diese Bedingung erfüllt, so konnte überprüft werden, ob sich die angeschlossenen Motoren bedienen lassen.

Am Entwicklungsrechner sind dafür jeweils eine Motorcontrollerkarte vom Typ C-832 und C-842 installiert. Und an jede Karte ist wiederum ein Testmotor angeschlossen, um die ausgeführten Bewegungen sichtbar zu machen.

Es war nun stets sicher zu stellen, dass sich beide Testmotoren problemlos parallel fahren ließen. Diese Tests wurden unter zu Hilfenahme der neuen Manuellen Justage ausgeführt.
Dabei muss auch innerhalb des Dialogs der neuen Manuellen Justage die aktuelle Motorposition während der Bewegung angezeigt werden. Kommt ein Motor real zum Stehen, so darf sich dann auch die Anzeige der Position im Dialog nicht mehr ändern.

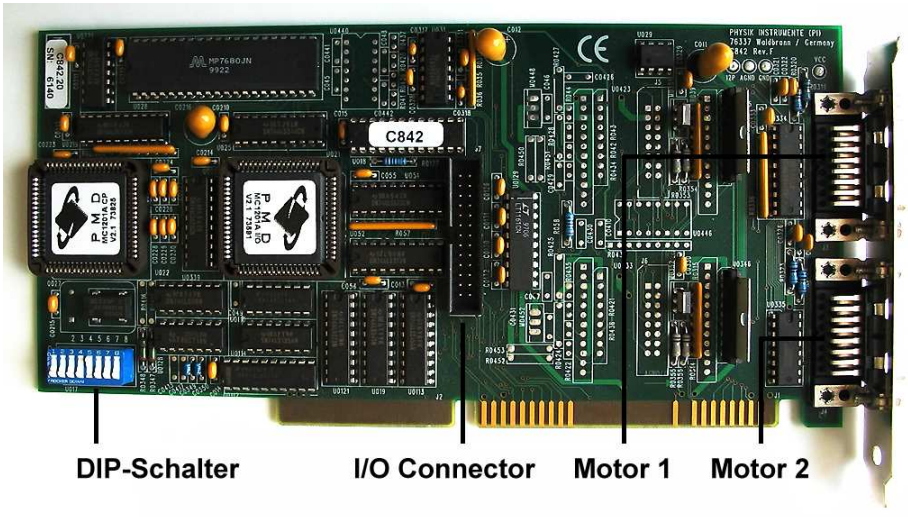


Abbildung 3.5: Controllerkarte C-842

3.2.2.2 Regressionstest mit *ATOS*

Um das Funktionieren des gesamten Systems wie vor der Erweiterung zu garantieren, wurde nach Abschluss aller Arbeiten ein Regressionstest durchgeführt. Das Programm *ATOS* wurde dazu verwendet, um alle bisher definierten Testfälle auf Fehlerfreiheit zu testen.

Erfolge

Die drei, für die Erweiterung durch den C-842-Motorcontroller, wichtigsten Testfälle, die sich mit der Motorsteuerung befassten, konnte ohne Probleme erfolgreich ausgeführt werden.

Konflikte

Die folgenden Testfälle meldeten Konflikte in der Komponente Protokollbuch, da einige Elemente (z.B. Buttons oder Dialoge) nicht identifiziert werden konnten.

- `Test_ARS.PD.1`
- `Test_LS.PD.1`
- `Test_PD.1`
- `Test_PD.2`
- `Test_PT.1`
- `Test_PT.2`
- `Test_TP.HWB.PT.1`

Fehlermeldungen der folgenden Art wurden von `ATOS` angezeigt:

`ACTION,"Protokollbuch Diffraktometrie/Reflektometrie (deutsch)",BUTTON,CLICK,"Blende/Absorber"`
Konsistenz-Fehler: Parameter 4 ist ungültiges/nicht existentes Control für das akt. Fenster!

D.h. es konnte der Button für Blende/Absorber nicht gefunden werden.

Fehler

Testsequenz: `Test_ARS.1`

- Bei der Auswahl der maximalen Intensität trat ein Ausführungs-Fehler auf:
  Ursache: Der Wert aus dem statischen Element liegt außerhalb des Toleranzbereichs!
- Außerdem kommt es bei mehreren Testfällen vor, das die zu löschende Datei `STANDARD.BAK` nicht vorhanden ist.

Testsequenz: `Test_DM.2`

- Ausführungsfehler wegen fehlender oder nicht zu findender MessageBox
  Ursache: Die Messagebox mit dem Titel "Daten-Erhebung - Information" ist nicht vorhanden!

Testsequenz: `Test_LS.1`

- Ausführungsfehler wegen fehlender oder nicht zu findender MessageBox
  Ursache: Die Messagebox mit dem Titel "Meldung" ist nicht vorhanden!
- Fehler während der Bereinigung (Cleanup). Die zu löschenden Dateien `C:\TEST2.CRV` und `C:\TESTCRV.BK` existieren nicht.

Testsequenz: `Test_LS.3`

- Ausführungsfehler wegen fehlender oder nicht zu findender MessageBox
  Ursache: Die Messagebox mit dem Titel "Meldung" ist nicht vorhanden!

Testsequenz: `Test_AJ.1`

- Ausführungsfehler bei Zugriff auf das Fenster der "Automatischen Justage".
  Ursache: Das Fenster ist nicht vorhanden!
- Die zu löschende Datei `Justage.log` wird nicht gefunden.

Testsequenz: `Test_MJ.1`

- Ausführungsfehler bei Zugriff auf das Fenster der "Manuellen Justage".
  Ursache: Das Fenster ist nicht vorhanden!

Testsequenz: `Test_MJN.1, Test_MJN.3, Test_MJN.4, Test_MJN.5, Test_MJN.6`

- In allen Fällen wird das Fenster "Manuelle Justage NEU" nach dem Öffnen nicht gefunden.
  Ursache: Das Fenster ist nicht vorhanden!

Zusammenfassung

Die Testsequenzen des Regressionstests aus dem Ordner `ATOS_XCTL32Projekt` im *CVS* konnten nicht vollständig mit *ATOS* ausgeführt werden.

21 Testsequenzen konnten erfolgreich durchgeführt werden. 7 fehlerhafte Testsequenzen wurden ignoriert.
11 Testsequenzen wurden wegen Fehlern bei der Durchführung beendet.

3.3 Arbeitsrechner

Die neue Motorcontrollerkarte vom Typ C-842 wurde an einem Messplatz der Physik eingebaut und auf ihre Funktionweise überprüft. Dabei bewegte sich ein zuvor an einer C-832-Karte angeschlossener Motor nun über die C-842-Karte.

Aber anscheinende arbeitet die Karte mit anderen Parametern und Bereichen, so dass neben der Anpassung der Hardware-Parameter in der `Hardware.ini` auch Veränderungen an den Werten für `MaxVelocity`, `SpeedScale` und `Koeff_1` vorgenommen werden mußten. Diese Werte, insbesondere `Koeff_1`, wurden mittels Test direkt am Hardwareplatz näherungsweise ermittelt.

Im Folgenden wird ein Beispiel-Motor beschrieben. Die drei wichtigsten Änderungen, sind die Anpassung des Typs (C-842), die Benutzung des Speicherbereichs (0x220) und die ID des Motors auf der Karte. Dabei ist zu beachten, dass die BoardID nur Werte größer Null annehmen darf. Sollte die Karte also zwei Anschlüsse besitzen, so sind die Werte 1 und 2 erlaubt, bei vier Anschlüssen die Werte 1 bis 4. Der Wert 0 darf nicht verwendet werden, weil er für das Ansprechen aller Motoren steht. Das würde dazu führen, dass ein Kommando, das in diesem Beispiel nur für Theta bestimmt ist, an alle angeschlossenen Motoren gesendet und ausgeführt werden würde.

```
[Motor1]
Name=Theta
Unit=Grad
Type=C-842
Type=TMotor
DeathBand=1
Digits=3
IOAddr=0x220
BoardId=1
MaxVelocity=8000
Velocity=8000
SpeedScale=1000.0
Gain=900
DynamicGain=90
IntegralGain=5
IntegralLimit=4000
Acceleration=20
RemoveLimit=14400
InitialMove=1
InitialAngle=15.0
MoveFirstToLimit=1
IndexLine=1
DistanceToZero=0
PositionMin=-1512000
MinimalWidth=0
MaximalWidth=20001
PositionWidth=1000
PositionMax=316800
AngleMin=105.000
AngleWidth=-1.0000
AngleBias=0.000
AngleMax=-22.000
Orientation=1
Direction=0
Correction=0
Hysteresis=0
DeltaPosition=-397629
Koeff_3=0.0
Koeff_2=0.0
Koeff_1=-0.2500
Koeff_0=0.0
Upwards=0
AngleZero=0.0
MaxFailure=30.0
RestartPossible=0
MJ_Offset=0
MJ_AngleDest=30.500
RestartPosible=NOT IN USE, OBSOLETE
MJ_Speed=5.000
```

Kapitel 4

Fehlersuche und Problembehandlung

4.1 Fehlverhalten

Der erste Versuch der Inbetriebnahme einer C-832-Motor-Controller-Karte auf einem Entwicklungsrechner verlief nicht reibungslos. Zunächst konnte die Hardware nicht gefunden werden, was daran lag, dass bei der Installation des Treibers kein gültiger Speicherbereich, der die Kommunikation zwischen Karte und Software ermöglicht, angegeben wurde.

Nachdem die Konsistenz zwischen Controller, Treiber und Software hergestellt worden war, liess sich das *XCTL*-Programm starten. Nach dem Starten eines Motors kam dieser jedoch nicht mehr zum Stillstand. Die Software zeigte aber an, dass er angehalten hätte, obwohl sich die aktuelle Position in der Anzeige auch nicht änderte.

Im Fahrbetrieb reagierte der Motor aber trotzdem auf Richtungsänderungen, wenn man ihn nach links bzw. nach rechts fahren ließ.

Startete man das *XCTL*-Programm im Debug-Modus erneut, so kam der Motor zum Stillstand. Eine Alternative, um den Antrieb anzuhalten, war ein Neustart, der jedoch erheblich länger benötigte.

4.2 Behebung von Fehlern

Um systematisch nach der Ursache des Fehlverhaltens zu forschen, wurde im folgenden eine Liste mit den zu überprüfenden Einstellungen und Gegebenheiten entwickelt. Es wird jeweils eine knappe Beschreibung der Strategie angegeben, mit der sich der entsprechende Fehler beseitigen lassen sollte.

1. BIOS-Einstellungen

   Da es sich bei den Karten um ISA-Karten handelt, muß im BIOS von Plug-And-Play-Software (*PnP*) auf Legacy-Software umgestellt werden. Dies muß auf jeden Fall bei den IRQs geschehen.

2. Steckkontakte

   Alle Steckkontakte, die die Verbindung zwischen Controller-Karten und Motoren herstellen, sollten überprüft werden. Dabei sind insbesondere jene Kontakte zu inspizieren, die nur an einen Stecker angelötet wurden, denn möglicherweise hat sich dort eine Verbindung gelöst und muß erneut angelötet werden.

3. ISA-Steckplatz

   Möglicherweise ist ein Steckplatz oder ein Kontakt defekt. Die Controller-Karte sollte an einen anderen ISA-Steckplatz angeschlossen werden.

4. Speicherbereich

   Es kann sein, dass bei dem verwendeten Speicherbereich Konflikte auftreten. Um einen anderen Bereich zu wählen, muß man die DIP-Schalter auf der Karte entsprechend setzen und muss dann auch in der Datei `Hardware.ini` den entsprechenden Startbereich wählen.

5. Controller-Karte

   Sofern ein weiterer Controller der gleichen Bauart verfügbar ist, sollte dieser ausgetauscht werden, um ausschließen zu können, dass der Fehler nicht an der Controller-Karte liegt.

6. Motor

   Anschließen eines anderen Motors, um festzustellen, ob der Motor nicht erwartungsgemäß reagiert. Eventuell sollte ein Motor vom echten Hardware-Platz angeschlossen werden, da die Einstellungs-Parameter (wie Geschwindigkeit, Beschleunigung, ...) nicht mit dem Testmotor kompatibel sind.

7. Treiber

   Die aktuellsten Treiber der Hardware aus dem *CVS*-System sollten neu kompiliert werden. Entfernen der alten Treiber und Neuinstallation.

8. Software

   Möglicherweise hat sich beim Programmieren der Software ein Fehler eingeschlichen, der in einer älteren Version nicht vorhanden war. Um dies festzustellen, müssen also verschiedene Versionen aus dem *CVS*-System geholt und auf ihre Funktionalität unter den gegebenen Umständen getestet werden.

Kapitel 5

Zusammenfassung

5.1 Fazit

Eine Analyse der Quellen mit *Understand* hat folgende neue Struktur ergeben. Die zwei neu hinzugekommen Klassen sind `TC_842` und `TOptimizeDC_842Dlg`. Es gibt mehrere neue Dateien durch weitere Header-Dateien und die Einbindung der DLL. Der Quellcode wuchs um 1300 Zeilen, worunter sich 300 Kommentarzeilen befinden.

	1999	A. Just. 05/2001	Diffr/Refl. 02/2002	Protokoll 01/2003	Port. 07/2003	PSD 08/2004	C-842 10/2006
Classes	84	94	97	134	142	155	157
Files	51	58	66	113	118	123	132
Functions	1088	1167	1192	2024	2204	2471	2505
Lines	29534	38910	42694	58402	65260	68453	69726
Lines Blank	2763	4192	4742	7715	9053	9624	9817
Lines Code	22488	25702	27649	35997	41630	43141	43841
Lines Comment	2819	7168	8428	13176	13686	16213	16530
Lines Inactive	528	792	871	991	923	678	682
Declar. Statment	4461	4802	4993	7480	7945	8820	8979
Exec. Statement	13648	15475	16819	22624	23210	25144	25379
Ratio Com./Code	0.13	0.28	0.30	0.37	0.33	0.38	0.38

Tabelle 5.1: Analyse der Quellen/Code

5.2 Ausblick

Es besteht die Möglichkeit, für die ebenfalls vorhandene Controller-Karte C-844 dieselbe Strategie anzuwenden, und damit auch diese Karte in das *XCTL*-System zu integrieren.

Anhang A

Motor-Controller C-842/C-844

A.1 Motor-Controller C-842

Für den Motor-Controller C-842 gibt es die Varianten mit zwei Motoren (C-842.20) und mit vier Motoren (C-842.40). Der C-842 Motor-Controller unterstützt die closed-loop Servokontrolle für unterschiedliche Servomotoren. Jede Achse/Motor hat ein kompliziertes Trajektorienprofil und digitale Filter, so dass auch sehr kleine Positionen (kleine Schrittweiten) und geringe Geschwindigkeiten erreicht werden können. Die einzelnen Achsen/Motoren können synchron oder unabhängig von einander programmiert werden, um komplexe Bewegungen zu realisieren.

A.1.1 Board Register

A.1.1.1 Register Tabelle

Die Motorkommandos und Bewegungen werden durch Lesen und Schreiben an der entsprechenden Adresse realisiert. Die standardmäßige Einstellung für die Basisadresse (BA) ist `0x210`. Die folgende Tabelle gibt eine Übersicht über die Verfügbarkeit und Aufgabe der internen Register.

A.1.1.2 Status Register

Auf das Board Status Register kann nur lesend zugegriffen werden und beinhaltet die grundlegenden Statusmeldungen des C-842.

Wenn das Board Interrupt Bit nicht gesetzt ist, so ist in diesem Moment ein Interrupt aktiv.
Das System Status Bit gibt an, ob das System bereit ist (Bit ist gesetzt) oder ob im Moment keine neue Anweisung ausgeführt werden kann (Bit nicht gesetzt).

BA	Read/Write	Parametertransfer
BA+1	Read	Lese im Board Status Register
BA+1	Write	Schreibe Kommando für DSP
BA+2	Write only	Schreibe im Board Function Register
BA+3	Read	Lese digitalen Output
BA+3	Write	Schreibe digitalen Output

Tabelle A.1: Register Tabelle

Bit	Wert	Funktion
0	1	Negatives Index Signal auf dem Kanal 1
1	2	Negatives Index Signal auf dem Kanal 2
2	4	Negatives Index Signal auf dem Kanal 3
3	8	Negatives Index Signal auf dem Kanal 4
4	16	-
5	32	-
6	64	Board Interrupt
7	128	System Status

Tabelle A.2: Status Register

A.1.1.3 Function Register

Das Function Register kann nur beschrieben werden und ermöglicht es, bestimmte Grundeinstellungen für den C-842 festzulegen.

Bit	Wert	Funktion
0	1	On-Board Verstärker einschalten
1	2	Encodersignale im Differentialmodus (Achse 1 und 2)
2	4	Encodersignale im Differentialmodus (Achse 3 und 4)
3	8	Limit Switch line levels
4	16	IRQ10 einschalten
5	32	IRQ11 einschalten
6	64	IRQ12 einschalten
7	128	IRQ15 einschalten

Tabelle A.3: Function Register

A.1.2 Output-Limit

Die Stromversorgung für die internen Verstärker kann durch das Bit 0 im Function Register ein- bzw. ausgeschaltet werden.

Alle Output-Kanäle besitzen eine eigene Begrenzung für die Kapazität des fließenden Stromes. Von der Fabrik her ist pro Kanal ein Limit von 1,2A gesetzt. Überschreitet ein Motor dieses Limit, so wird die Ausgangsspannung für 100ms unterbrochen. Danach wird die Spannung wieder eingeschaltet, falls das Limit nicht überschritten wurde, ansonsten bleibt die Spannung für weitere 100ms deaktiviert.

Zusätzlich kann ein optionales Limit eingestellt werden, bei dessen Überschreiten die Spannung des entsprechenden Motors für 15ms abgeschaltet wird.

A.1.3 Trajectory Profile Generation

Der Profil-Generator dient dazu, die Zielposition, Geschwindigkeit und Beschleunigung eines Motors zu bestimmen. Die unterschiedliche Profile können mit den folgenden Kommandos ausgewählt werden.

Alle Profile besitzen einen doppelten Puffer für die Parameter, um so diese Werte im voraus zu laden. Abhängig vom Profil können auch einige Parameter während der Bewegung der Motoren aktualisiert werden. Jede der zwei bzw. vier Achsen kann unabhängig voneinander ein anderes Profil geladen haben (Ausnahme: Electronic Gearing).

Profil	Kommando	Parameteranzahl
S-curve Point to Point	SET_PRFL_S_CRV	4
Trapezoidal Point to Point	SET_PRFL_TRAP	3
Velocity Contouring	SET_PRFL_VEL	2
Electronic Gearing	SET_PRFL_GEAR	1

Tabelle A.4: Bewegungsprofile

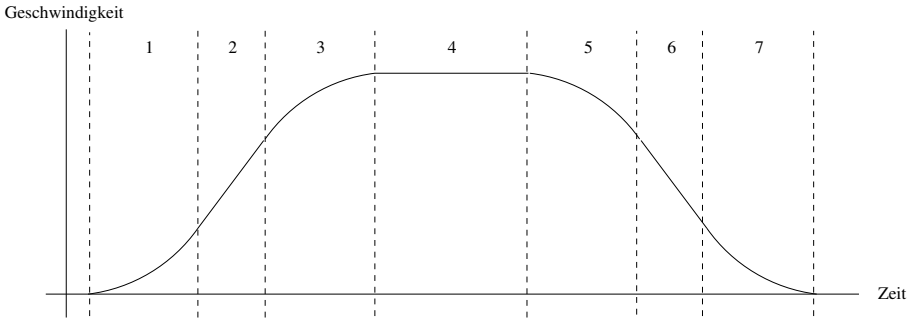


Abbildung A.1: S-curve Profil

A.1.3.1 S-curve Point to Point

Dieses Profil fährt die ausgewählte Achse mit dem spezifizierten Stoß, solange bis die maximale Beschleunigung erreicht ist (Phase 1). Dann wird die Achse nur noch mit Stoß=0, d.h. konstanter Beschleunigung, gefahren (Phase 2). Danach wird die Achse mit dem negativen Stoß betrieben (Phase 3), so daß die maximale Geschindigkeit mit Beschleunigung=0 erreicht wird (Phase 4). Damit wird die Beschleunigungsphase beendet und die Achse mit konstanter Geschwindigkeit gefahren. Hat die Achse die Zielposition fast erreicht, so wird die Bremsphase (Phase 5,6,7) eingeleitet. Diese verläuft symmetrisch zur Beschleunigungsphase, so dass die Zielposition mit Beschleunigung=0 und Geschwindigkeit=0 erreicht wird.

Dabei kann es passieren, dass die Phasen 2 und 6 wegfallen, da auf dem Weg zur maximalen Geschindigkeit die maximale Beschleunigung nicht erreicht werden kann. Andererseits kann es passieren, dass die maximale Geschindigkeit nicht erreicht wird, weil die Zielposition zu nah liegt, so dass Phase 4 entfällt.

Parameter	Repräsentation	Einheit
Zielposition	signed 32 Bit	counts
max. Geschwindigkeit	unsigned 32 Bit	counts/smpl
max. Beschleunigung	unsigned 16 Bit	counts/smpl^2
Stoß	unsigned 32 Bit	counts/smpl^3

Tabelle A.5: Parameter zu S-curve Point to Point

A.1.3.2 Trapezoidal Point to Point

In diesem Profil wird die Zielposition, die maximale Geschwindigkeit und eine Beschleunigung festgelegt. Die Trajektorie wird dann gebildet, indem solange beschleunigt wird, bis die maximale Geschindigkeit erreicht wurde. Diese wird beibehalten bis kurz vor dem Ziel, wo dann mit der negativen Beschleunigung abgebremst wird.

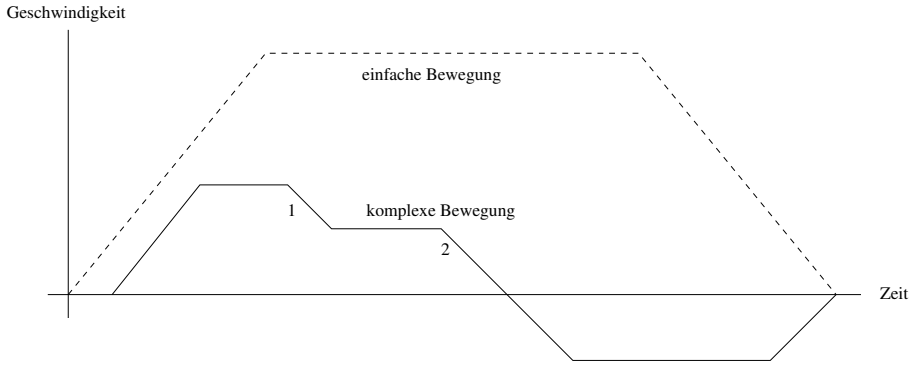


Abbildung A.2: Trapezoidal Profil

Eine komplexere Bewegung ist auch in der Abbildung dargestellt. Zum Zeitpunkt 1 wird die maximale Geschindigkeit herabgesetzt, zum Zeitpunkt 2 erfolgt eine Änderung der Zielposition in die entgegengesetzte Richtung.

Dabei dürfen die Geschindigkeit und die Zielposition in der Bewegung verändert werden. Um die Beschleunigung zu variieren ist zuvor ein Stopp der entprechenden Achse notwendig.

Um in dieses Profil umzuschalten, ist es ebenfalls erforderlich, die entsprechende Achse zuvor zu stoppen.

Parameter	Repräsentation	Einheit
Zielposition	signed 32 Bit	counts
max. Geschwindigkeit	unsigned 32 Bit	counts/smpl
Beschleunigung	unsigned 32 Bit	counts/smpl^2

Tabelle A.6: Parameter zu Trapezoidal Point to Point

A.1.3.3 Velocity Contouring

In diesem Profil muß man nur die Geschwindigkeit und Beschleunigung spezifizieren. Die Trajektorie wird kontinuierlich mit der angegebenen Beschleunigung gefahren, solange bis die maximale Geschwindigkeit erreicht wird. Dann wird die Achse mit konstanter Geschwindigkeit betrieben, es sei denn die maximale Geschindigkeit ändert sich (siehe 1) oder die Beschleunigung (siehe 2) oder beides gleichzeitig (siehe 3).

Die Geschwindigkeit muß immer einen positiven Wert besitzen. Die Richtung der Bewegung wird durch die Beschleunigung angegeben. Eine positive Beschleunigung resultiert in einer Bewegung in die positive Richung, eine negative Beschleunigung in die negative Richtung.

Alle Parameter dürfen während des Betriebes jederzeit geändert werden. Es ist aber zu beachten, keine utopischen Werte anzugeben, um eine gleichmäßige, nicht ruckartige, Bewegung zu garantieren.

Parameter	Repräsentation	Einheit
max. Geschwindigkeit	unsigned 32 Bit	counts/smpl
Beschleunigung	signed 32 Bits	counts/smpl^2

Tabelle A.7: Parameter zu Velocity Contouring

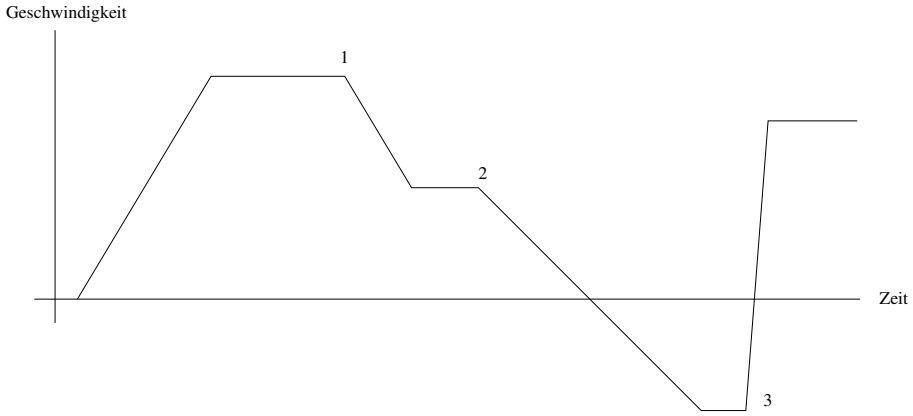


Abbildung A.3: Velocity Contouring Profil

A.1.3.4 Electronic Gearing

Dieses Profil benötigt nur den Parameter des Getriebeverhältnisses (Gear Ratio). Die Zielposition wird dadurch bestimmt, dass das Getriebeverhältnis der Achse auf die aktuelle Position einer anderen Achse übertragen wird (Slave).

Parameter	Repräsentation	Einheit
Getriebeverhältnis	signed 32 Bit	-

Tabelle A.8: Parameter zu Electronic Gearing

A.1.4 Digital Servo Filtering

Die folgenden beiden Filter dienen dazu, den Motor zu steuern, unter Berücksichtigung von Positionsfehlern.

A.1.4.1 PID

PID = Proportinal Gain, **I**ntegral Gain, **D**erivative Gain

Die zugehörigen Parameter sind im folgenden dargestellt.

Parameter	Symbol	Repräsentation
Proportional Gain	*PG*	unsigned 16 Bit
Integral Gain	*IG*	unsigned 16 Bit
Derivative Gain	*DG*	unsigned 16 Bit
Integration Limit	*IL*	unsigned 16 Bit

Tabelle A.9: Parameter zu **PID**

Input:

- Zielposition (Z_n)

Parameter	Symbol	Repräsentation
Proportional Gain	PG	unsigned 16 Bit
Integral Gain	IG	unsigned 16 Bit
Velocity Gain	VG	unsigned 16 Bit
Feed forward Gain	$VGff$	unsigned 16 Bit
Integration Limit	IL	unsigned 16 Bit

Tabelle A.10: Parameter zu **PIVff**

- aktuelle Position (P_n)
- $\Rightarrow$ daraus resultiert der aktuelle Positionsfehler $E_n = Z_n - P_n$

Berechnung:

- P $= E_n \cdot PG$
- I $= (S + E_n) \cdot IG; S = S + E_n$
- D $= (E_n - E_{n-1}) \cdot DG$

Output:

- Output = P + I + D

A.1.4.2 PIVff

PIVff = PI with **V**elocity **f**eed **f**orward
Die zugehörigen Parameter sind in folgender Tabelle dargestellt.
Input:

- Zielposition (Z_n)
- aktuelle Position (P_n)
- $\Rightarrow$ daraus resultiert der aktuelle Positionsfehler $E_n = Z_n - P_n$
- Zielgeschwindigkeit (ZV_n)
- aktuelle Geschwindigkeit (V_n)

Berechnung:

- Positionskommando $PC_n = E_n * PG + \int E_n \cdot \frac{IG}{256}$
- Geschwindigkeitsfehler $VE_n = \frac{PC_n}{256} + 32 \cdot \left(ZV_n \cdot \frac{VGff}{16384} - V_n \right)$

Output

- Output = $VE_n \cdot VG$

A.2 Motor-Controller C-844

Der Motor-Controller C-844 unterstützt vier Motoren. Der C-844 basiert auf einer Multi-Prozessor-Architektur. Ein internes Echtzeit-System basierend auf der Multitasking-Architektur ermöglicht es, Interrupts, I/O-Operationen, Grenzwerte und Benutzeranfragen zu behandeln. Der C-844 bietet eine leistungsfähige Bewegungskontrolle mit vielen Optionen zur Bildung von Trajektorien und Filtereinstellungen.

A.3 Vergleich C-842 und C-844

Die folgende Tabelle gibt einen knappen Überblick über diese beiden Controller, die sich auch nicht wesentlich unterscheiden.

	C-842	C-844
CPU	MC1401 (16 Bit)	MC1401A (16 Bit, 10 MHz)
Operating Modes	- closed loop - open loop	- closed loop - open loop
Trajectory Profile Generator	- S-curve Point to Point - Trapezoidal Point to Point - Velocity Contouring - Electronic Gear	- S-curve Point to Point - Trapezoidal Point to Point - Velocity Contouring - Electronic Gear
Filter Modes	-PID -PIVff	-PID -PIVff -Bias

Tabelle A.11: Übersicht C-842/C-844

A.4 Unterstützung für Windows 2000

Für den Motor-Controller C-842 gibt es verschiedene Bibliotheken (Stand 1996):

- C-Libraries für Borland und Microsoft C (Name: QFLC,QFLMC)
- DLL-Library (Name: QFLW152.DLL)

Alle Kommandos die zur Steuerung des Prozessors notwendig sind, werden in den C-Bibliotheken zur Verfügung gestellt. Außerdem werden noch weitere nützliche Kommandos emuliert, so daß man insgesamt auf über 100 Befehle zurückgreifen kann.

Die DLL ist eine dynamische 16-Bit-Bibliothek und ermöglicht den Zugriff auf den Controller von Windows aus. Es werden unter anderem MS-Visual C++ 1.5 und C++ unterstützt.

Zitat aus “Position und Bewegung News” von PI, Ausgabe 26, 1999:

Die Schutzmechanismen des NT Betriebssystems lassen keine direkten Hardwarezugriffe zu und erschweren so den Einsatz von ISA-Bus Steckkarten, wie z.B. den C-842 Motor-Controller.
Für den Einsatz des Controllers unter NT steht ein Systemtreiber zur Verfügung, der zunächst im Betriebssystem installiert wird. Danach ist es möglich, mit der mitgelieferten DLL über den Treiber auf die Motorkarte zuzugreifen.

Für den Controller C-844 konnten bisher keine Dokumente gefunden werden, die darauf hinweisen, dass es für diesen Motor auch schon Bibliotheken gibt.

Anhang B

Vergleich der Funktion von C-832 und C-842

Die folgenden Tabellen sollen Aufschluß darüber geben, welche Funktionalität die einzelnen Funktionen der Klasse `TC_832` besitzen, welche Routinen sie verwenden und wiederum von welchen Funktionen sie selbst aufgerufen werden. Außerdem wird das Analogon für den Motor `TC_842` aus der bereitgestellten DLL-Bibliothek von PhysikInstrumente aufgelistet, um einen direkten Vergleich zu ermöglichen.

B.1 Motorfunktionen mit Hardwarezugriffen

C832	Klasse	`TC_832`
	Funktion	`ActivateDrive(void)`
	Eigenschaft	geschützt (`protected`)

	Verwendung	`TMotorParamDlg::CanClose(void);`
		`TDC_Drive::Initialize(void);`
		`TManJustage::DoDirect(const int aIndex);`
		`TManJustage::DoDrive(const int aIndex,`
		`  const Edirection aDirection);`
		`TManJustage::DoStep(const int aIndex,`
		`  const EDirection aDirection);`
		`_MOTORCLASS WINAPI mActivateDrive(void)`

	Aufrufe	`Hardware->UpdateController; Drive628; _GetPosition`
	Beschreibung	Update der Filter-Parameter. Ermitteln der realen Position des Motors. Laden der Trajektorie (einfaches Laden der Position). Starten der Bewegung.
C842	DLL-Funktion	-
	Beschreibung	Funktionalität beim C-842 überflüssig.

Tabelle B.1: `TC_832::ActivateDrive(void)`

C832	Klasse	`TC_832`
	Funktion	`ActivateFilterParameters(void)`
	Eigenschaft	geschützt (`protected`)
	Verwendung	`TOptimizeDC_832Dlg::Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);` `TOptimizeDC_832Dlg::LeaveDialog(void);` `TDC_Drive::Initialize(void)`
	Aufrufe	`Hardware->UpdateController; Drive628; SetVelocity; SetAcceleration; GetSpeed`
	Beschreibung	Lade die Filter-Parameter zur Optimierung eines Motors (Static Gain, Dynamic Gain, Integral Gain, Integral Limit). Anschließend Update der Filter-Parameter.
C842	DLL-Funktion	`Execute`
	Beschreibung	Filter-Parameter mit Hilfe von `Execute` laden.

Tabelle B.2: `TC_832::ActivateFilterParameters(void)`

C832	Klasse	`TC_832`
	Funktion	CheckBoardOk(void)
	Eigenschaft	geschützt (`protected`)
	Verwendung	`TDC_Drive::Initialize(void)`
	Aufrufe	`Hardware->UpdateController; Hardware->Put; Hardware->Get`
	Beschreibung	Testet den Motorcontroller, ob Daten geschrieben und gelesen werden können.
C842	DLL-Funktion	`Board_Installed`
	Beschreibung	Diese Funktion liefert `true` zurück, falls eine C-842-Karte in einem PC-Slot installiert ist, ansonsten `false`.

Tabelle B.3: `TC_832::CheckBoardOk(void)`

C832	Klasse	`TC_832`
	Funktion	ExecuteCmd(LPSTR)
	Eigenschaft	privat (`private`)
	Verwendung	Keine Verwendung erkennbar.
	Aufrufe	`Hardware->UpdateController`
	Beschreibung	In der Klasse `TC_812` gibt es auch diese Funktion, sie liefert aber nur einen boolschen Wert zurück. Vermutlich soll dieser Wert angeben, ob ein Kommando erfolgreich ausgeführt wurde.
C842	DLL-Funktion	`Execute`
	Beschreibung	Es existiert eine Funktion `Execute(axis:byte, command:TCmd, parameter:longint, report:string)`, die ein angebenes Kommando ausführt. Es funktioniert wie `Translate(command:string, report:string)`, nur das hier als Parameter Strings übergeben werden können, wobei das Kommando in *QMove*-Schreibweise angegeben werden muss. (z.B. `MR5000`) `Execute` ist die zentrale Prozedur für C-Programmierung. Das vollständige Kommando-Set des Prozessors kann verwendet werden. Außerdem können viele durch Software emulierte Kommandos genutzt werden. Über 100 Kommandos können ausgewählt und in einfacher Weise ausgeführt werden.

Tabelle B.4: `TC_832::ExecuteCmd(LPSTR)`

C832	Klasse	`TC_832`
	Funktion	GetStatus(void)
	Eigenschaft	privat (`private`)
	Verwendung	Keine Verwendung erkennbar.
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Status des Motorcontrollers. Auslesen der Status-Bytes.
C842	DLL-Funktion	`get_stat(axis:byte)`
	Beschreibung	Diese Funktion gibt das Statuswort der ausgewählten Achse zurück.

Tabelle B.5: `TC_832::GetStatus(void)`

C832	Klasse	`TC_832`
	Funktion	`_GetFailure(long *failure)`
	Eigenschaft	privat (`private`)
	Verwendung	`TDC_Drive::GetFailure(void)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Ermitteln der Abweichung von tatsächlicher Position und gewünschter Position.
C842	DLL-Funktion	`get_PosErr(axis:byte)` oder `get_err(axis:byte)` Fehlerhafte Dokumentation bei PI, die Funktion heißt `get_posErr`.
	Beschreibung	Diese Funktion gibt die gegenwärtige Zahl von Fehlercounts, d.h. die Abweichung der realen Position von der virtuellen Position, zurück.

Tabelle B.6: `TC_832::_GetFailure(long *failure)`

C832	Klasse	`TC_832`
	Funktion	`_GetPosition(long *position)`
	Eigenschaft	privat (`private`)
	Verwendung	`TDC_Drive::GetPosition(BOOL bSave)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Liefert die reale Position zurück.
C842	DLL-Funktion	`get_pos(axis:byte)`
	Beschreibung	Diese Funktion gibt die gegenwärtige aktuelle Position des gewählten Motors/Achse zurück. Die Funktion `get_pos4(longint array_of_positions)` liest die aktuellen Positionen aller vier Achsen aus und speichert diese in einem Positionsfeld.

Tabelle B.7: `TC_832::_GetPosition(long *position)`

C832	Klasse	`TC_832`
	Funktion	`IsIndexArrived(void)`
	Eigenschaft	privat (`private`)
	Verwendung	`TCalibrateDlg::Dlg_OnTimer(HWND hwnd, UINT id)`
	Aufrufe	`Hardware->UpdateController; Drive628; IsLimitHit; DelayTime`
	Beschreibung	Motor abrupt stoppen. Komplexes Kommando, siehe Quellcode.
C842	DLL-Funktion	-

	Beschreibung	`TCmd: aAB`
		`a = axis (0..4)`
		generisches Kommando: nicht vorhanden
		Die Bewegung des angegebenen Motors wird sofort/abrupt gestoppt, ohne den Motor zu entschleunigen.

Tabelle B.8: `TC_832::IsIndexArrived(void)`

C832	Klasse	`TC_832`
	Funktion	`IsLimitHit(void)`
	Eigenschaft	privat (`private`)
	Verwendung	`TC_832::IsIndexArrived(void)`
	Aufrufe	`Hardware->UpdateController; Hardware->LM628Ready; Hardware->Put; Hardware->Get; Drive628; IsMoveFinish; DelayTime`
	Beschreibung	Motor abrupt stoppen. Komplexes Kommando, siehe Quellcode
C842	DLL-Funktion	-
	Beschreibung	Siehe -> `IsIndexArrived`

Tabelle B.9: `TC_832::IsLimitHit(void)`

C832	Klasse	`TC_832`
	Funktion	`IsMoveFinish(void)`
	Eigenschaft	privat (`private`)

	Verwendung	`TCalibrateDlg::Dlg_OnTimer(HWND hwnd, UINT id);`
		`TPosControlDlg::Dlg_OnTimer(HWND hwnd, UINT id);`
		`TMotor::SetRelativeZero(void);`
		`TMotor::ResetRelativeZero(void);`
		`TMotor::SetAngleBias(double aAngleBias);`
		`TC_832::StopDrive(BOOL bOffOn);`
		`TC_832::IsLimitHit(void);`
		`TC_832::IsIndexArrived(void);`
		`_MOTORCLASS CALLBACK mSavePosition(...);`
		`_MOTORCLASS WINAPI mlIsMoveFinish(int mid)`

	Aufrufe	`Hardware->UpdateController; Drive628; DelayTime`
	Beschreibung	Testet, ob die Bewegung des Motors abgeschlossen ist.
C842	DLL-Funktion	`moving(byte:axis)`
	Beschreibung	Diese Funktion liefert `true` zurück, falls die gewählte Achse noch in Bewegung ist, d.h. noch nicht die Zielposition erreicht hat, ansonsten `false`.

Tabelle B.10: `TC_832::IsMoveFinish(void)`

C832	Klasse	`TC_832`
	Funktion	`MoveAbsolute(long pos)`
	Eigenschaft	privat (`private`)
	Verwendung	`TDC_Drive::MoveToPosition(long position)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Führe eine absolute Bewegung des Motors aus. Lade die Zielposition und starte die Bewegung.
C842	DLL-Funktion	`MoveA(axis:byte, position:longint)`
	Beschreibung	Diese Funktion startet einen oder alle Motoren, um eine absolute Position anzufahren. Ist die gewählte Achse 0, so werden alle verfügbaren Achsen auf die gleiche Position gefahren. Wird die Funktion aufgerufen, wird sofort die Bewegung gestartet und die Funktion danach sofort verlassen. Erfolgt der nächste Aufruf von `MoveA` noch bevor der/die Motor(en) die Bewegung abgeschlossen haben, so wird die alte Zielposition durch die neue ersetzt und die Motoren bewegen sich auf die neue Zielposition zu. Um Bewegungen zu synchronisieren und das nächste Bewegungs-Kommando nicht vor erreichen des Ziels ausführen zu lassen, muss das WaitStop-Kommando ausgeführt werden, bevor eine neue Bewegung gestartet wird.

Tabelle B.11: `TC_832::MoveAbsolute(long pos)`

C832	Klasse	`TC_832`
	Funktion	`MoveRelative(long position)`
	Eigenschaft	privat (`private`)
	Verwendung	`TDC_Drive::MoveByPosition(long distance)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Startet eine Motorbewegung relativ zur aktuellen Position. Einlesen der aktuellen Position des Motors, Hinzuaddieren der relativen Position, Laden dieser neu berechneten Position und Starten der Bewegung.
C842	DLL-Funktion	`MoveR(axis:byte, counts:longint)`
	Beschreibung	Diese Funktion startet einen Motor, um eine relative Position anzufahren. Das angegebene `count` wird zur gegenwärtigen Position hinzu addiert und das Ergebnis ist die zu erreichende Zielposition. Wird die Funktion aufgerufen, wird sofort die Bewegung gestartet und die Funktion danach sofort verlassen. Erfolgt der nächste Aufruf von `MoveR` noch bevor der/die Motor(en) die Bewegung abgeschlossen haben, so wird die alte Zielposition durch die neue ersetzt und die Motoren bewegen sich auf die neue Zielposition zu. Um Bewegungen zu synchronisieren und das nächste Bewegungs-Kommando nicht vor erreichen des Ziels ausführen zu lassen, muss das `WaitStop`-Kommando ausgeführt werden, bevor eine neue Bewegung gestartet wird.

Tabelle B.12: `TC_832::MoveRelative(long position)`

C832	Klasse	`TC_832`
	Funktion	`Reset(void)`
	Eigenschaft	geschützt (`protected`)
	Verwendung	`TDC_Drive::Initialize(void)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Initialisierung. Zurücksetzen des LM628, der Intterrupts und Maskieren der Interrupts.
C842	DLL-Funktion	`ClearAxisStatus(axis:byte)`
	Beschreibung	Diese Prozedur setzt das Statusregister der gewählten Achse zurück.

Tabelle B.13: `TC_832::Reset(void)`

C832	Klasse	`TC_832`
	Funktion	`SetAcceleration(DWORD acceleration)`
	Eigenschaft	privat (`private`)
	Verwendung	`TC_832:: ActivateFilterParameters(void)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Einstellen der Beschleunigung.
C842	DLL-Funktion	`Set Acceleration`

	Beschreibung	`TCmd: aSAn`
		`a = axis (0..4)`
		`n = integer, 0..200000`
		generisches Kommando: `SET_ACC` (`0x12`)
		Setzt die Beschleunigung des gewählten Motors auf $n \cdot \frac{counts}{s^2}$.

Tabelle B.14: `TC_832::SetAcceleration(DWORD acceleration)`

C832	Klasse	`TC_832`
	Funktion	`SetVelocity(DWORD velocity)`
	Eigenschaft	privat (`private`)
	Verwendung	`TDC_Drive::SetSpeed(double speed);` `TC_832:: ActivateFilterParameters(void)`
	Aufrufe	`Hardware->UpdateController; Drive628`
	Beschreibung	Einstellen der Geschwindigkeit.
C842	DLL-Funktion	`Set Velocity`

	Beschreibung	`TCmd: aSVn`
		`a = axis (0..4)`
		`n = integer, 0..500000`
		generisches Kommando: `SET_VEL` (`0x11`)
		Setzt die Geschwindigkeit der angegebenen Achse auf $n \cdot \frac{counts}{s}$.

Tabelle B.15: `TC_832::SetVelocity(DWORD velocity)`

C832	Klasse	`TC_832`
	Funktion	`SetHome(void)`
	Eigenschaft	privat (`private`)
	Verwendung	`TCalibrateDlg::Dlg_OnCommand(HWND hwnd, int id,` `HWND hwndCtl, UINT codeNotify)`
	Aufrufe	`Hardware->UpdateController; Drive628; TMotor::SetHome`
	Beschreibung	Definieren der Home-Position. Alles auf Null setzen. Setzt aktuelle Position als Null-Position. Starten der Bewegung.
C842	DLL-Funktion	`Define Home`

	Beschreibung	`TCmd: aDH`
		`a = axis (0..4)`
		generisches Kommando: `ZERO_POS` (`0x3F`)
		`Definiert die aktuelle Position als Home-Position.`

Tabelle B.16: `TC_832::SetHome(void)`

C832	Klasse	`TC_832`
	Funktion	`StartToIndex(long &oldPos)`
	Eigenschaft	privat (`private`)
	Verwendung	`TCalibrateDlg::Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)`
	Aufrufe	`Hardware->UpdateController; SetSpeed; SetCalibrationState; SetCorrectionState; Drive628; DelayTime`
	Beschreibung	Motor wird auf Startposition zurück gefahren. Anfahren einer utopischen Position pos = -4000000001. Komplexes Kommando, siehe Quellcode.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.17: `TC_832::StartToIndex(long &oldPos)`

C832	Klasse	`TC_832`
	Funktion	`StopDrive(BOOL bOffOn)`
	Eigenschaft	privat (`private`)

	Verwendung	`TMList::SaveModuleSettings(void);`
		`TCalibrateDlg::LeaveDialog(void);`
		`TPosControlDlg::LeaveDialog(void);`
		`TManJustage::DoStop (const int aIndex);`
		`_MOTORCLASS WINAPI mStopDrive(BOOL restart)`

	Aufrufe	`Hardware->UpdateController; Drive628; IsMoveFinish; DelayTime`
	Beschreibung	Laden der Trajektorie “Stop Smoothly”. Motor anhalten. Warten, bis der Motor wirklich steht.
C842	DLL-Funktion	`Execute`
	Beschreibung	Stoppen des Motors mittels `Execute` und `ST`-Kommando.

Tabelle B.18: `TC_832::StopDrive(BOOL bOffOn)`

C832	Klasse	`TC_832`
	Funktion	`Drive628(BYTE cmd, WORD ctrl_word, long param)`
	Eigenschaft	privat (`private`)

	Verwendung	`TC_832::Reset(void);`
		`TC_832::ActivateFilterParameters(void);`
		`TC_832::SetHome(void);`
		`TC_832::ActivateDrive(void);`
		`TC_832::StopDrive(BOOL bOffOn);`
		`TC_832::IsLimitHit(void);`
		`TC_832::IsIndexArrived(void);`
		`TC_832::StartToIndex(long &oldPos);`
		`TC_832::SetVelocity(DWORD velocity);`
		`TC_832::SetAcceleration(DWORD acceleration);`
		`TC_832::GetStatus(void);`
		`TC_832::_GetPosition(long *position);`
		`TC_832::_GetFailure(long *failure);`
		`TC_832::MoveRelative(long position);`
		`TC_832::MoveAbsolute(long pos);`
		`TC_832::IsMoveFinish(void)`

	Aufrufe	`Hardware->UpdateController, Hardware->Drive628c`
	Beschreibung	Realisiert die Hardwarezugriffe auf den Controller.
C842	DLL-Funktion	-
	Beschreibung	Funktion wird nicht mehr benötigt, da die Zugriffe in den Funktionen selbst durch die Verwendung der Routinen aus der DLL realisiert werden.

Tabelle B.19: `TC_832::Drive628(BYTE cmd, WORD ctrl_word, long param)`

B.2 Mutator/Accessor-Methoden

C832	Klasse	`TC_832`
	Funktion	`GetAcceleration(void)`
	Eigenschaft	privat (`private`)
	Verwendung	`_MOTORCLASS WINAPI mlGetValue(int mid, EValueType vtype);` `_MOTORCLASS WINAPI mGetValue(EValueType vtype)`
	Aufrufe	`Hardware->UpdateController`
	Beschreibung	Liefert den Wert der Beschleunigung des aktuellen Motors/Controllers.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.20: `TC_832::GetAcceleration(void)`

C832	Klasse	`TC_832`
	Funktion	`GetVelocity(void)`
	Eigenschaft	privat (`private`)
	Verwendung	`TDC_Drive::SetSpeed(double speed);` `TDC_Drive::GetSpeed(void)`
	Aufrufe	`Hardware->UpdateController`
	Beschreibung	Liefert die Geschwindigkeit des Motors/Controllers.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.21: `TC_832::GetVelocity(void)`

B.3 Funktionen zur Verwaltung und sonstige

C832	Klasse	`TC_832`
	Funktion	`TC_832(void)`
	Eigenschaft	öffentlich (`public`)
	Verwendung	
	Aufrufe	
	Beschreibung	Konstruktor. Initialisierung.
C842	DLL-Funktion	-
	Beschreibung	Ebenfalls Konstruktor beim C-842-Controller. Dort erfolgt die Initialisierung der DLL, und ermöglicht danach den Zugriff auf diese Bibliothek und damit die Kommunikation mit der Controller-Karte.

Tabelle B.22: `TC_832::TC_832(void)`

C832	Klasse	`TC_832`
	Funktion	`SetLimit(DWORD limit)`
	Eigenschaft	privat (`private`)
	Verwendung	`TMotorParamDlg::CanClose(void)`
	Aufrufe	`Hardware->UpdateController`
	Beschreibung	Setzen des `RemoveLimits`.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.23: `TC_832::SetLimit(DWORD limit)`

C832	Klasse	`TC_832`
	Funktion	`StartCheckScan(void)`
	Eigenschaft	geschützt (`protected`)
	Verwendung	`_MOTORCLASS WINAPI mStartMoveScan(int tic, int)`
	Aufrufe	`timeSetEvent; MoveByPosition`
	Beschreibung	Timer starten und Motor bewegen.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.24: `TC_832::StartCheckScan(void)`

C832	Klasse	`TC_832`
	Funktion	`StartLimitWatch(void)`
	Eigenschaft	öffentlich (`public`)
	Verwendung	`TMotorParamDlg::Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)`
	Aufrufe	`Hardware->UpdateController; timeSetEvent`
	Beschreibung	Setzen eines Timers.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.25: `TC_832::StartLimitWatch(void)`

C832	Klasse	`TC_832`
	Funktion	`StopLimitWatch(void)`
	Eigenschaft	öffentlich (`public`)
	Verwendung	`TMotorParamDlg::Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)`
	Aufrufe	`Hardware->UpdateController; timeKillEvent`
	Beschreibung	Löschen eines Timers.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.26: `TC_832::StopLimitWatch(void)`

C832	Klasse	`TC_832`
	Funktion	`OptimizingDlg(void)`
	Eigenschaft	öffentlich (`public`)
	Verwendung	`_MOTORCLASS WINAPI mlOptimizingDlg(void)`
	Aufrufe	`TOptimizeDC_832Dlg; dlg->ExecuteDialog; GetFrameHandle`
	Beschreibung	Dialog zum Optimieren eines Motors (der Parameter).
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.27: `TC_832::OptimizingDlg(void)`

C832	Klasse	`TC_832`
	Funktion	`SaveSettings(int LastSave)`
	Eigenschaft	geschützt (`protected`)

	Verwendung	`TMList::SaveModuleSettings(void);`
		`TCalibrateDlg::Dlg_OnCommand(HWND hwnd, int id,`
		`HWND hwndCtl, UINT codeNotify);`
		`TOptimizeDC_832Dlg::CanClose(void)`

	Aufrufe	`Hardware->UpdateController; WritePrivateProfileString; GetHWFile`
	Beschreibung	Einstellungen in Hardware-Datei sichern.
C842	DLL-Funktion	-
	Beschreibung	-

Tabelle B.28: `TC_832::SaveSettings(int LastSave)`

B.4 Verwendung der Funktionen aus TC_832Controller

Get

Diese Funktion liest aus den Registern der Controller-Karte. Die Funktion `Get` wird in folgenden Methoden aufgerufen:

- `TC_832::IsLimitHit`,
- `TC_832::CheckBoardOk`,
- `TC_832Controller::LimitWatch`,
- `TC_832Controller::Check`,
- `TC_832Controller::Drive628c`,
- `TC_832Controller::GetWord`,
- `TC_832Controller::GetDWord`,
- `TC_832Controller::LM628Ready`.

Put

Diese Funktion schreibt in die Register der Controller-Karte. Die Funktion `Put` wird in folgenden Methoden verwendet:

- `TC_832::IsLimitHit`,
- `TC_832::CheckBoardOk`,
- `TC_832Controller::LimitWatch`,
- `TC_832Controller::Check`,
- `TC_832Controller::Drive628c`,
- `TC_832Controller::GetWord`,

- TC_832Controller::GetDWord,
- TC_832Controller::PutWord,
- TC_832Controller::PutDWord,
- TC_832Controller::LM628Ready.

LimitWatch

LimitWatch ist eine CallBack-Funktion und arbeitet mit

- TC_832::StartLimitWatch und
- TC_832::StopLimitWatch

zusammen.

LM628Ready

Diese Funktion wird verwendet in:

- TC_832::IsLimitHit,
- TC_832Controller::LimitWatch,
- TC_832Controller::GetWord,
- TC_832Controller::GetDWord,
- TC_832Controller::PutWord,
- TC_832Controller::PutDWord.

Die Entsprechung für den Controller C-842 ist QMCready bzw. QMCbusy.

QMCready: falls der Controller C-842 bereit ist, ein neues Kommando oder Daten zu akzeptieren, so ist der Rückgabewert true.

QMCbusy: diese Funktion gibt 0 zurück, wenn das Controller-Board nicht fertig wurde bevor ein timeout aufgetreten ist. Wenn 2500 Versuche fehl schlagen, wird das timeout ausgelöst. Wenn der Controller seine Aktion beendet hat, wird die Anzahl der Versuche zurückgegeben.

Drive628c

Die Funktion realisiert den Hardwarezugriff auf die Controller-Karte und wird verwendet in:

- TC_832::Drive628,
- TC_832Controller::LimitWatch.

UpdateController

Diese Methode wird in nahezu jeder Funktion verwendet und soll dem aktualisieren des Controllers dienen. Die Aufrufe erfolgen in:

- `TC_832::Reset`,
- `TC_832::ActivateFilterParameters`,
- `TC_832::SaveSettings`,
- `TC_832::SetHome`,
- `TC_832::ActivateDrive`,
- `TC_832::StopDrive`,
- `TC_832::StartLimitWatch`,
- `TC_832::StopLimitWatch`,
- `TC_832::IsLimitHit`,
- `TC_832::IsIndexArrived`,
- `TC_832::StartToIndex`,
- `TC_832::SetVelocity`,
- `TC_832::GetVelocity` (Aufruf von `UpdateController` überflüssig, da kein Hardwarezugriff),
- `TC_832::SetAcceleration`,
- `TC_832::GetAcceleration` (überflüssiger Aufruf),
- `TC_832::SetLimit` (überflüssiger Aufruf),
- `TC_832::GetStatus`,
- `TC_832::_GetPosition`,
- `TC_832::_GetFailure`,
- `TC_832::MoveRelative`,
- `TC_832::MoveAbsolute`,
- `TC_832::IsMoveFinish`,
- `TC_832::CheckBoardOk`,
- `TC_832::ExecuteCmd`,
- `TC_832::Drive628`.

Check

Diese Funktion wird nie verwendet. Sie sollte testen, ob die Hardware tatsächlich angeschlossen ist. Für den C-832 existiert dafür die Funktion `CheckBoardOk()`.

Anhang C

LM628 User Command Set

Hier sollen kurz und knapp die wichtigsten Funktionen des User Command Set vom LM628 des Motorcontrollers C-832 beschrieben werden. Dadurch soll die funktionsweise des Programms besser verstanden werden. Für eine ausführliche Beschreibung ist in den Dokumenten von *PI* (siehe Literatur *PhysikInstrumente*) nachzulesen.

C.1 Initialisierung

C.1.1 RESET

Beschreibung	Reset LM628
Kommando Code (hex)	0x00
Datenbytes	keine
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RESET, 0, 0)

Dieses Kommando (und der Hardware Reset Input, Pin 27) bewirkt, dass folgende Daten auf Null zurückgesetzt werden:

- Filterkoeffizienten und deren Input Buffers,
- Trajektorienparameter und deren Input Buffers.

Außerdem werden folgende Einstellungen vorgenommen:

- fünf der sechs Interruptmasken zurückgesetzt (nur SBPA/SBPR bleibt maskiert),
- Output Port Size wird auf 8 Bits festgelegt,
- aktuelle Position wird als Home-Position definiert.

C.1.2 DFH

Beschreibung	Define Home
Kommando Code (hex)	0x02
Datenbytes	keine
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(DFH, 0, 0)

Dieses Kommando deklariert die aktuelle Position als Home-Position bzw. als absolute Null. Wenn DFH während der Bewegung ausgeführt wird, so wird es nicht auf die Position, wo der Motor hält angewendet, sondern erst, wenn auch das Kommando STT ausgeführt wird.

C.2 Interrupts

C.2.1 MSKI

Beschreibung	Mask Interrupts
Kommando Code (hex)	0x1C
Datenbytes	2
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(MSKI, 0, 0)

Dieses Kommando lässt den Nutzer festlegen, welche potentiellen Interrupt-Bedingungen beim Host tatsächlich ein Interrupt auslösen. Bits 1 bis 6 vom Status-Byte sind Indikatoren für die sechs Bedingungen, welche Kandidaten für Host-Interrupts sind. Wenn ein Interrupt ausgelöst wird, dann liest der Host das Status-Byte, um herauszufinden, welche Bedingungen aufgetreten sind.

Das Kommando wird von zwei Datenbytes gefolgt. Bits 1 bis 6 des zweiten Bytes legen den Status (maskiert/unmaskiert) jedes potentiellen Interrupts fest. Jede Null in diesem 6-Bit Feld maskiert den entsprechenden Interrupt. Jede Eins aktiviert den Interrupt. Alle anderen Bits dieser zwei Bytes haben keine Funktion.

Diese Maske regelt nur den Host-Interrupt Prozeß. Das Auslesen des Status-Byte zeigt die aktuellen Bedingungen, unabhängig von der Intterupt-Maske. D.h. es könnte im Status-Byte stehen, dass eine Interrupt-Bedingung erfüllt wurde; wenn aber nun dieses Byte maskiert ist (also auf Null gesetzt), so hat es keine Auswirkungen und löst beim Host keinen Interrupt aus.

Bit Position	**Interrupt**
Bit 15 - 7	nicht genutzt
Bit 6	Breakpoint Interrupt
Bit 5	Position-Error Interrupt
Bit 4	Wrap-Around Interrupt
Bit 3	Index-Pulse Interrupt
Bit 2	Trajectory-Complete Interrupt
Bit 1	Command Error Interrupt
Bit 0	nicht genutzt

C.2.2 RSTI

Beschreibung	Reset Interrupts
Kommando Code (hex)	0x1D
Datenbytes	2
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RSTI, 0, 0)

Wenn eine der potentiellen Interrupt-Bedingungen aus obiger Tabelle auftritt, so wird das Kommando RSTI genutzt, um das korrespondierende Interrupt-Flag-Bit im Status-Byte zurückzusetzen.

C.2.3 SIP

Beschreibung	Set Index Position
Kommando Code (hex)	0x03
Datenbytes	keine
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(SIP, 0, 0)

Nachdem dieses Kommando ausgeführt wurde, wird die absolute Position, die mit dem Auftreten des nächsten Index-Impulses korrespondiert, im Index-Register gespeichert. Außerdem wird Bit 3 vom Status-Byte auf Eins gesetzt.

C.3 Filter

C.3.1 LFIL

Beschreibung	Load Filter Parameters
Kommando Code (hex)	0x1E
Datenbytes	2-10
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(LFIL, wSample | LD_IL, wKL)

Dieses Kommando dient dazu, enstprechende Filterparameter (z.B. Static Gain, Dynamic Gain, Integral Gain, Integral Limit) zu laden. Diese beeinflussen das Beschleunigungs- und Fahrverhalten der Antriebe.

Das angegebene Beispiel lädt das Integral Limit. wSample = 0x0100 entspricht dezimal 512 und gibt damit das Sampling-Intervall von 512μs an. LD_IL = 0x0001 bestimmt, dass das IntegralLimit geladen werden soll. Der zu übermittelnde Wert steht in wKL.

C.3.2 UDF

Beschreibung	Update Filter
Kommando Code (hex)	0x04
Datenbytes	0
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(UDF, 0, 0)

Dieses Kommando muß ausgeführt werden, um die mittels LFIL übermittelten Änderungen aktiv zu schalten.

C.4 Trajektorie

C.4.1 LTRJ

Beschreibung	Load Trajectory Parameters
Kommando Code (hex)	0x1F
Datenbytes	2-14
Ausführbar während Bewegung	teilweise
Anwendungsbeispiel	Drive628(LTRJ, LD_POS, 0)

Die Kontrollparameter der Trajektorie, die an LM628 gesendet werden, um die Bewegung zu kontollieren, sind:

- Beschleunigung
- Geschwindigkeit
- Position

Zusätzlich gibt es die Möglichkeit, diese drei Parameter als absolute oder relative Werte zu übermitteln. Außerdem kann eingestellt werden, ob die Motoren im Positionsmodus oder im Velocity-Modus fahren sollen. Und es besteht die Möglichkeit, den Motor auf verschiedene Arten zu stoppen:

- Stop Smoothly (langsam entschleunigen wie im Profil festgelegt; 0x400)
- Stop Abruptly (sofort Anhalten mit maximaler Entschleunigung; 0x200)
- Turn Off Motor (sicherstellen, dass Motor steht; 0x100)

Bevor mittels dieses Kommandos eine neue Beschleunigung gesetzt werden kann, muss ein "Motor Off"-Kommando (0x100) ausgeführt werden, d.h. es muss sichergestellt sein, dass der Motor auch wirklich steht.

Nachdem der Kommando-Code übermittelt wurde, geben jeweils die ersten zwei Datenbytes an, welcher Parameter geändert werden soll.

Beispiel. Laden der absoluten Position 0.

Drive628(LTRJ, LD_POS, 0)) mit LD_POS = 0x0002

Beispiel. Den Antrieb langsam anhalten/entschleunigen.

Drive628(LTRJ, 0x400, 0)

Beispiel. Eine relative Position laden.

Drive628(LTRJ, LD_RPOS, labs(RemoveLimit)) mit LD_RPOS = 0x0003

Beispiel. Setzen der Geschwindigkeit.

Drive628(LTRJ, 0x0008, dwVelocity)

Beispiel. Setzen der Beschleunigung.

Drive628(LTRJ, 0x0020, dwAcceleration)

C.4.2 STT

Beschreibung	Start Motion
Kommando Code (hex)	0x01
Datenbytes	0
Ausführbar während Bewegung	ja, wenn Beschleunigung nicht geändert wurde
Anwendungsbeispiel	Drive628(STT, 0, 0)

Dieses Kommando wird genutzt, um die gewünschte Trajektorie auszuführen, deren Parameter zuvor mittels LTRJ eingestellt wurden.

Bit Position	Funktion
Bit 7	Motor Off
Bit 6	Breakpoint Reached (Interrupt)
Bit 5	Excessive Position-Error (Interrupt)
Bit 4	Wrap-Around Occured (Interrupt)
Bit 3	Index-Pulse Observed (Interrupt)
Bit 2	Trajectory-Complete (Interrupt)
Bit 1	Command Error (Interrupt)
Bit 0	Busy Bit

Tabelle C.1: Status Register (Bedeutung der einzelnen Bits)

C.5 Report

C.5.1 RDSTAT

Beschreibung	Read Status Byte
Kommando Code (hex)	kein
Datenbytes	1
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RDSTAT, 0, 0)

Hier handelt es sich nicht wirklich um ein Kommando, aber es wird mit den anderen aufgeführt, weil es sehr oft zur Kommunikation mit dem Host genutzt wird. Es wird direkt von der Hardware unterstützt und kann jederzeit ausgeführt werden. Deshalb gibt es keinen Kommando Code. Das Status-Byte wird gelesen, indem bei CS, PS und RD eine logische Null gesetzt wird. Die Tabelle gibt Aufschluss über die Funktion der einzelnen Bits.

C.5.2 RDSIGS

Beschreibung	Read Signals Register
Kommando Code (hex)	0x0C
Datenbytes	2
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RDSIGS, 0, 0) & 0x08

Mit Hilfe dieses Kommandos kann das interne Signalregister ausgelesen werden. Das zweite Byte ist dabei eine Kopie des Status-Bytes (siehe C.1), ausgenommen Bit 0.

Beispiel. Drive628(RDSIGS, 0, 0) & 0x08

Auslesen des Signalregisters und feststellen, ob ein Index-Pulse aufgetreten ist. Genauso gut hätte diese Funktionalität durch Auslesen des Status-Byte erreicht werden können, da es ja im Signalregister dupliziert vorliegt. Der Aufruf hätte dann wie folgt lauten müssen: Drive628(RDSTAT, 0, 0) & 0x08.

C.5.3 RDIP

Beschreibung	Read Index Position
Kommando Code (hex)	0x09
Datenbytes	4
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RDIP, 0, 0)

Bit Position	Funktion
Bit 15	Host Interrupt
Bit 14	Acceleration Loaded (but not updated)
Bit 13	UDF Executed (but filter not yet updated)
Bit 12	Forward Direction
Bit 11	Velocity Mode
Bit 10	On Target
Bit 9	Turn Off Upon Excessive Position Error
Bit 8	8-Bit Output Mode
Bit 7	Motor Off
Bit 6	Breakpoint Reached (Interrupt)
Bit 5	Excessive Position-Error (Interrupt)
Bit 4	Wrap-Around Occured (Interrupt)
Bit 3	Index-Pulse Observed (Interrupt)
Bit 2	Trajectory-Complete (Interrupt)
Bit 1	Command Error (Interrupt)
Bit 0	Acquire Next Index (SIP Executed)

Tabelle C.2: Signal Register (Bedeutung der einzelnen Bits)

Liest die Position, die im Indexregister gespeichert ist. Das Lesen des Indexregisters kann dazu benutzt werden, um Herauszufinden, ob ein Fehler aufgetreten ist. Wann immer das SIP Kommando ausgeführt wird, so muss die neue Indexposition minus der alten Indexposition, dividiert durch die Encoderauflösung (Encoderlinien mal vier), stets eine integrale Zahl sein.

Das Kommando kann genutzt werden, um die Home-Position oder andere spezielle Positionen, zu identifizieren bzw. zu verifizieren.

C.5.4 RDRP

Beschreibung	Read Real Position
Kommando Code (hex)	0x0A
Datenbytes	4
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RDRP, 0, 0)

Das Kommando liest die gegenwärtige aktuelle Position des Motors.

C.5.5 RDDP

Beschreibung	Read Desired Position
Kommando Code (hex)	0x08
Datenbytes	4
Ausführbar während Bewegung	ja
Anwendungsbeispiel	Drive628(RDDP, 0, 0)

Das Kommando liest die gewünschte Position des Profilgenerators, d.h. die errechnete Position, die vom Motor angefahren werden soll.

Anhang D

Dokumentation des Quellcodes

Der gesamte Quellcode der überarbeiteten Klassen, insbesondere von `TC_842`, wurde nahezu vollständig mittels *Doxygen* dokumentiert, um den Quellcode lesbarer und die funktionsweise der einzelnen Routinen verständlicher zu machen. In der Dokumentation werden auch Anmerkungen und Vorschläge zur Verbesserung gemacht.

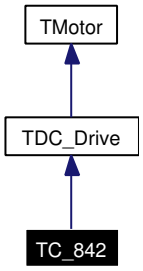
Doxygen generierte für das gesamte *XCTL*-Projekt die Dokumentation, die mehrere hundert Seiten umfasst und im Manual `refman.ps` gespeichert ist. Als Auszug wird hier nur der wichtige Teil für den C-842 gezeigt, der die Seiten 251 bis 265 des Manuals wiedergibt.

Die Graphiken, die die Abhängigkeiten der Klassen untereinander (Klassendiagramm) und zwischen einzelnen Attributen (Kollaborationsdiagramm) zeigen, wurden mit Hilfe von *Dot* erzeugt.

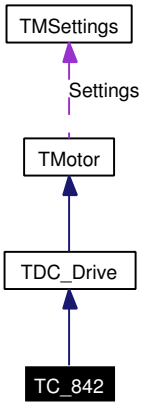
5.41 TC_842 Klassenreferenz

`#include <TC_842.h>`

Klassendiagramm für TC_842:



Zusammengehörigkeiten von TC_842:



Öffentliche Methoden

- **TC_842** (void)
  Initialisieren des Motors C842.
- BOOL **StopLimitWatch** (void)
  Stoppen der Überwachung des Limits.
- BOOL **StartLimitWatch** (void)
  Startet den LimitWatch, d.h.
- void CALLBACK **LimitWatch** (UINT, UINT, DWORD, DWORD, DWORD)
  LimitWatch.
- void **OptimizingDlg** (void)
  Erzeugt einen neuen Dialog, in dem Optimierungseinstellungen für den C842 vorgenommen werden können.

Öffentliche Attribute

- void(∗ **lpfnLimitWatch**)(UINT, UINT, DWORD, DWORD, DWORD)

Geschützte Methoden

- int **SaveSettings** (BOOL)

  Sichern der Einstellungen in der Hardware.ini.

- BOOL **ActivateFilterParameters** (void)

  Einlesen der Filterparameter, die in der Hardware.ini gespeichert sind und übermitteln der Werte an die Karte.

- BOOL **CheckBoardOk** (void)

  Überprüfen, ob die Karte angeschlossen und verfügbar ist.

- BOOL **ActivateDrive** (void)

  Aktiviert den aktuellen Antrieb.

- void **StartCheckScan** (void)
- BOOL **Reset** (void)

  Alte Funktionalität beim C832: Zurücksetzen des Prozessors LM628, der Interrupts und Maskieren der Interrupts.

Geschützte Attribute

- WORD **wKI**
- WORD **wKL**
- WORD **wKP**
- WORD **wKD**

Private Methoden

- BOOL **ExecuteCmd** (LPSTR)

  Führt das angegebene Kommando als Kommando-String aus.

- BOOL **IsMoveFinish** (void)

  Feststellen, ob sich die aktuelle Achse (nOnBoardId) noch in Bewegung befindet.

- BOOL **IsLimitHit** (void)

  Feststellen, ob ein Limit erreicht/überschritten wurde.

- BOOL **IsIndexArrived** (void)

  Feststellen, ob ein Index erreicht wurde.

- BOOL **StartToIndex** (long &)

  Indexposition anfahren.

- BOOL **StopDrive** (BOOL)

  Langsames Anhalten des Motors unter Berücksichtigung der programmierten Beschleunigung, im Gegensatz zum abrupten Stoppen des Motors.

- BOOL **SetLimit** (DWORD)

  Setzen des RemoveLimits.

- BOOL **SetVelocity** (DWORD)

  Einstellen der Geschwindigkeit.

- DWORD **GetVelocity** (void)

  Ermitteln der gesetzten Geschwindigkeit für den aktuellen Antrieb (nOnBoardId).

- BOOL **SetHome** (void)

  Aktuelle Position als Null-Position festlegen.

- BOOL **SetAcceleration** (DWORD)

  Einstellen der Beschleunigung.

- DWORD **GetAcceleration** (void)

  Ermitteln der gesetzten Beschleunigung für den aktuellen Antrieb (nOnBoardId).

- WORD **GetStatus** (void)

  Gibt das Statuswort zurück.

- BOOL **_GetPosition** (long *)

  Bestimmen der gegenwärtigen aktuellen Position des gewählten Motors (nOnBoardId).

- BOOL **_GetFailure** (long *)

  Ermittelt die gegenwärtige Anzahl von Fehlercounts, d.h.

- BOOL **MoveRelative** (long)

  Startet den aktuellen Antrieb (nOnBoardId) und fährt die relative Position an, ausgehend von der aktuellen Position.

- BOOL **MoveAbsolute** (long)

  Startet den aktuellen Antrieb (nOnBoardId) und fährt die absolute Position an.

Private Attribute

- WORD **wBaseAddr**
- int **nOnBoardId**
- BYTE **cConfig**
- WORD **wSample**
- WORD **baddr**
- UINT **nEvent**
- boolean **bLimitHit**

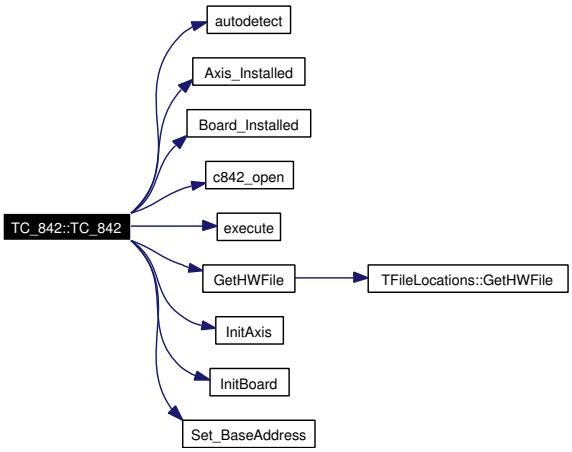
5.41.1 Beschreibung der Konstruktoren und Destruktoren

5.41.1.1 TC_842::TC_842 (void)

Initialisieren des Motors C842.

Es erscheint eine Fehlermeldung (Popup), falls keine Verbindung mittels DLL-Bibliothek zur Karte aufgebaut werden konnte, d.h. wenn **c842_open()**(S. 1031) fehl schlägt.

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2 Dokumentation der Elementfunktionen

5.41.2.1 BOOL TC_842::_GetFailure (long * *failure*) [private, virtual]

Ermittelt die gegenwärtige Anzahl von Fehlercounts, d.h.

die Abweichung der realen Position von der virtuellen Position.

Parameter:

**failure* Anzahl der Fehlercounts.

Rückgabe:

true.

Noch zu erledigen

Anstatt stets true als Rückgabewert zu liefern, könnte die Anzahl der Fehlercounts zurückgegeben werden, und somit wäre der Parameter *failure überflüssig.

Erneute Implementation von **TDC_Drive** (S. 368).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.2 BOOL TC_842::_GetPosition (long * *position*) [private, virtual]

Bestimmen der gegenwärtigen aktuellen Position des gewählten Motors (nOnBoardId).

Parameter:
**position* Die Position des Motors als Rückgabe.

Rückgabe:
true.

Noch zu erledigen
Die Position als Rückgabewert zurückliefern und damit den Parameter einsparen. Oder den überflüssigen Rückgabewert true entfernen und die Funktion auf void ändern.

Erneute Implementation von **TDC_Drive** (S. 369).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.3 BOOL TC_842::ActivateDrive (void) [protected, virtual]

Aktiviert den aktuellen Antrieb.

Rückgabe:
true.

Veraltet
Eine alte Funktion vom C832, die der Einfachheit halber auch für den C842 übernommen wurde, um kaum Änderungen im Quelltext an den Stellen vornehmen zu müssen, an denen diese Funktion verwendet wird. Es wird keine Aktion ausgeführt (Dummy-Methode für C842). Im alten Code für den C832 war folgende Funktionsweise implementiert:

- Update der Filterparameter - aktiviert die Parameter, die zuvor mittels der entsprechenden Befehle an die Karte übermittelt wurden
- Auslesen der Motorposition von der Karte
- Erneutes Laden genau dieser Position
- Starten der Motorbewegung (in diesem Fall keine, da Positionen übereinstimmen)

Noch zu erledigen
Diese Funktion diente scheinbar nur dem Aktivieren der eingelesenen Filterparameter und einer Bewegung des Antriebs um 0. Das Aktivieren der Filterparameter ist auch beim C842 sicherzustellen. Das wird aber auch schon bei der weiteren Funktion **ActivateFilterParameters()**(S. 256) ausgeführt. Zusätzlich werden dort die Parameter eingelesen und danach auf der Karte aktiviert. Dazu diente der Aufruf von Drive628(UDF,0,0) beim C832.

Erneute Implementation von **TDC_Drive** (S. 369).

5.41.2.4 BOOL TC_842::ActivateFilterParameters (void) [protected, virtual]

Einlesen der Filterparameter, die in der Hardware.ini gespeichert sind und übermitteln der Werte an die Karte.

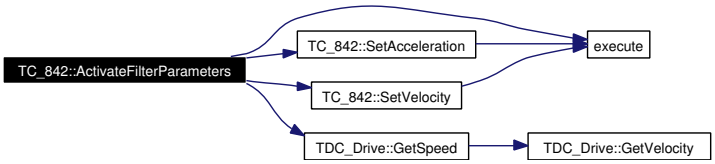
Anschließend Aktivieren der Filterparameter.

Rückgabe:
true.

Noch zu erledigen
Eventuell weitere Filterparamter, die für den C842 relevant sind, hinzufügen.

Erneute Implementation von **TDC_Drive** (S. 369).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.5 BOOL TC_842::CheckBoardOk (void) [protected, virtual]

Überprüfen, ob die Karte angeschlossen und verfügbar ist.

Rückgabe:
true.

Noch zu erledigen
Entweder keinen Rückgabewert verwenden, da der Status der Karte in bControlBoardOk gehalten wird, oder Rückgabe des Status der Karte. In diesem Fall true, wenn eine C-842-Karte in einem PC-Slot installiert ist, ansonsten false.

Erneute Implementation von **TDC_Drive** (S. 369).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.6 BOOL TC_842::ExecuteCmd (LPSTR *command*) [private, virtual]

Führt das angegebene Kommando als Kommando-String aus.

Diese Funktion wird bisher nicht verwendet. Die DLL-Bibliothek stellt aber eine ähnliche Funktion zur Verfügung, die damit angesprochen werden könnte: translate. Diese ist aber nicht zu verwechseln mit der Funktion execute aus der DLL-Bibliothek. Ein beispielhafter Aufruf könnte lauten: translate("1MA10000", report); oder mit Hilfe dieser Funktion: ExecuteCmd("1MA10000");

Parameter:

command Der Kommando-String in der Form aCn, wobei a den Motor identifiziert (a = 0, 1, 2). Bei a = 0 werden beide Motoren gleichzeitig angesprochen. C ist das Kommando (siehe verfügbare Kommandos, z.B. MA für MoveAbsolute). n gibt den Wert des Parameters für das entsprechende Kommando an.

Rückgabe:

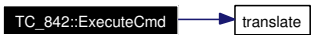
true.

Noch zu erledigen

Den Rückgabewert ändern, so dass der Report des ausgeführten Kommandos als String zurück geliefert wird.

Erneute Implementation von **TMotor** (S. 613).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.7 DWORD TC_842::GetAcceleration (void) [private, virtual]

Ermitteln der gesetzten Beschleunigung für den aktuellen Antrieb (nOnBoardId).

Rückgabe:

Beschleunigung.

Erneute Implementation von **TDC_Drive** (S. 369).

5.41.2.8 WORD TC_842::GetStatus (void) [private, virtual]

Gibt das Statuswort zurück.

Rückgabe:

Statuswort. (unsigned short integer)

Erneute Implementation von **TMotor** (S. 616).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.9 DWORD TC_842::GetVelocity (void) [private, virtual]

Ermitteln der gesetzten Geschwindigkeit für den aktuellen Antrieb (nOnBoardId).

Rückgabe:

Geschwindigkeit.

Noch zu erledigen

Es genügt, wenn nur beim Setzen der Geschwindigkeit überprüft wird, ob nicht die maximale Geschwindigkeit überschritten wird.

Erneute Implementation von **TDC_Drive** (S. 370).

5.41.2.10 BOOL TC_842::IsIndexArrived (void) [private, virtual]

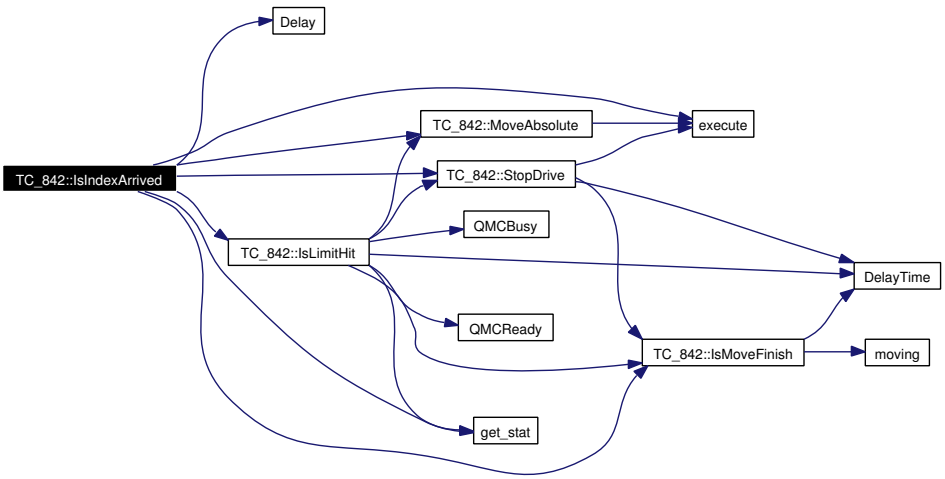
Feststellen, ob ein Index erreicht wurde.

Rückgabe:

true, falls Index erreicht. False, sonst.

Erneute Implementation von **TMotor** (S. 617).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.11 BOOL TC_842::IsLimitHit (void) [private, virtual]

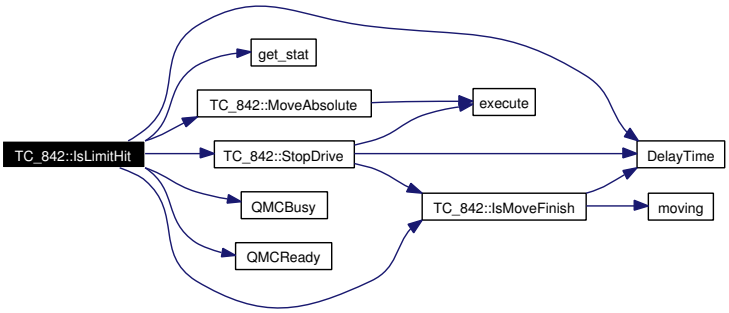
Feststellen, ob ein Limit erreicht/überschritten wurde.

Rückgabe:

true, falls Limit erreicht wurde. False sonst.

Erneute Implementation von **TDC_Drive** (S. 370).

Hier ist der Graph aller Aufrufe für diese Funktion:



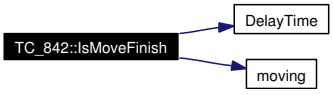
5.41.2.12 BOOL TC_842::IsMoveFinish (void) [private, virtual]

Feststellen, ob sich die aktuelle Achse (nOnBoardId) noch in Bewegung befindet.

Rückgabe:
true, falls die Achse nicht mehr in Bewegung ist. Ansonsten false.

Erneute Implementation von **TDC_Drive** (S. 371).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.13 void CALLBACK TC_842::LimitWatch (UINT *id*, UINT *u1*, DWORD *u2*, DWORD *u3*, DWORD *u4*)

LimitWatch.

Parameter:
id Identifiziert Timer-Event.

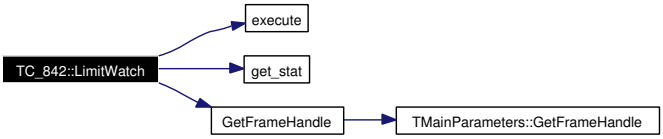
u1 not used.

u2 not used.

u3 not used.

u4 not used.

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.14 BOOL TC_842::MoveAbsolute (long *position*) [private, virtual]

Startet den aktuellen Antrieb (nOnBoardId) und fährt die absolute Position an.

Parameter:
position Absolute Zielposition.

Rückgabe:
true.

Noch zu erledigen
Ändern des Rückgabewerts auf String, damit der Report des Kommandos zurückgegeben werden kann. Rückgabe von immer true ohne Sinn.

Erneute Implementation von **TDC_Drive** (S. 371).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.15 BOOL TC_842::MoveRelative (long *position*) [private, virtual]

Startet den aktuellen Antrieb (nOnBoardId) und fährt die relative Position an, ausgehend von der aktuellen Position.

Parameter:
position Relative Zielposition.

Rückgabe:
true.

Noch zu erledigen
Ändern des Rückgabewerts auf String, damit der Report des Kommandos zurückgegeben werden kann. Rückgabe von immer true ohne Sinn.

Erneute Implementation von **TDC_Drive** (S. 371).

Hier ist der Graph aller Aufrufe für diese Funktion:

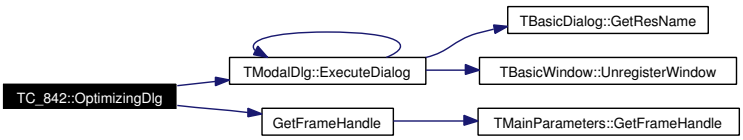
5.41.2.16 void TC_842::OptimizingDlg (void) [virtual]

Erzeugt einen neuen Dialog, in dem Optimierungseinstellungen für den C842 vorgenommen werden können.

Siehe auch:
TOptimizeDC_842Dlg::TOptimizeDC_842Dlg(void) (S. 666)

Erneute Implementation von **TMotor** (S. 618).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.17 BOOL TC_842::Reset (void) [protected, virtual]

Alte Funktionalität beim C832: Zurücksetzen des Prozessors LM628, der Interrupts und Maskieren der Interrupts.

Entspricht einer neuen Initialisierung der gesamten Karte?!

Rückgabe:
true.

Noch zu erledigen
Überprüfen, ob die Verwendung von InitBoard(0x09) der gewünschten Funktionalität entspricht. Ändern des Rückgabewertes. InitBoard liefert den Status der Karte als Integer.

Erneute Implementation von **TDC_Drive** (S. 372).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.18 int TC_842::SaveSettings (BOOL *LastSave*) [protected, virtual]

Sichern der Einstellungen in der Hardware.ini.

Speziell werden die Filterparameter, Geschwindigkeit und Beschleunigung des aktuellen Motors gespeichert.

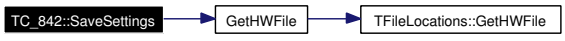
Parameter:
LastSave Falls true, so werden noch weitere Parameter in der Datei gespeichert. (siehe int **TMotor::SaveSettings(BOOL LastSave)**(S. 619)) Wenn false, dann werden nur die hier angegebenen Werte gesichert.

Rückgabe:
false, wenn bControlBoardOK false ist, ansonsten immer true.

Siehe auch:
TMotor::SaveSettings(BOOL LastSave)(S. 619)

Erneute Implementation von **TDC_Drive** (S. 372).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.19 BOOL TC_842::SetAcceleration (DWORD *acceleration*) [private, virtual]

Einstellen der Beschleunigung.

Parameter:
acceleration Wert der Beschleunigung.

Rückgabe:
true.

Noch zu erledigen
Rückgabewert ändern und eventuell den Report des execute-Kommandos zurückliefern. Alternativ den übermittelten Wert der Beschleunigung von der Karte auslesen und auf Übereinstimmung überprüfen. Falls ungleich, so schlug das Kommando fehl und Rückgabe des Wertes false, ansonsten true. Nachtrag: vermutlich läßt sich die Beschleunigung nicht von der Karte auslesen, es wurde kein entsprechendes Kommando gefunden. Was passiert, wenn die Beschleunigung zu hoch ist? Bei der Geschwindigkeit wurde es zuvor überprüft, hier nicht.

Erneute Implementation von **TDC_Drive** (S. 372).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.20 BOOL TC_842::SetHome (void) [private, virtual]

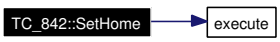
Aktuelle Position als Null-Position festlegen.

Rückgabe:
true.

Noch zu erledigen
Überflüssigen Rückgabewert ändern. Report zurückgeben.

Erneute Implementation von **TMotor** (S. 621).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.21 BOOL TC_842::SetLimit (DWORD *limit*) [private, virtual]

Setzen des RemoveLimits.

Parameter:
limit Wert für RemoveLimit.

Rückgabe:
true.

Erneute Implementation von **TMotor** (S. 621).

5.41.2.22 BOOL TC_842::SetVelocity (DWORD *velocity*) [private, virtual]

Einstellen der Geschwindigkeit.

Parameter:
velocity Wert der Geschwindigkeit.

Rückgabe:
true.

Noch zu erledigen
Rückgabewert ändern und eventuell den Report des execute-Kommandos zurückliefern. Alternativ den übermittelten Wert der Geschwindigkeit von der Karte auslesen und auf Übereinstimmung überprüfen. Falls ungleich, so schlug das Kommando fehl und Rückgabe des Wertes false, ansonsten true. Nachtrag: vermutlich läßt sich die Geschwindigkeit nicht von der Karte auslesen, es wurde kein entsprechendes Kommando gefunden.

Erneute Implementation von **TDC_Drive** (S. 373).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.23 void TC_842::StartCheckScan (void) [protected, virtual]

Erneute Implementation von **TMotor** (S. 621).

Hier ist der Graph aller Aufrufe für diese Funktion:

5.41.2.24 BOOL TC_842::StartLimitWatch (void) [virtual]

Startet den LimitWatch, d.h.

es wird beobachtet, ob irgendwann ein Limit erreicht/überschritten wird.

Rückgabe:
false, falls die Beobachtung schon gestartet wurde und aktiv ist; ansonsten true, nach dem Starten der Beobachtung.

Erneute Implementation von **TMotor** (S. 622).

5.41.2.25 BOOL TC_842::StartToIndex (long & *oldPos*) [private, virtual]

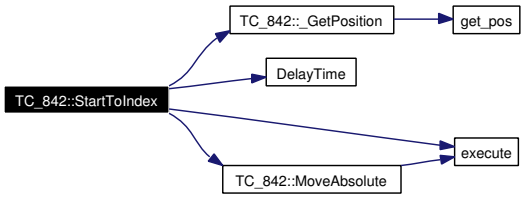
Indexposition anfahren.

Parameter:
&oldPos alte Position vor dem Anfahren der Indexposition.

Rückgabe:
true.

Erneute Implementation von **TMotor** (S. 622).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.26 BOOL TC_842::StopDrive (BOOL *bOffOn*) [private, virtual]

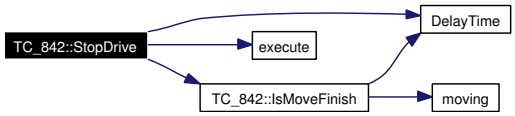
Langsames Anhalten des Motors unter Berücksichtigung der programmierten Beschleunigung, im Gegensatz zum abrupten Stoppen des Motors.

Parameter:
bOffOn Not used.

Rückgabe:
true.

Erneute Implementation von **TMotor** (S. 622).

Hier ist der Graph aller Aufrufe für diese Funktion:



5.41.2.27 BOOL TC_842::StopLimitWatch (void) [virtual]

Stoppen der Überwachung des Limits.

Rückgabe:
false, falls keine Überwachung aktiv ist; true, wenn die aktive Überwachung beendet wurde.

Erneute Implementation von **TMotor** (S. 622).

5.41.3 Dokumentation der Datenelemente

5.41.3.1 WORD TC_842::baddr [private]

5.41.3.2 boolean TC_842::bLimitHit [private]

5.41.3.3 BYTE TC_842::cConfig [private]

5.41.3.4 void(* TC_842::lpfnLimitWatch)(UINT, UINT, DWORD, DWORD, DWORD)

5.41.3.5 UINT TC_842::nEvent [private]

5.41.3.6 int TC_842::nOnBoardId [private]

5.41.3.7 WORD TC_842::wBaseAddr [private]

5.41.3.8 WORD TC_842::wKD [protected]

5.41.3.9 WORD TC_842::wKI [protected]

5.41.3.10 WORD TC_842::wKL [protected]

5.41.3.11 WORD TC_842::wKP [protected]

5.41.3.12 WORD TC_842::wSample [private]

Die Dokumentation für diese Klasse wurde erzeugt aufgrund der Dateien:

- TC_842.h
- M_LAYER.CPP
- MOTORS.CPP

Anhang E

Quellcode

Es werden die Quellen der Klassen `TC_842` und `TOptimizeDC_842Dlg` dokumentiert, an denen die meisten Änderungen im Code vorgenommen worden sind.

```
#ifndef __TC_842_H
#define __TC_842_H

#include "utils\u_utils.h"
#include "motrstrg\motorcontroller.h"
#include "motrstrg\M_MOTHW.h"

class TC_842 : public TDC_Drive
{
        friend class TOptimizeDC_842Dlg;

public:
        TC_842( void );

        BOOL StopLimitWatch( void );
        BOOL StartLimitWatch( void );
        void CALLBACK LimitWatch( UINT, UINT, DWORD, DWORD, DWORD );
        void ( *lpfnLimitWatch ) ( UINT, UINT, DWORD, DWORD, DWORD );
        void OptimizingDlg( void );

protected:
        int SaveSettings( BOOL );
        BOOL ActivateFilterParameters( void );
        BOOL CheckBoardOk( void );
        BOOL ActivateDrive( void );
        void StartCheckScan( void );
        BOOL Reset( void );
        WORD wKI;    //  IntegralGain
        WORD wKL;    //  IntegralLimit
        WORD wKP;    //  StaticGain
        WORD wKD;    //  DynamicGain

private:
        BOOL ExecuteCmd( LPSTR );
        BOOL IsMoveFinish( void );
        BOOL IsLimitHit( void );
        BOOL IsIndexArrived( void );
        BOOL StartToIndex( long & );
        BOOL StopDrive( BOOL );
        BOOL SetLimit( DWORD );
        BOOL SetVelocity( DWORD );
        DWORD GetVelocity( void );
        BOOL SetHome( void );
        BOOL SetAcceleration( DWORD );
        DWORD GetAcceleration( void );
        WORD GetStatus( void );
        BOOL _GetPosition( long * );
        BOOL _GetFailure( long * );
        BOOL MoveRelative( long );
        BOOL MoveAbsolute( long );

        WORD wBaseAddr;
        int nOnBoardId;
        BYTE cConfig;
        WORD wSample;
        WORD baddr;

        UINT nEvent;
        boolean bLimitHit;
};

#endif __TC_842_H
```

```
/** Initialisieren des Motors C842. Es erscheint eine Fehlermeldung (Popup),
 *  falls keine Verbindung mittels DLL-Bibliothek
 *  zur Karte aufgebaut werden konnte, d.h. wenn c842_open() fehl schlägt.
 */
TC_842::TC_842( void ) : TDC_Drive()
{
        char Ident[MaxString], buf[MaxString];

        // Parameter aus Hardware.ini einlesen
        // Motor-Identifier
        sprintf(Ident, "motor%d", nId);
        // IO-Adresse
        GetPrivateProfileString(Ident, "IOAddr", "0x220", (LPSTR)buf, MaxString,
                GetHWFile());
        sscanf(buf, "%x", &wBaseAddr);
        // Offset für Basisadresse
        baddr = 0;
        // BoardID
        nOnBoardId = GetPrivateProfileInt(Ident, "BoardId", 0, GetHWFile());
        // IntegralGain
        wKI = (WORD)GetPrivateProfileInt(Ident, "IntegralGain", 10, GetHWFile());
        // IntegralLimit
        wKL = (WORD)GetPrivateProfileInt(Ident, "IntegralLimit", 10, GetHWFile());
        // Gain
        wKP = (WORD)GetPrivateProfileInt(Ident, "Gain", 100, GetHWFile());
        // DynamicGain
        wKD = (WORD)GetPrivateProfileInt(Ident, "DynamicGain", 37, GetHWFile());

        cConfig = 0x80;
        if (GetPrivateProfileInt(Ident, "DifferentialEncoder", 0, GetHWFile()))
        {
                if (0 == nOnBoardId)
                        cConfig |= 0x10;
                else
                        cConfig |= 0x20;
        }
        if (GetPrivateProfileInt(Ident, "EnableInterrupts", 0, GetHWFile()))
        {
                cConfig |= 0x40;
        }
        // Sample-Frequenz
        wSample = 0x0100; //=512

        int  i;
        int  limitmode;
        BYTE MaxAxis;

        char report[80];

        // Fehlermeldung, wenn DLL nicht geöffnet werden konnte
        if (!c842_open()) {
                printf("\n\nError while opening UIO or C842\n\n");
                MessageBox(0,"Treiber konnten für C842 nicht aus DLL geladen werden.",
                        "Open DLL",MB_ICONERROR | MB_OK);
                return;
        }

        // Basisadresse setzen: select board at address h220
        Set_BaseAddress(wBaseAddr);
        // maximale Anzahl der vorhandenen Achsen
        MaxAxis = Axis_Installed();
```



```
    if (Board_Installed()) {
                // initialisieren der Karte
                InitBoard(0x09);
                limitmode = autodetect();

            // alle Achsen jeweils initialisieren
            for (i=1; i <= MaxAxis; i++)
               InitAxis(i);

                // falls ein externer PWM-Amplifier benutzt wird:
        //execute(1,SOP,0,report);

                // set capture home
        execute(nOnBoardId,SCH,0,report);

                // set proportional gain
        execute(nOnBoardId,DP,150,report);

                // set velocity
                execute(nOnBoardId,SV,110000,report);

                // set accelaration
        execute(nOnBoardId,SA,300,report);
        }
}
;

/** Alte Funktionalität beim C832: Zurücksetzen des Prozessors LM628, der
 *  Interrupts und Maskieren der Interrupts.
 *  Entspricht einer neuen Initialisierung der gesamten Karte?!
 *  @return true.
 *  @todo Überprüfen, ob die Verwendung von InitBoard(0x09) der gewünschten
 *  Funktionalität entspricht.
 *  Ändern des Rückgabewertes. InitBoard liefert den Status der Karte als
 *  Integer.
 */
BOOL TC_842::Reset( void )
{
        InitBoard(0x09);
        return true;
};

/** Erzeugt einen neuen Dialog, in dem Optimierungseinstellungen für den
 *  C842 vorgenommen werden können.
 *  @see TOptimizeDC_842Dlg::TOptimizeDC_842Dlg(void)
 */
void TC_842::OptimizingDlg( void )
{
        TModalDlg* dlg;

        dlg = (TOptimizeDC_842Dlg *) new TOptimizeDC_842Dlg();
        dlg->ExecuteDialog(GetFrameHandle());
        delete dlg;
};

/** Einlesen der Filterparameter, die in der Hardware.ini gespeichert sind
 *  und übermitteln der Werte an die Karte. Anschließend Aktivieren der
 *  Filterparameter.
 *  @return true.
 *  @todo Eventuell weitere Filterparamter, die für den C842 relevant sind,
 *  hinzufügen.
 */
```

```
BOOL TC_842::ActivateFilterParameters(void)
{
        char report[80];

        // definiere Proportional Gain (== Static Gain)
        execute(nOnBoardId,DP,wKP,report);
    // definiere Derivative Gain (== Dynamic Gain)
        execute(nOnBoardId,DD,wKD,report);
        // definiere Integral Gain
        execute(nOnBoardId,DI,wKI,report);
        // definiere Integration Limit (== Integral Limit)
        execute(nOnBoardId,DL,wKL,report);

        SetVelocity(dwMaxVelocity);
        SetAcceleration(dwAcceleration);
        GetSpeed();

        return true;
};

/** Sichern der Einstellungen in der Hardware.ini. Speziell werden die
 *  Filterparameter, Geschwindigkeit und Beschleunigung des aktuellen Motors
 *  gespeichert.
 *  @param LastSave Falls true, so werden noch weitere Parameter in der Datei
 *  gespeichert. (siehe int TMotor::SaveSettings( BOOL LastSave ))
 *  Wenn false, dann werden nur die hier angegebenen Werte gesichert.
 *  @return false, wenn bControlBoardOK false ist, ansonsten immer true.
 *  @see TMotor::SaveSettings(BOOL LastSave)
 */
int TC_842::SaveSettings(BOOL LastSave)
{
        char buf[MaxString];
        char Ident[MaxString];

        sprintf(Ident,"motor%d",nId);
        sprintf(buf,"%u",wKD);
        WritePrivateProfileString(Ident,"DynamicGain",  buf,GetHWFile());
        sprintf(buf,"%u",wKP);
        WritePrivateProfileString(Ident,"Gain",         buf,GetHWFile());
        sprintf(buf,"%u",wKI);
        WritePrivateProfileString(Ident,"IntegralGain", buf,GetHWFile());
        sprintf(buf,"%u",wKL);
        WritePrivateProfileString(Ident,"IntegralLimit",buf,GetHWFile());
        sprintf(buf,"%lu",dwMaxVelocity);
        WritePrivateProfileString(Ident,"MaxVelocity",  buf,GetHWFile());
        sprintf(buf,"%lu",dwAcceleration);
        WritePrivateProfileString(Ident,"Acceleration", buf,GetHWFile());
        sprintf(buf,"%lu",dwVelocity);
        WritePrivateProfileString(Ident,"Velocity",     buf,GetHWFile());

        return TDC_Drive::SaveSettings(LastSave);
};

/** Aktuelle Position als Null-Position festlegen.
 *  @return true.
 *  @todo Überflüssigen Rückgabewert ändern. Report zurückgeben.
 */
BOOL TC_842::SetHome(void)
{
    char report[80];
        execute(nOnBoardId,DH,0,report);
        return true;
```



```
};

/** Aktiviert den aktuellen Antrieb.
 *  @return true.
 *  @deprecated Eine alte Funktion vom C832, die der Einfachheit halber auch
 *  für den C842 übernommen wurde, um kaum Änderungen im Quelltext an den
 *  Stellen vornehmen zu müssen, an denen diese Funktion verwendet wird.
 *  Es wird keine Aktion ausgeführt (Dummy-Methode für C842).
 *  Im alten Code für den C832 war folgende Funktionsweise implementiert:
 *  <ul>
 *  <li>Update der Filterparameter - aktiviert die Parameter, die zuvor
 *      mittels der entsprechenden Befehle an die Karte übermittelt wurden</li>
 *  <li>Auslesen der Motorposition von der Karte</li>
 *  <li>Erneutes Laden genau dieser Position</li>
 *  <li>Starten der Motorbewegung (in diesem Fall keine,
 *      da Positionen übereinstimmen)</li>
 *  </ul>
 *  @todo Diese Funktion diente scheinbar nur dem Aktivieren der eingelesenen
 *  Filterparameter und einer Bewegung des Antriebs um 0.
 *  Das Aktivieren der Filterparameter ist auch beim C842 sicherzustellen.
 *  Das wird aber auch schon bei der weiteren Funktion
 *  ActivateFilterParameters() ausgeführt. Zusätzlich werden dort
 *  die Parameter eingelesen und danach auf der Karte aktiviert.
 *  Dazu diente der Aufruf von Drive628(UDF,0,0) beim C832.
 */
BOOL TC_842::ActivateDrive(void)
{
        /* C832
        long pos;
        Hardware->UpdateController(cConfig,nOnBoardId);
        Drive628( UDF, 0, 0 );
        _GetPosition( &pos );
        Drive628( LTRJ, LD_POS, pos );
        Drive628( STT, 0, 0 );
        */

        return true;
};

/** Langsames Anhalten des Motors unter Berücksichtigung der programmierten
 *  Beschleunigung, im Gegensatz zum abrupten Stoppen des Motors.
 *  @param bOffOn Not used.
 *  @return true.
 */
BOOL TC_842::StopDrive(BOOL bOffOn)
{
        char report[80];

        execute(nOnBoardId,ST,0,report);

        while(!IsMoveFinish())
                DelayTime(4);

        return true;
};

/** Startet den LimitWatch, d.h. es wird beobachtet, ob irgendwann ein Limit
 *  erreicht/überschritten wird.
 *  @return false, falls die Beobachtung schon gestartet wurde und aktiv ist;
 *  ansonsten true, nach dem Starten der Beobachtung.
 */
BOOL TC_842::StartLimitWatch(void)
```

```
{
        if (bLimitWatchActive)
                return false;
        //nEvent = timeSetEvent(150,1,(LPTIMECALLBACK) &TC_842::LimitWatch,
        //      0, TIME_PERIODIC);
        bLimitWatchActive = true;
        return true;
};

/** LimitWatch.
 *  @param id Identifiziert Timer-Event.
 *  @param u1 not used.
 *  @param u2 not used.
 *  @param u3 not used.
 *  @param u4 not used.
 */
void CALLBACK TC_842::LimitWatch(UINT id,UINT u1,DWORD u2,DWORD u3,DWORD u4)
{
        char report[80];
        BYTE status = get_stat(baddr + 1);;
        static count = 0;
        static MessageIsPosted = 0;

        bLimitHit = 0;
        count++;

        PostMessage( GetFrameHandle( ),WM_COMMAND,cm_SetWatchIndicator,
                ( LPARAM )  count );
        if( status & 0x01 )
        {
                execute(nOnBoardId,ST,0,report);
            bLimitHit = 1;
        }
        if( status & 0x02 )
        {
                execute(nOnBoardId,ST,0,report);
            bLimitHit = 2;
        }
        if(bLimitHit && !MessageIsPosted)
            PostMessage(GetFrameHandle( ),WM_COMMAND,cm_LimitHitOccure,
                        ( LPARAM ) bLimitHit);
};


/** Stoppen der Überwachung des Limits.
 *  @return false, falls keine Überwachung aktiv ist; true, wenn die aktive
 *  Überwachung beendet wurde.
 */
BOOL TC_842::StopLimitWatch(void)
{
        if (!bLimitWatchActive)
                return false;
        timeKillEvent(nEvent);
        bLimitWatchActive = false;

        return true;
};

/** Feststellen, ob ein Limit erreicht/überschritten wurde.
 *  @return true, falls Limit erreicht wurde. False sonst.
 */
BOOL TC_842::IsLimitHit(void)
```



```
{
  // Status auslesen
  BYTE status = get_stat(baddr + 1);

  // Interrupt-Register warten
  QMCReady();

  if(status & (nOnBoardId+1))
  {
    // Motor langsam anhalten/entschleunigen
        StopDrive(true);

    // bUpwards gibt Bewegungsrichtung des Motors an?
    if(bUpwards)
      // Position anfahren (negativer Betrag des RemoveLimits)
      MoveAbsolute(-labs(dwRemoveLimit));
    else
      // Postition anfahren (Betrag des RemoveLimits)
      MoveAbsolute(labs(dwRemoveLimit));

    // nach erreichen Position, wird die Bewegungsrichtung
    // des Motors umgekehrt
    bUpwards = !bUpwards;

    // Warten, bis der Motor still steht
    while(!IsMoveFinish())
      DelayTime(10);
    DelayTime(100);

    // Warten, das Busy-Bit im Interruptregister nicht
    // gesetzt ist
        QMCBusy();

        // aktuellen Status setzen
    bLimitHit = bRangeHit = true;

    return true;
  }

  bLimitHit = false;
  return false;
};

/** Feststellen, ob ein Index erreicht wurde.
 *  @return true, falls Index erreicht. False, sonst.
 */
BOOL TC_842::IsIndexArrived(void)
{
        char report[80];
        long ipos, pos;

        if (IsLimitHit())
        {
                // Zurücksetzen der Interrupts
                execute(nOnBoardId,RTI,0,report);

                if (bIndexLine)
                {
                        // Nachdem dieses Kommando ausgeführt wurde, wird die ab
solute
                        // Position, die mit dem Auftreten des nächsten Index-Im
pulses
```

```
                        // korrespondiert, im Index-Register gespeichert.
                        // Außerdem wird Bit 3 vom Status-Byte auf Eins gesetzt
                        execute(nOnBoardId,SCI,0,report);

                        // gigantische Position anfahren
                        MoveAbsolute(4000000001);

                        // Bewegungsrichtung festlegen
                        bUpwards = true;

                        // Index nicht erreicht
                        return false;
                }
                return true;
        }
        if (bIndexLine && bIndexDetected && IsMoveFinish())
                return true;

        if (!bIndexLine && bRangeHit && IsMoveFinish())
                return true;

        if (!bRangeHit && bMoveFirstToLimit)
                return false;

        // Auslesen des Signalregisters/Hardware-Interrupts und feststellen,
        // ob ein Index-Pulse aufgetreten ist
        if (0x08 & get_stat(baddr + 1))
        {
                // Motor langsam anhalten
                StopDrive(true);

                // Index festgestellt
                bIndexDetected = true;

                // Zurücksetzen der Interrupts
                execute(nOnBoardId,RTI,0,report);

                // liest die Position, die im Indexregister gespeichert ist
                ipos = execute(nOnBoardId,TX,0,report);

                // warte solange, bis der Motor gehalten hat
                while (!IsMoveFinish());

                // gegenwärtige aktuelle Position des Motors auslesen
                pos = execute(nOnBoardId,TP,0,report);

                // Bewegungsrichtung feststellen
                bUpwards = ( ipos > pos );

                // Indexposition anfahren
                MoveAbsolute(ipos);
                Delay(150);

                return false;
        }
        return false;
};

/** Indexposition anfahren.
 *  @param &oldPos alte Position vor dem Anfahren der Indexposition.
 *  @return true.
 */
```



```
BOOL TC_842::StartToIndex(long &oldPos)
{
        char report[80];

        // sehr hohe (maximale) Geschwindigkeit setzen
        SetSpeed(10000.0);

        // keinen Index erkannt, Bereich nicht überschritten, kein Limit erreich
t
        bIndexDetected = false;
        bRangeHit      = false;
        bLimitHit      = false;
        lDeltaPosition = 0;

        SetCalibrationState(false);
        SetCorrectionState(false);

        // gegenwärtige aktuelle Position des Motors auslesen
        _GetPosition(&lPosition);
        // Bewegungsrichtung
        bUpwards = false;

        if (!bMoveFirstToLimit && bIndexLine)
        {
                // Zurücksetzen der Interrupts
                execute(nOnBoardId,RTI,0,report);

                // Nachdem dieses Kommando ausgeführt wurde, wird die absolute
                // Position, die mit dem Auftreten des nächsten Index-Impulses
                // korrespondiert, im Index-Register gespeichert.
                // Außerdem wird Bit 3 vom Status-Byte auf Eins gesetzt
                execute(nOnBoardId,SCI,0,report);
        }

        // entgegengesetzte gigantische relative Position laden
        // (siehe auch isIndexArrived)
        MoveAbsolute(-4000000001);

        // Erreichen der Stillstands-Position
        DelayTime(150);

        return true;
};

/** Einstellen der Geschwindigkeit.
 *  @param velocity Wert der Geschwindigkeit.
 *  @return true.
 *  @todo Rückgabewert ändern und eventuell den Report des execute-Kommandos
 *  zurückliefern. Alternativ den übermittelten Wert der Geschwindigkeit von
 *  der Karte auslesen und auf Übereinstimmung überprüfen. Falls ungleich,
 *  so schlug das Kommando fehl und Rückgabe des Wertes false, ansonsten true.
 *  Nachtrag: vermutlich läßt sich die Geschwindigkeit nicht von der Karte
 *  auslesen, es wurde kein entsprechendes Kommando gefunden.
 */
BOOL TC_842::SetVelocity(DWORD velocity)
{
        char report[80];
        dwVelocity = (velocity < dwMaxVelocity) ? velocity : dwMaxVelocity;
        execute(nOnBoardId,SV,dwVelocity,report);
        return true;
};
```

```
/** Ermitteln der gesetzten Geschwindigkeit für den aktuellen Antrieb
 *  (nOnBoardId).
 *  @return Geschwindigkeit.
 *  @todo Es genügt, wenn nur  beim Setzen der Geschwindigkeit überprüft wird,
 *  ob nicht die maximale Geschwindigkeit überschritten wird.
 */
DWORD TC_842::GetVelocity(void)
{
        dwVelocity = (dwVelocity < dwMaxVelocity) ? dwVelocity : dwMaxVelocity;
        return dwVelocity;
};

/** Einstellen der Beschleunigung.
 *  @param acceleration Wert der Beschleunigung.
 *  @return true.
 *  @todo Rückgabewert ändern und eventuell den Report des execute-Kommandos
 *  zurückliefern. Alternativ den übermittelten Wert der Beschleunigung von
 *  der Karte auslesen und auf Übereinstimmung überprüfen. Falls ungleich, so
 *  schlug das Kommando fehl und Rückgabe des Wertes false, ansonsten true.
 *  Nachtrag: vermutlich läßt sich die Beschleunigung nicht von der Karte
 *  auslesen, es wurde kein entsprechendes Kommando gefunden.
 *  Was passiert, wenn die Beschleunigung zu hoch ist? Bei der Geschwindigkeit
 *  wurde es zuvor überprüft, hier nicht.
 */
BOOL TC_842::SetAcceleration(DWORD acceleration)
{
        char report[80];
        dwAcceleration = acceleration;
        execute(nOnBoardId,SA,dwAcceleration,report);
        return true;
};

/** Ermitteln der gesetzten Beschleunigung für den aktuellen Antrieb
 *  (nOnBoardId).
 *  @return Beschleunigung.
 */
DWORD TC_842::GetAcceleration(void)
{
        return dwAcceleration;
};

/** Setzen des RemoveLimits.
 *  @param limit Wert für RemoveLimit.
 *  @return true.
 */
BOOL TC_842::SetLimit(DWORD limit)
{
        dwRemoveLimit = limit;
        return true;
};

/** Gibt das Statuswort zurück.
 *  @return Statuswort. (unsigned short integer)
 */
WORD TC_842::GetStatus(void)
{
        return (WORD) get_stat(baddr + 1);
};

/** Bestimmen der gegenwärtigen aktuellen Position des gewählten Motors
 *  (nOnBoardId).
 *  @param *position Die Position des Motors als Rückgabe.
```



```
 *  @return true.
 *  @todo Die Position als Rückgabewert zurückliefern und damit den Parameter
 *  einsparen.
 *  Oder den überflüssigen Rückgabewert true entfernen und die Funktion auf
 *  void ändern.
 */
BOOL TC_842::_GetPosition(long *position)
{
        *position = get_pos(nOnBoardId);
        return true;
};

/** Ermittelt die gegenwärtige Anzahl von Fehlercounts, d.h. die Abweichung
 *  der realen Position von der virtuellen Position.
 *  @param *failure Anzahl der Fehlercounts.
 *  @return true.
 *  @todo Anstatt stets true als Rückgabewert zu liefern, könnte die Anzahl der
 *  Fehlercounts zurückgegeben werden, und somit wäre der Parameter *failure
 *  überflüssig.
 */
BOOL TC_842::_GetFailure(long *failure)
{
        // GPE = Get Position Error - richtige Funktionalität?
        *failure = get_posErr(nOnBoardId);
        return true;
};

/** Startet den aktuellen Antrieb (nOnBoardId) und fährt die relative Position
 *  an, ausgehend von der aktuellen Position.
 *  @param position Relative Zielposition.
 *  @return true.
 *  @todo Ändern des Rückgabewerts auf String, damit der Report des Kommandos
 *  zurückgegeben werden kann. Rückgabe von immer true ohne Sinn.
 */
BOOL TC_842::MoveRelative(long position)
{
        char report[80];
        execute(nOnBoardId,MR,position,report);

        return true;
};

/** Startet den aktuellen Antrieb (nOnBoardId) und fährt die absolute Position
 *  an.
 *  @param position Absolute Zielposition.
 *  @return true.
 *  @todo Ändern des Rückgabewerts auf String, damit der Report des Kommandos
 *  zurückgegeben werden kann. Rückgabe von immer true ohne Sinn.
 */
BOOL TC_842::MoveAbsolute(long position)
{
        char report[80];
        execute(nOnBoardId,MA,position,report);
        // example:
        // translate("1ma100000",report);
        // execute(1,MA,100000,report);

        return true;
};

/** Feststellen, ob sich die aktuelle Achse (nOnBoardId) noch in Bewegung
 *  befindet.
```

```
 *  @return true, falls die Achse nicht mehr in Bewegung ist. Ansonsten false.
 */
BOOL TC_842::IsMoveFinish(void)
{
        DelayTime(5);
        if (moving(nOnBoardId))
                return false;
        else
                return true;
};

/** Überprüfen, ob die Karte angeschlossen und verfügbar ist.
 *  @return true.
 *  @todo Entweder keinen Rückgabewert verwenden, da der Status der Karte
 *  in bControlBoardOk gehalten wird, oder Rückgabe des Status der Karte.
 *  In diesem Fall true, wenn eine C-842-Karte in einem PC-Slot installiert
 *  ist, ansonsten false.
 */
BOOL TC_842::CheckBoardOk()
{
        bControlBoardOk = Board_Installed();
        return true;
};

/** Führt das angegebene Kommando als Kommando-String aus. Diese Funktion wird
 *  bisher nicht verwendet. Die DLL-Bibliothek stellt aber eine ähnliche
 *  Funktion zur Verfügung, die damit angesprochen werden könnte: translate.
 *  Diese ist aber nicht zu verwechseln mit der Funktion execute aus der
 *  DLL-Bibliothek. Ein beispielhafter Aufruf könnte lauten:
 *  translate("1MA10000", report);
 *  oder mit Hilfe dieser Funktion:
 *  ExecuteCmd("1MA10000");
 *  @param command Der Kommando-String in der Form aCn, wobei a den Motor
 *  identifiziert (a = 0, 1, 2). Bei a = 0 werden beide Motoren gleichzeitig
 *  angesprochen. C ist das Kommando (siehe verfügbare Kommandos, z.B. MA für
 *  MoveAbsolute). n gibt den Wert des Parameters für das entsprechende
 *  Kommando an.
 *  @return true.
 *  @todo Den Rückgabewert ändern, so dass der Report des ausgeführten
 *  Kommandos als String zurück geliefert wird.
 */
BOOL TC_842::ExecuteCmd(LPSTR command)
{
        char report[80];
        translate(command,report);

        return true;
};
```

```
/** Erzeugen des Optimierungsdialoges als Modal-Dialog.
 */
TOptimizeDC_842Dlg::TOptimizeDC_842Dlg( void ) :
        TModalDlg( "OptimizeDC_842", hModuleInstance )
{
        Drive = (TC_842*)lpMList->MP( );
        bCancel = TRUE;
};

/** Initialisieren des Optimierungsdialoges. Zwischenspeichern der alten
 *  Parameterwerte.
 *  @param hwnd
 *  @param hwndCtl
 *  @param u not used.
 *  @return true.
 */
BOOL TOptimizeDC_842Dlg::Dlg_OnInit( HWND hwnd, HWND hwndCtl, LPARAM u )
{
        old_vel = Drive->dwMaxVelocity;
        old_accel = Drive->dwAcceleration;
        old_kp = Drive->wKP;
        old_kd = Drive->wKD;
        old_ki = Drive->wKI;
        old_kl = Drive->wKL;
        old_poswidth = Drive->wPositionWidth;

        FORWARD_WM_COMMAND( hwnd, cm_ParamSet, hwndCtl, 0, SendMessage );

        return true;
};

/** ï¿œernehmen der neu eingestellten Parameter.
 *  @param hwnd
 *  @param id
 *  @param hwndCtl
 *  @param codeNotify
 */
void TOptimizeDC_842Dlg::Dlg_OnCommand( HWND hwnd, int id, HWND hwndCtl,
        UINT codeNotify )
{
        char buf[ MaxString ];

        switch ( id )
        {
                case cm_ParamSet:
                // Testgruppe
                sprintf( buf, szMsgLine011, "C842", Drive->pCharacteristic( ) );
                SetWindowText( hwnd, buf );
                SetWindowText( GetHandle( ), buf );
                sprintf( buf, "%ld", Drive->dwMaxVelocity );
                SetDlgItemText( GetHandle( ), id_Velocity, ( LPSTR ) buf );
                sprintf( buf, "%ld", Drive->dwAcceleration );
                SetDlgItemText( GetHandle( ), id_Acceleration, ( LPSTR ) buf );
                sprintf( buf, "%d", Drive->wKD );
                SetDlgItemText( GetHandle( ), id_kd, ( LPSTR ) buf );
                sprintf( buf, "%d", Drive->wKP );
                SetDlgItemText( GetHandle( ), id_kp, ( LPSTR ) buf );
                sprintf( buf, "%d", Drive->wKI );
                SetDlgItemText( GetHandle( ), id_ki, ( LPSTR ) buf );
                sprintf( buf, "%d", Drive->wKL );
                SetDlgItemText( GetHandle( ), id_kl, ( LPSTR ) buf );
                sprintf( buf, "%d", Drive->wPositionWidth );
```



```
                SetDlgItemText( GetHandle( ), id_PositionWidth, ( LPSTR ) buf );
                break;

                case cm_StartMoveScan:
                GetDlgItemText( GetHandle( ), id_Velocity, buf, MaxString );
                Drive->dwMaxVelocity = atol( buf );
                GetDlgItemText( GetHandle( ), id_Acceleration, buf, MaxString );
                Drive->dwAcceleration = atol( buf );
                GetDlgItemText( GetHandle( ), id_kd, buf, MaxString );
                Drive->wKD = ( WORD ) atoi( buf );
                GetDlgItemText( GetHandle( ), id_kp, buf, MaxString );
                Drive->wKP = ( WORD ) atoi( buf );
                GetDlgItemText( GetHandle( ), id_ki, buf, MaxString );
                Drive->wKI = ( WORD ) atoi( buf );
                GetDlgItemText( GetHandle( ), id_kl, buf, MaxString );
                Drive->wKL = ( WORD ) atoi( buf );
                GetDlgItemText( GetHandle( ), id_PositionWidth, buf, MaxString);
                Drive->wPositionWidth = ( WORD ) atoi( buf );
                Drive->ActivateFilterParameters( );
                mStartMoveScan( 10, 0 );
                return ;

                default:
                TModalDlg::Dlg_OnCommand( hwnd, id, hwndCtl, codeNotify );
        }
        return ;
};

/** Setzen von bCancel auf false. Dieser Wert gibt an, dass beim Schliessen
 *  des Dialogs die neuen Werte beibehalten werden.
 *  @return true
 *  @see TOptimizeDC_842Dlg::LeaveDialog(void)
 */
BOOL TOptimizeDC_842Dlg::CanClose( void )
{
        bCancel = false;
        Drive->SaveSettings(0);
        return true;
};

/** ï¿œernimmt beim Verlassen des Dialoges die alten Werte, falls
 *  bCancel = true.
 *  @see TOptimizeDC_842Dlg::CanClose(void)
 */
void TOptimizeDC_842Dlg::LeaveDialog( void )
{
        if ( bCancel )
        {
                Drive->dwMaxVelocity = old_vel;
                Drive->dwAcceleration = old_accel;
                Drive->wKP = old_kp;
                Drive->wKD = old_kd;
                Drive->wKI = old_ki;
                Drive->wKL = old_kl;
                Drive->wPositionWidth = old_poswidth;
        }
        Drive->ActivateFilterParameters( );
};
```

Anhang F

Literatur

1. PhysikInstrumente, Operating Manual MS 45 E, C-842 DC-Motor Controllers, August 1996
2. PhysikInstrumente, Software Application Manual SM 130 E, 842 Tools Version 4.0, Dezember 1996
3. PhysikInstrumente, Operating Manual MP 34 E, Precision Rotation Stages, April 1999
4. Harder/Paschold, Portierungsstrategie für ein Hardwaresteuerungsprogramm unter Anwendung von Reverse Engineering Techniken, August 2003
5. Projektwebseite, http://www2.informatik.hu-berlin.de/swt/projekt98/