RE-ENGINEERING DER OBJEKTORIENTIERTEN TEILE DES SUBSYSTEMS MOTORSTEUERUNG

Dokument zur Diplomarbeit
- Designphase -

Autoren	Thomas Kullmann, Günther Reinecker
Dokumentversion	1.1
Zustand	abgeschlossen
letzte Bearbeitung	15.12.02

Inhalt

I Überblick

Dieses Dokument ist als Resultat eines Reengineeringprozesses an den Klassen zur Motorsteuerung aufzufassen. Motivation für die Überarbeitung war das eliminieren von totem Code, das Erkennen der Ursachen schlechter Metriken und damit die Verbesserung des Designs mittels Refactoring. Ein besonderer Punkt war hierbei die Erhöhung der Datenkapselung.

Neue Member sind durch **`NEU`** und geänderte Signaturen sind durch **`GEÄ`** in der letzten Spalte (nach dem Zugriffsschutz, z.B. **`PRIVATE`**) markiert. Der Fährnis halber wurde, zur Bestimmung der Metriken, der auskommentierte, tote Code entfernt, um die Ergebnisse nicht zu verfälschen.

Des Weiteren stand bei der Erarbeitung dieses Dokuments die Vollständigkeit im Mittelpunkt. Jeder Typ, jede Klasse, jeder Member, u.ä. wird kurz und präzise erläutert. Dazu wurde dieses Dokument stark formalisiert, die Layoutkonventionen sind unter **[6]** zu finden. Wo immer es möglich war, wurden Quellenangaben hinzugefügt, um ggf. weiterführende Informationen zum Thema zu erhalten. Um das Dokument übersichtlicher zu gestalten, wurden die behandelten Attribute und Methode nach "Sinneinheiten" geordnet.

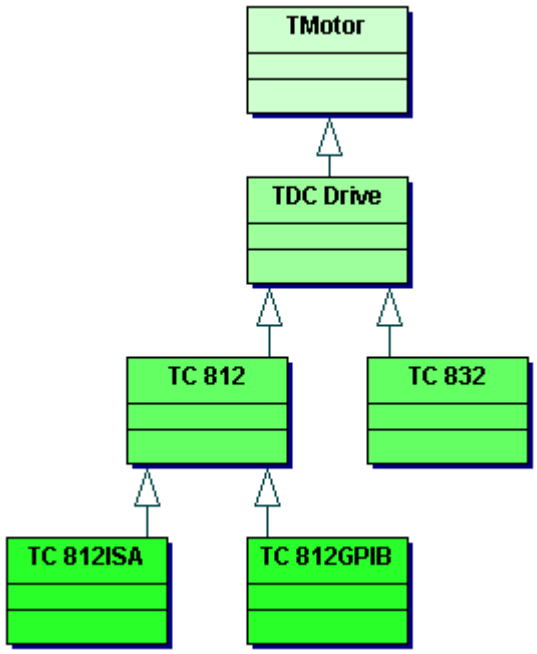
► Dokumentation des Istzustands

Die Dokumentation bezieht sich auf den Quellcode der hier aufgelisteten Dateien (Stand: 08.08.2002). Nachfolgende Änderungen oder Neuimplementierungen können nicht berücksichtigt werden!

Die an den Laborplätzen verwendeten Antriebe besitzen grundsätzlich dieselbe Funktionalität. Sie sind über eine Controller-Karte mit den PC's verbunden, wobei pro Karte mehrere Antriebe an den PC angeschlossen werden.

Derzeit werden die beiden Controller-Karten C812 und C832 eingesetzt[1], deren Ansteuerung sich an vielen Teilen unterscheidet – die Klassen `TC_812` und `TC_832` implementieren diese Ansteuerung. Beide wurden von `TDC_Drive` abgeleitet, weil hier auf die Besonderheiten von real existierenden Antrieben (wie z.B. *Antriebsspiel*) eingegangen wird. Die Klasse `TMotor` bildet mit der Funktionalität die alle Antriebe bieten, das Grundgerüst für alle Motor-Klassen.

[1] Typ 844 wird gegenwärtig durch Andreas Wenzel realisiert



Klasse	Deklaration	Implementation
TMotor	M_MOTCOM.H	MOTORS.CPP
TDCDrive	M_MOTHW.H	MOTORS.CPP
TC_832	M_MOTHW.H	MOTORS.CPP M_LAYER.CPP
TC_812	M_MOTHW.H	MOTORS.CPP
TC_812ISA	M_MOTHW.H	MOTORS.CPP M_LAYER.CPP
TC_812GPIB	M_MOTHW.H	MOTORS.CPP M_LAYER.CPP

Abbildung 1 „UML-Klassendiagramm: Vererbungshierarchie der Antriebe“ (Quelle: Together® ,Version 6.0)

Jeder Antrieb wird durch einen [MOTOR<M>]-Abschnitt in der HARDWARE.INI repräsentiert, wobei <M> für den Index des Antriebs steht. Alle zum Subsystem Motorsteuerung gehörigen Dateien (siehe **Tabelle 1**) werden in einer Dynamic Link Library zusammengelinkt. Während der Initialisierungsphase dieser MOTORS.DLL, wird eine Liste mit den verfügbaren Antrieben erstellt (siehe **IV.2 Klasse TMList**). Je nach Typ der verwendeten Controller-Karte (ermittelt über den ini-Eintrag Type), wird ein entsprechendes Objekt (entweder der Klasse `TC_812ISA` oder `TC_812GPIB` oder `TC_832`) in die Antriebsliste aufgenommen. Die Liste ist eine Komposition von `TMotor`-Objekten, weil diese Klasse (wie bereits erwähnt) das Grundgerüst für alle Motor-Klassen bildet. Für die Benutzung der objektorientierten Motorsteuerung sind daher keine genauen Kenntnisse über die verwendeten Antriebe oder deren Controller-Karten erforderlich. Es genügt stets, Objekte der Klasse `TMotor` zu verwenden, weshalb hier nur auf diese Klasse eingegangen werden soll.

Detaillierte Informationen zur Ansteuerung der Hardware, wie sie die übrigen Klassen implementieren, werden hier nicht behandelt – näheres hierzu findet man unter „Motorenansteuerung“ – Vortrag vom 09./16./30. Juni 1999 von Derrick Hepp und Sebastian Freund.

h-Dateien (Deklaration)	cpp-Dateien (Implementation)
• M_MOTCOM.H	• MOTORS.CPP
• M_MOTHW.H	
• MSIMSTAT.H	• MSIMSTAT.CPP
• M_LAYER.H	• M_LAYER.CPP
• MOTRSTRG.H	
• IEEE.H	
• C8X2.H	

Tabelle 1 „Auflistung der zum Subsystem zugehörigen Dateien“ (Quelle: **[1]**)

II Typen

► **`typedef enum { … } TAxisType`**

eindeutige Achse eines Antriebs – im Gegensatz zur "beliebigen", texturellen Bezeichnung, wie sie in den meisten Dialogfenstern angezeigt werden

"beliebige" Bezeichnung	TAxisType
"AzimutalRotation"	Rotation
"AZ"	
"Rotation"	
"Azimute"	
"X"	X
"x"	
"Horizontal"	
"x-Achse"	
"x-Axis"	
"Y"	Y
"y"	
"y-Achse"	
"y-Axis"	
"Z"	Z
"z"	
"Vertical"	
"z-Achse"	
"z-Axis"	

"beliebige" Bezeichnung	TAxisType
"Theta"	Theta
"Omega"	Omega
"Tilt"	Tilt
"DC"	DC
"Diffraction coarse"	
"Beugung grob"	
"DF"	DF
"Diffraction fine"	
"Beugung fein"	
"Psi"	Psi
"Phi"	Phi
"CC"	CC
"Collimator"	
"Kollimator"	
"Absorber"	Absorber
"Monochromator"	Monochromator
"Encoder"	Encoder

Tabelle 2 „Achse: Beziehungen zwischen Bezeichnung und eindeutigem Aufzählungstyp – Groß-/ Kleinschreibung ist zu beachten!" (Quelle: selbst)

Die Methode `TMList::ParsingAxis` ist für die Abbildung von *Bezeichnung* auf `TAxisType` zuständig. Eine Umkehrfunktion ist nicht erforderlich und deshalb nicht vorhanden.

► **`typedef enum { … } TUnitType`**

eindeutige Einheit eines Antriebs – im Gegensatz zur "beliebigen", texturellen Bezeichnung

"beliebige" Bezeichnung	TUnitType
"GRAD"	Grad
"SEKUNDEN"	Sekunden
"MINUTEN"	Minuten
"MINUTS"	
"MILLIMETER"	Millimeter
"MIKROMETER"	Mikrometer
"MIKCOMETER"	
"CHANNEL"	Channel

Tabelle 3 „Einheit: Beziehungen zwischen der Bezeichnung und dem eindeutigen Aufzählungstyp – Groß-/ Kleinschreibung ist zu beachten!" (Quelle: selbst)

Diese beiden Methoden sind für die Umrechnungen zuständig:

- `TUnitType UnitEnum( LPSTR )`
- `LPSTR UnitStr( TUnitType )`

Die fett hervorgehobenen Bezeichnungen sind für die eindeutige Abbildung von `TUnitType` in *Bezeichnung* – in Methode `UnitStr` – erforderlich!

► **`typedef enum { … } TCorrect`**
bestimmt die Art der Korrektur, die bei der Umrechnung zwischen:

- **interne Antriebspositionen** in **Encoderschritten** (die intern zur direkten Antriebssteuerung verwendet werden) und
- **Absolutpositionen** in **Nutzereinheiten** (zum Anzeigen in den meisten Dialogfenstern)

beachtet werden muss; Details zur Umrechnung sind bei `TMotor::dKoeff_1` zu finden.

Zusammenhang zwischen interner Antriebsposition und Absolutposition	`TCorrect`
linear	`CorrLinear`
nicht-linear (Annäherung über Polynom)	`CorrPolynom`

Tabelle 4 „symbolische Werte von `TCorrect`“ (Quelle: selbst)

► **`typedef enum TSimulationType { … } TCorrect`**
Art der Antriebssimulation:

- `no_simulation` → keine Simulation, es wird nur die Hardware verwendet
- `simulation_only` → keine Hardwareansteuerung, nur Simulation
- `test_simulation` → Ergebnis der Simulation mit Rückgabewert der Hardware vergleichen

III Konstanten

► **`#define R_OK`**
Rückgabewert bei mehreren Methoden, signalisiert Erfolg

► **`#define R_Failure`**
Rückgabewert bei mehreren Methoden, signalisiert Misserfolg

► **`const _extern UINT _MAXLENCHARACTERISTIC= 50`**
maximale Länge von `TMotor::szCharacteristic`

► **`const _extern UINT _MAXLENUNIT= 20`**
maximale Länge von `TMotor::szUnit`

► **`const _extern UINT _MAXLENFORMAT= 5`**
maximale Länge von `TMotor::DFmt` und `TMotor::SFmt`

IV Klassen

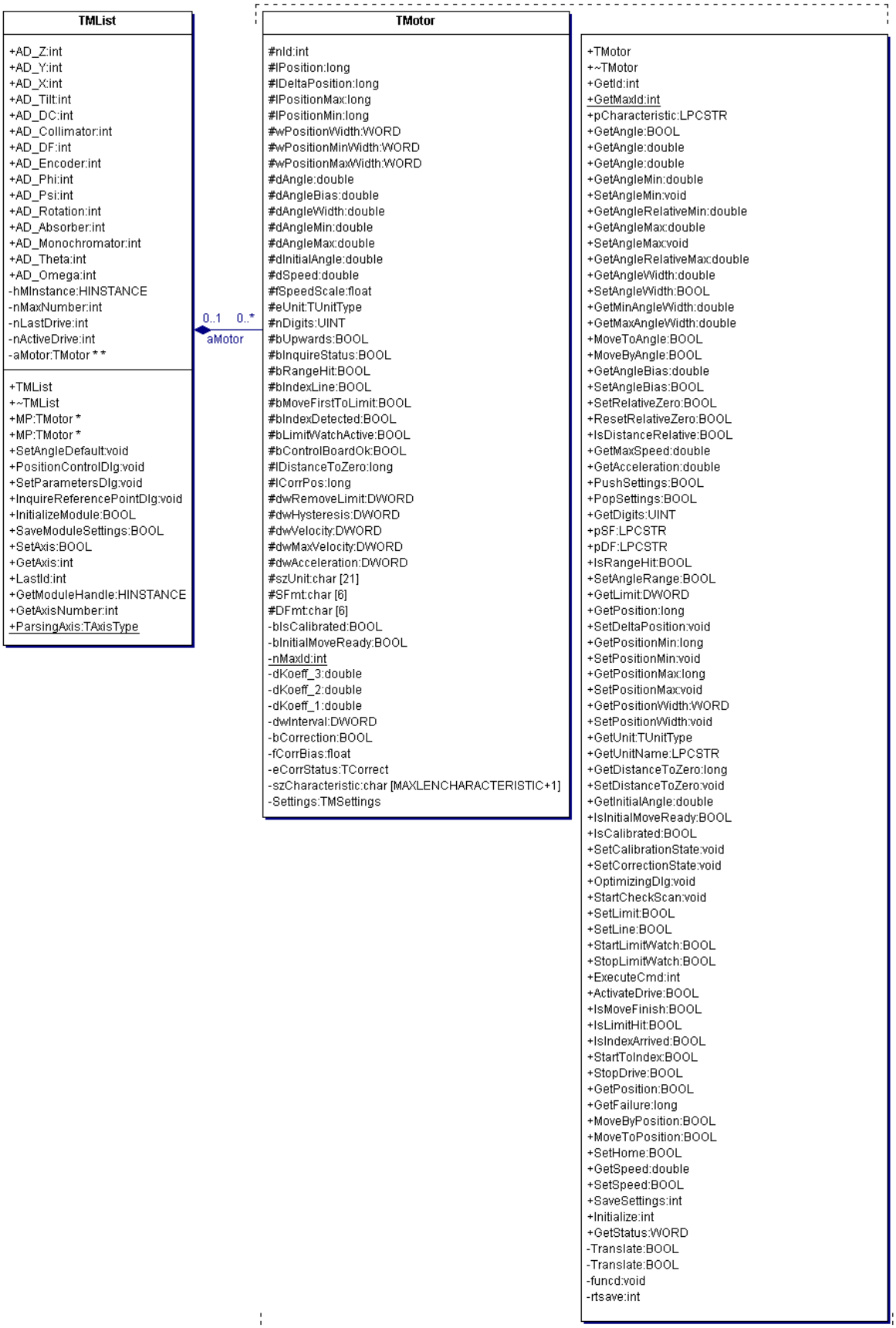


Abbildung 2 „UML-Klassendiagramm: `TMotor` und `TMList`“ (Quelle: Together® ,Version 6.0)

IV.1 Klasse TMotor

Deklaration : M_MOTCOM.H
Implementation : MOTORS.CPP
Es gibt keine abstrakten Methoden, d.h. Objekte von `TMotor` können erzeugt werden.

IV.1.1 Attribute

▶ **`char szCharacteristic[ _MAXLENCHARACTERISTIC+1 ]`** **PRIVATE**
"Beliebige" texturelle Bezeichnung (z.B. „Kollimator") des Antriebs, so wie diese in Dialogfenstern anzuzeigen ist; Für die korrekte Ermittlung der Wirkachse ist es erforderlich, dass `szCharacteristic` einen der Werte aus **Tabelle 2** annimmt.
HARDWARE.INI –> [MOTOR<M>] –> Name

▶ **`long lPosition`** **PROTECTED**
aktuelle, *interne Antriebsposition* (in **Encoderschritten**)
HARDWARE.INI –> [MOTOR<M>] –> DeltaPosition
wird hier nur geschrieben und bei `lDeltaPosition` nur gelesen

▶ **`long lPositionMin`** **PROTECTED**
definiert die untere Schranke (in **Encoderschritten**) für die aktuelle, interne Ist- und Sollposition des Antriebs; siehe: **[1]**
HARDWARE.INI –> [MOTOR<M>] –> PositionMin

▶ **`long lPositionMax`** **PROTECTED**
wie `lPositionMin`, definiert jedoch die obere Schranke; siehe: **[1]**
HARDWARE.INI –> [MOTOR<M>] –> PositionMax

▶ **`WORD wPositionWidth`** **PROTECTED**
aktuelle Schrittweite (in **Encoderschritten**) für die Bewegung des Antriebs; siehe: **[1]**
HARDWARE.INI –> [MOTOR<M>] –> PositionWidth
Inkonsistente Mehrfachverwendung! (siehe **[1]**: bei F4 anders als bei F6 verwendet)

▶ **`WORD wPositionMinWidth`** **PROTECTED**
untere Schranke (in **Encoderschritten**) für `wPositionWidth`
HARDWARE.INI –> [MOTOR<M>] –> MinimalWidth

▶ **`WORD wPositionMaxWidth`** **PROTECTED**
wie `wPositionMinWidth`, definiert jedoch die obere Schranke
HARDWARE.INI –> [MOTOR<M>] –> MaximalWidth

▶ **`char szUnit[ _MAXLENUNIT+1 ]`** **PROTECTED**
"Beliebige", texturelle Bezeichnung der Einheit, so wie diese in Dialogfenstern anzuzeigen ist; Für die korrekte Initialisierung von `eUnit` ist es erforderlich, dass `szUnit` einen der Werte aus **Tabelle 3** annimmt!
HARDWARE.INI –> [MOTOR<M>] –> Unit

► **TUnitType eUnit** **PROTECTED**

Nutzereinheit – Einheit eines Antriebs, die Positionsangaben in Dialogfenstern sind i.d.R. in Nutzereinheiten; siehe `szUnit` und [1]

► **double dAngle**[2] **PROTECTED**

aktuelle *Absolutposition* (d.h. wie `lPosition`), jedoch in **Nutzereinheiten**
bedingt redundant, da diese Werte jederzeit aus den Angaben in Encoderschritten berechnet werden können; mögliche Fehlerquelle, da bei Schreibzugriffen stets beide Angaben zu berücksichtigen sind

► **double dAngleMin** **PROTECTED**

wie `lPositionMin`, jedoch in **Nutzereinheiten**; siehe: [1]
HARDWARE.INI –> [MOTOR<M>] –> AngleMin

► **double dAngleMax** **PROTECTED**

wie `lPositionMax`, jedoch in **Nutzereinheiten**; siehe: [1]
HARDWARE.INI –> [MOTOR<M>] –> AngleMax

► **double dAngleWidth** **PROTECTED**

wie `wPositionWidth`, jedoch in **Nutzereinheiten**; siehe: [1]
HARDWARE.INI –> [MOTOR<M>] –> AngleWidth

► **long lCorrPos** **PROTECTED**

ist immer bei der Umrechnung zwischen der hardwaretechnischen <A> und softwareseitigen <B>, *internen Antriebsposition* (beide in **Encoderschritten**) zu beachten; stets gilt:
`<B> := <A> + lCorrPos - lDeltaPosition`

Nur bei `eCorrStatus == CorrPolynom` ist, kann `lCorrPos` von `0` verschiedene Werte annehmen.

► **long lDeltaPosition** **PROTECTED**

ähnlich `lCorrPos`; Koordinatenverschiebung in **Encoderschritten**; siehe: `lCorrPos` und [1]
HARDWARE.INI –> [MOTOR<M>] –> DeltaPosition
wird hier nur gelesen und bei `lPosition` nur geschrieben

► **float fCorrBias** **PRIVATE**

Unabhängig von `eCorrStatus`, muss diese Verschiebung (in **Nutzereinheiten**), bei jeder Umrechnung zwischen *internen Antriebspositionen* <A> und *Absolutpositionen* <B> beachtet werden. Es gilt: `<B> := <A> + dAngleBias - fCorrBias`

► **double dAngleBias** **PROTECTED**

ähnlich `fCorrBias` ist dieses *Offset*[3] (in **Nutzereinheiten**) stets zu beachten; siehe `fCorrBias` und [1]
HARDWARE.INI –> [MOTOR<M>] –> AngleBias

---

[2] bedingt redundant, da diese Werte jederzeit aus den Angaben in Encoderschritten berechnet werden können; mögliche Fehlerquelle, da bei Schreibzugriffen stets beide Angaben zu berücksichtigen sind

[3] *Offset für <Antrieb>* ODER *Relative Null*, d.h. aktuelle Absolutposition des Antriebs, wenn zuvor kein *Offset* definiert ist

► **BOOL bCorrection** **PRIVATE**

bestimmt, wie `eCorrStatus` in Methode `SetCorrectionState` gesetzt wird:

- `TRUE` → Methode arbeit „normal“: je nach Parameter sind `CorrLinear` und `CorrPolynom` möglich
- `FALSE` → nur `eCorrStatus == CorrLinear` möglich, egal mit welchem Parameter die Methode aufgerufen wird, entspricht immer `SetCorrectionState( FALSE )`

HARDWARE.INI –> [MOTOR<M>] –> Correction

► **TCorrect eCorrStatus** **PRIVATE**

Welcher Zusammenhang herrscht bei der Umrechnung zwischen *internen Antriebspositionen* `<A>` und *Absolutpositionen* `<B>` bei diesem Antrieb? Wert kann nur bei Methode `SetCorrectionState` gesetzt werden; siehe `TCorrect`, `bKoeff_1` und Methode `SetCorrectionState`

► **double dKoeff_1** **PRIVATE**

abhängig von `eCorrStatus`; siehe: [1]

- `CorrLinear` → Faktor zur linearen Umrechnung; es gilt: `<B> := <A> · dKoeff_1`
- `CorrPolynom` → erster Koeffizient im (zur Umrechnung verwendeten) Polynom:
  `<B> := <A> · ( dKoeff_1 + <A> · (dKoeff_2 + <A> · dKoeff_3) )`

HARDWARE.INI –> [MOTOR<M>] –> Koeff_1

► **double dKoeff_2** **PRIVATE**

wird nur bei `eCorrStatus == CorrPolynom` (als zweiter Koeffizient im Polynom) verwendet; siehe: [1]

HARDWARE.INI –> [MOTOR<M>] –> Koeff_2

► **double dKoeff_3** **PRIVATE**

wie `bKoeff_2`, jedoch als dritten Koeffizienten; siehe: [1]

HARDWARE.INI –> [MOTOR<M>] –> Koeff_3

► **DWORD dwInterval** **PRIVATE**

wird nur bei `eCorrStatus == CorrPolynom` als Intervallgrenze beim *Newtonverfahren* (siehe Methode `rtsave`) verwendet; Die Initialisierung von `dwIntervall` hängt von `eUnit` ab:

- `Grad` → `dwIntervall := 3 ·(<MaxFailure> / dKoeff_1) · 3600`
- `Minuten` → `dwIntervall := 3 ·(<MaxFailure> / dKoeff_1) · 60`
- sonst → `dwIntervall := 3 ·(<MaxFailure> / dKoeff_1)`

(wobei `<MaxFailure>`: HARDWARE.INI –> [MOTOR<M>] –> MaxFailure)

► **DWORD dwAcceleration** **PROTECTED**

Beschleunigung bzw. Verzögerung (in **Encoderschritten/ Sekunde²**); nur Werte größer `0` sind sinnvoll; siehe: [1]

HARDWARE.INI –> [MOTOR<M>] –> Acceleration

► **DWORD dwVelocity** **PROTECTED**

Bewegungsgeschwindigkeit des Antriebs in **Encoderschritten/ Sekunde**; Minimum `1`; siehe: [1]

HARDWARE.INI –> [MOTOR<M>] –> Velocity

► **DWORD dwMaxVelocity** **PROTECTED**
Maximalgeschwindigkeit des Antriebs in **Encoderschritten/ Sekunde**; siehe: [1]
HARDWARE.INI –> [MOTOR<M>] –> MaxVelocity

► **float fSpeedScale** **PROTECTED**
für die Umrechnung zwischen `dSpeed` und `dwVelocity` gilt stets:
`dwVelocity := dSpeed · fSpeedScale` (siehe: [1])
HARDWARE.INI –> [MOTOR<M>] –> SpeedScale

► **double dSpeed** **PROTECTED**
wie `dwVelocity`, jedoch in **Nutzereinheiten/ Sekunde**; siehe: [2]

► **DWORD dwHysteresis** **PROTECTED**
Antriebsspiel (in **Encoderschritten**) ist systematischer Anteil am Positionierungsfehler der Antriebe, entsteht durch fertigungsbedingte Toleranzen bei den Getrieben – wird durch die Software korrigiert; siehe: [1]
HARDWARE.INI –> [MOTOR<M>] –> Hysteresis

► **BOOL bUpwards** **PROTECTED**
`TRUE` ↔ Die letzte Antriebsbewegung war Vorwärts gerichtet. wichtig um *Antriebsspiel* (`dwHysteresis`) beim Richtungswechsel (d.h. `bUpwards` hat sich geändert) zu berücksichten; siehe: [1]
HARDWARE.INI –> [MOTOR<M>] –> Upwards

► **BOOL bInquireStatus** **PROTECTED**
Wie kann der Stillstand des Antriebs ermittelt werden? (Quelle: [1])
- `TRUE` → über das Statusbit des Controllers
- `FALSE` → der folgenden Test zeigt an, dass der Antrieb steht (Einheiten in **Encoderschritten**):
  `| <Sollposition> - <Istposition> | ≤ <DeathBand>`
  (wobei `<Deathband>`: HARDWARE.INI –> [MOTOR<M>] –> Deathband)

HARDWARE.INI –> [MOTOR<M>] –> InquireStatus

► **BOOL bRangeHit** **PROTECTED**
`TRUE` ↔ Die zuletzt mit Methode `Translate` übersetzte *Antriebsposition* verletzte den Wertebereich und wurde auf den Minimal- oder Maximalwert korrigiert (je nachdem ob der Wertebereich unter- oder überschritten wurde).

► **DWORD dwRemoveLimit** **PROTECTED**
Distanz (in **Encoderschritten**) die der Antrieb zurückgefahren werden soll, wenn die Hardwareschranke erreicht wird (bei C812 hardware-, bei C832 softwaregesteuert) – benötigt zum Entspannen der Endlagenschalter; Quelle: [1]
HARDWARE.INI –> [MOTOR<M>] –> RemoveLimit

► **BOOL bIndexLine** **PROTECTED**
kennzeichnet, ob der Antrieb einen **Index-Schalter** besitzt; Wenn ja, dann besitzt er eine als **Referenzpunkt** ausgewiesene Position. Diese ist eine hardwareseitig erkennbare *interne Antriebsposition*, die reproduzierbar angefahren werden kann. Man benötigt sie, um zwischen *internen Antriebspositionen* und *Absolutpositionen* umrechnen, siehe `lDistanceToZero`. Quelle: [1]
HARDWARE.INI –> [MOTOR<M>] –> IndexLine

► **BOOL bInitialMoveReady** **PRIVATE**

Referenzpunktlauf (Verfahren zum Anfahren des *Referenzpunkts*) zulässig? (Quelle: [1])

- `TRUE` → kann über das Dialogfenster ‚Grundstellung anfahren', Schaltfläche ‚Referenzpunktlauf' (siehe Methode `StartToIndex`) durchgeführt werden; siehe [7]
- `FALSE` → Referenzpunktlauf ist nicht zulässig – erforderlich, wenn keine Hardwareschranken definiert oder kein Index-Schalter vorhanden ist

HARDWARE.INI –> [MOTOR<M>] –> InitialMove

► **BOOL bMoveFirstToLimit** **PROTECTED**

Richtung, in der der Antrieb seinen *Referenzpunkt* anfahren soll; Voraussetzung: `bInitialMoveReady == TRUE` (Quelle: [1])

- `TRUE` → Antrieb fährt zuerst rückwärts bis zur linken, minimalen Hardwareschranke und anschließend vorwärts, bis der *Index-Schalter* erreicht wird (Referenzpunkt von links angefahren)
- `FALSE` → Antrieb fährt rückwärts zur linken, minimalen Hardwareschranke und stoppt sobald der *Index-Schalter* oder die Hardwareschranke erreicht werden (von rechts angefahren)

HARDWARE.INI –> [MOTOR<M>] –> MoveFirstToLimit

► **BOOL bIndexDetected** **PROTECTED**

kennzeichnet, dass der *Referenzpunkt* erreicht wurde; Antrieb kann mittlerweile an einer anderen Stelle stehen; am *Referenzpunkt* gilt stets: `lDeltaPosition := -lDistanceToZero`

► **long lDistanceToZero** **PROTECTED**

wird während des *Referenzpunktlaufes* bestimmt; entspricht dem Abstand (in **Encoderschritten**) zwischen *Referenzpunkt* und absoluter Null; berechnet sich wie folgt:

`lDistanceToZero := <Referenzpunkt> - <absolute Null>` (Quelle: [1])

HARDWARE.INI –> [MOTOR<M>] –> DistanceToZero

► **double dInitialAngle** **PROTECTED**

Absolutposition, die nach dem Erreichen des *Referenzpunktes* angefahren werden soll (in **Nutzereinheiten**); Quelle: [1]

HARDWARE.INI –> [MOTOR<M>] –> InitialAngle

► **BOOL bIsCalibrated** **PRIVATE**

gibt an, ob der *Referenzpunktlauf* erfolgreich durchgeführt wurde; Quelle: [3]

► **BOOL bControlBoardOk** **PROTECTED**

Arbeitet der Controller fehlerfrei? kann nur bei Methode `CheckBoardOk` auf `TRUE` gesetzt werden

► **BOOL bLimitWatchActive** **PROTECTED**

gibt an, ob *Limit Watch* aktiv ist; siehe `StartLimitWatch`

► **char DFmt[_MAXLENFORMAT+1]** **PROTECTED**

Zahlenformat, wie es von Methode `printf` verwendet wird, Standard ist „`%.2lf`", wobei `2` durch die gewünschte Nachkommastellengenauigkeit (`<Digits>`) ersetzt wird

(wobei `<Digits>`: HARDWARE.INI –> [MOTOR<M>] –> Digits)

► **char SFmt[_MAXLENFORMAT+1]** **PROTECTED**
wie `DFmt`, die `2` wird jedoch durch `<Digits> + 1` ersetzt

► **TMSettings Settings** **PRIVATE**
zum Zwischenspeichern[4] des aktuellen Inhalts der folgenden Attribute: `dAngle` und `dAngleMin` und `dAngleMax` und `dAngleWidth` und `dSpeed`
Diese Werte können später mit Methode `PopSettings` wiederhergestellt werden, dabei wird auch `dAngle` wieder angefahren.

► **Int nId** **PROTECTED**
dient zur Zuordnung der Antriebe in `TMList`, entspricht dem Index des Antriebs; stets im Intervall [`0` ... `nMaxId-1`]

► **static int nMaxId** **PRIVATE**
die Anzahl der erzeugten Antriebe, beginnend bei `0`

► **int nDigits** **PROTECTED**
die Anzahl der Nachkommastellen für alle reellwertigen Zahlenangaben innerhalb eines Antriebs; HARDWARE.INI –> [MOTOR<M>] –> Digits

IV.1.2 Methoden

betrachtet werden alle in `TMotor` deklarierten Methoden, wo möglich, wird auch auf das Verhalten in den abgeleiteten Klassen hingewiesen, Da die Dokumentation des dazugehörigen C-Interfaces ([3]) sehr gut ist, wird wenn möglich darauf verwiesen.

► **TMotor(void)** **PUBLIC**
Der Konstruktor initialisiert einen Großteil, der unter **IV.1.1 Attribute** genannten Attribute und registriert den Antrieb bei Objekt `lpMList` (siehe **IV.2 Klasse TMList**). In Methode `Initialize` werden weitere Attribute initialisiert.
wird in allen abgeleiteten Klassen überschrieben und ergänzt; Die meisten Werte stammen dabei aus der HARDWARE.INI, Abschnitt [MOTOR<M>].

► **virtual int Initialize(void)** **PUBLIC**
initialisiert viele Attribute und überschreibt einige bereits im Konstruktor initialisierte Werte mit Werten aus der HARDWARE.INI; Rückgabewert stets `R_OK`
überschrieben und erweitert in `TDC_Drive` und `TC_812`; hier auch andere Rückgabewerte
Weder hier noch im Konstruktor werden `dSpeed` und `Settings` initialisiert!

► **virtual ~TMotor(void)** **PUBLIC**
Da kein Attribut dynamisch erzeugt wird, ist der Destruktor (auch in allen abgeleiteten Klassen) leer.

[4] mit Methode `PushSettings`

► **`virtual int SaveSettings( BOOL )`** **PUBLIC**

speichert einen Teil der Attribute in der HARDWARE.INI; Wenn `FALSE` als Parameter übergeben wird, werden weniger Attribute gespeichert[5]. Bei `TRUE` handelt es sich um den Speichervorgang direkt vor der Programmbeendigung: Wenn ein gültiger Referenzpunktlauf vorliegt, wird der ini-Eintrag RestartPossible `TRUE` gesetzt. Beim Neustart des Programms können die Angaben zur Antriebsposition demnach problemlos aus der ini-Datei entnommen werden.

Rückgabewert: `R_Failure` ↔ MOTORS.DLL wurde nicht erfolgreich initialisiert wurde (sonst `R_OK`)

wird bei `TC_812` und `TC_832` überschrieben, um zusätzliche Attribute in der ini-Datei abzulegen;

Hier wird nicht geprüft, ob MOTORS.DLL erfolgreich initialisiert wurde.

► **`virtual BOOL GetPosition( BOOL )`** **PUBLIC**

hier nur Rückgabe von TRUE

überschrieben bei `TC_Drive`, hier wird die *interne Antriebsposition* ermittelt und (bei Erfolg) in `lPosition` gespeichert; Der Parameter gibt an, ob die Position ein zweites Mal ermittelt und mit der ersten Position verglichen werden soll. Rückgabewert: `TRUE` ↔ Position[en] erfolgreich ermittelt [die beiden Positionen waren identisch]

► **`BOOL GetAngle( BOOL )`** **PUBLIC**

ermittelt die aktuelle, *interne Antriebsposition* (speichert diese in `lPosition`) und berechnet daraus die *Absolutposition* (speichert diese in `dAngle`); Nur wenn der Parameter `TRUE` ist und `CheckCorrect` definiert ist, werden Statusinformationen im Hauptfenster ‚Steuerprogramm' ausgegeben;
siehe [3]: `mlGetDistance`

Rückgabewert: `TRUE` ↔ Ermittlung und Übersetzung erfolgreich

► **`inline double GetAngle( void )`** **PUBLIC**

gibt `dAngle` zurück, ohne diesen Wert vorher, durch lesen der Position im Antrieb, zu aktualisieren

► **`double GetAngle( BOOL, BOOL& )`** **PUBLIC**

ruft `GetAngle( BOOL )` und gibt bei Erfolg den danach aktualisierten Inhalt von `dAngle` zurück; der erste Parameter ist der Parameter für `GetAngle`, der Zweite zeigt an, ob das Aktualisieren erfolgreich war

► **`inline double GetAngleWidth( void )`** **PUBLIC**

gibt `dAngleWidth` (aktuelle Schrittweite in **Nutzereinheiten**) zurück; siehe [3]:
`mlGetValue( …, Width, … ), mGetValue( Width, … )`

► **`double GetMinAngleWidth( void )`** **PUBLIC**

gibt den in **Nutzereinheiten** umgerechneten Wert von `wPositionMinWidth` zurück; Rückgabewert ist stets positiv; siehe [3]: `mlGetValue( …, MinWidth, …), mGetValue( MinWidth, …)`

[5] ausgelassen werden: `dwVelocity`, `wPositionWidth`, `bUpwards`, `lPosition`, `dAngleMin`, `dAngleWidth`, `dAngleBias`, `dAngleMax`, `bIsCalibrated`

► **virtual double GetSpeed(void)** **PUBLIC**
hier nur Rückgabe von 1; siehe [3]:
`mlGetValue( …, Speed, … ), mGetValue( Speed, … )`
bei `TC_DRIVE` überschrieben, um `dwVelocity / fSpeedScale` zurückzugeben (aktuelle Geschwindigkeit in **Nutzereinheiten/ Sekunde**)

► **double GetMinSpeed(void)** **PUBLIC**
gibt die Minimalgeschwindigkeit in **Nutzereinheiten/ Sekunde** zurück; entspricht: `1 / fSpeedScale`

► **double GetMaxSpeed(void)** **PUBLIC**
gibt die Maximalgeschwindigkeit in **Nutzereinheiten/ Sekunde** zurück; entspricht:
`dwMaxVelocity / fSpeedScale`
(siehe [3]: `mlGetValue( …, MaxSpeed, … ), mGetValue( MaxSpeed, … ) )`

► **double GetAcceleration(void)** **PUBLIC**
gibt die aktuelle Beschleunigung (`dwAcceleration` umgerechnet in **Nutzereinheiten/ Sekunden²**) zurück; Rückgabewert ist stets positiv; siehe [3]:
`mlGetValue( …, Acceleration, …), mGetValue( Acceleration, …)`

► **BOOL IsDistanceRelative(void)** **PUBLIC**
gibt zurück, ob die *Relative Null* gesetzt ist; siehe [3]: `mIsDistanceRelative`

► **inline double GetAngleBias(void)** **PUBLIC**
gibt `dAngleBias` zurück; Offset für <Antrieb> in **Nutzereinheiten**

► **BOOL SetAngleBias(double)** **PUBLIC**
setzt `dAngleBias`; Offset für <Antrieb> in **Nutzereinheiten**

► **BOOL SetRelativeZero(void)** **PUBLIC**
Nur wenn der Antrieb steht, wird die *Relative Null* auf die aktuelle Absolutposition gesetzt. siehe [3]:
`mSetRelativeZero( TRUE, … )`
Rückgabewert: `TRUE` ↔ *Relative Null* ist gesetzt; kann auch eintreten, wenn sich der Antrieb bewegt und die Relative Null bereits definiert ist

► **void ResetRelativeZero(void)** **PUBLIC**
Nur wenn der Antrieb steht, wird die *Relative Null* aufgehoben. siehe [3]:
`mSetRelativeZero( FALSE, … )`
Rückgabewert: `TRUE` ↔ *Relative Null* ist aufgehoben; kann auch eintreten, wenn sich der Antrieb bewegt und die Relative Null bereits aufgehoben ist

► **virtual WORD GetStatus(void)** **PUBLIC**
hier nur Rückgabe von 255 (Hexadezimal: FF)
bei `TC_812` und `TC_832` überschrieben, um Statusinformationen des Antriebs abzurufen; Dort werden auch andere Werte zurückgegeben.

► **virtual long GetFailure(void)** **PUBLIC**
hier nur Rückgabe von 0

überschrieben bei `TC_Drive` – ruft hier drei Mal die Methode `_GetFailure` auf; Sobald die Methode einen Fehler meldet, wird der ermittelte „Fehlercode“ zurückgegeben. Wenn kein Fehler auftritt wird zu zuletzt ermittelte „Fehlercode“ zurückgegeben.

► **inline int GetId(void)** **PUBLIC**
gibt den Index, an welcher Stelle der Antrieb in Objekt `lpMList` steht, zurück; entspricht: `nId`

► **inline DWORD GetLimit(void)** **PUBLIC**
gibt die Distanz (in **Encoderschritten**) zurück, die der Antrieb zurückfahren soll, wenn die Hardwareschranke erreicht wird; entspricht: `dwRemoveLimit`

► **inline double GetAngleMin(void)** **PUBLIC**
gibt `dAngleMin` zurück

► **inline void SetAngleMin(double)** **PUBLIC**
setzt `dAngleMin`

► **double GetAngleRelativeMin(void)** **PUBLIC**
gibt `dAngleMin` um den Offset korrigiert zurück; entspricht: `dAngleMin + dAngleBias`

► **inline double GetAngleMax(void)** **PUBLIC**
gibt `dAngleMax` zurück

► **inline void SetAngleMax(double)** **PUBLIC**
setzt `dAngleMax`

► **double GetAngleRelativeMax(void)** **PUBLIC**
gibt `dAngleMax` um den Offset korrigiert zurück; entspricht: `dAngleMax + dAngleBias`

► **double GetMaxAngleWidth(void)** **PUBLIC**
gibt den in **Nutzereinheiten** umgerechneten Wert von `wPositionMaxWidth` zurück; Rückgabewert ist stets positiv

► **inline double GetInitialAngle(void)** **PUBLIC**
gibt `dInitialAngle` zurück

► **inline long GetPosition(void)** **PUBLIC**
gibt `lPosition` zurück, ohne diesen Wert vorher, durch Lesen der Position im Antrieb, zu aktualisieren, aktuelle Antriebsposition in **Encoderschritten**

► **inline void SetPosition(long)** **PUBLIC**
setzt `lDeltaPosition`; aktuelle Position in **Encoderschritten**

► **inline long GetPositionMin(void)** **PUBLIC**
gibt `lPositionMin` zurück; minimale Antriebsposition in **Encoderschritten**

► **inline void SetPositionMin(long)** **PUBLIC**
setzt `lPositionMin`; minimale Position in **Encoderschritten**

► **inline long GetPositionMax(void)** **PUBLIC**
gibt `lPositionMax` zurück; maximale Antriebsposition in **Encoderschritten**

► **inline void SetPositionMax(long)** **PUBLIC**
setzt `lPositionMax`; maximale Position in **Encoderschritten**

► **inline WORD GetPositionWidth(void)** **PUBLIC**
gibt `wPositionWidth` zurück; aktuelle Schrittweite in **Encoderschritten**

► **inline void SetPositionWidth(WORD)** **PUBLIC**
setzt `wPositionWidth`; aktuelle Schrittweite in **Encoderschritten**

► **inline long GetDistanceToZero(void)** **PUBLIC**
gibt `lDistanceToZero` zurück

► **inline void SetDistanceToZero (long)** **PUBLIC**
setzt `lDistanceToZero`

► **UINT GetDigits(void)** **PUBLIC**
gibt die Anzahl der Nachkommastellen `nDigits`, für alle reellwertigen Zahlen eines Antriebs, zurück

► **inline static int GetMaxId(void)** **PUBLIC**
gibt `nMaxId` zurück; maximale Anzahl der Antriebe in der Antriebsliste

► **inline TUnitType GetUnit(void)** **PUBLIC**
gibt `eUnit` zurück

► **inline LPCSTR GetUnitName(void)** **PUBLIC**
gibt `szUnit` zurück

► **virtual BOOL IsMoveFinish(void)** **PUBLIC**
hier nur Rückgabe von TRUE; siehe [3]: `mlIsMoveFinish`, `mIsMoveFinish` überschrieben bei `TC_Drive` (nur Rückgabe von `FALSE`), `TC_812`, `TC_812ISA` und `TC_832`. Rückgabewert bei den letzten drei Klassen: `TRUE` ↔ Antrieb steht, d.h. Abweichung zwischen Soll- und Istposition liegt innerhalb bestimmter Toleranzen; siehe `Deathband`

► **BOOL IsCalibrated(void)** **PUBLIC**
gibt zurück, ob der *Referenzpunktlauf* erfolgreich durchgeführt wurde; entspricht: `bIsCalibrated`

► **virtual BOOL IsIndexArrived(void)** **PUBLIC**
hier nur Rückgabe von TRUE
überschrieben bei `TC_812` und `TC_832`; Rückgabewert: `TRUE` ↔ *Referenzpunktlauf* beendet (entweder weil eine Hardwareschranke oder der *Referenzpunkt* erreicht wurde)

► **inline BOOL IsInitialMoveReady(void)** **PUBLIC**
gibt `bInitialMoveReady` zurück

► **inline BOOL IsRangeHit(void)** **PUBLIC**
gibt zurück, ob die zuletzt mit Methode `Translate` übersetzte Antriebsposition den Wertebereich verletzte; entspricht: `bRangeHit`; siehe [3]: `mIsRangeHit`

► **virtual BOOL IsLimitHit(void)** **PUBLIC**
hier nur Rückgabe von TRUE
überschrieben bei `TC_Drive` (nur Rückgabe von `FALSE`), `TC_812` und `TC_832`;
Rückgabe bei den letzten beiden Klassen: `TRUE` ↔ Hardwareschranken erreicht

► **LPCSTR pCharacteristic(void)** **PUBLIC**
gibt die "beliebige" Bezeichnung des Antriebs zurück; entspricht: `szCharacteristic`; siehe [3]: `mGetAxisName`

► **inline LPCSTR pSF(void)** **PUBLIC**
gibt das Zahlenformat `SFmt` zurück; siehe [3]: `mGetSF`

► **inline LPCSTR pDF(void)** **PUBLIC**
gibt das Zahlenformat `DFmt` zurück; siehe [3]: `mGetDF`

► **void funcd(double, double, double*, double*)** **PRIVATE**
Umrechnung von *internen Antriebspositionen* (erster Parameter, **Wertebereich `long` ausreichend**) in *Absolutpositionen* (Rückgabe in den letzten beiden Parametern); Zur Berechnung des dritten Parameters wird der zweite Parameter (`offset`) verwendet; `funcd` ist Umkehrfunktion von Methode `rtsave`

► **int rtsave(double, double, double, double, double*)** **PRIVATE**
Umrechnung von *Absolutpositionen* (erster Parameter) in *interne Antriebspositionen* (Rückgabe als letzter Parameter; **Wertebereich `long` ausreichend**); Der zweite (x_L) und dritte (x_R) Parameter kennzeichnen das Intervall ($[x_L \dots x_R]$) für das **Newton'sche Approximationsverfahren**. `rtsave` ist Umkehrfunktion von Methode `funcd`; Rückgabewert: `0` ↔ Berechnung erfolgreich (`-1` und `-2` kennzeichnen Fehlersituationen)

► **BOOL Translate(long&, double)** **PRIVATE**
passt den zweiten Parameter (*Absolutposition*) ins Intervall [`dAngleMin` … `dAngleMax`] ein und übersetzt ihn dann in eine *interne Antriebsposition*; Rückgabewert: `FALSE` ↔ Methode `rtsave` schlug fehlt (nur bei `eCorrState == CorrPolynom` möglich)

► **BOOL Translate(double&, long)** **PRIVATE**
übersetzt den zweiten Parameter (*interne Antriebsposition*) in eine *Absolutposition* und gibt das Ergebnis als ersten Parameter zurück; Rückgabewert: stets `TRUE`

► **virtual BOOL MoveToPosition(long)** **PUBLIC**
setzt `lPosition` auf den als Parameter übergebenen Wert (in **Nutzereinheiten**); Rückgabewert: stets `TRUE`

bei `TC_Drive` überschrieben, um das Antriebsspiel zu berücksichtigen und um den Antrieb zur angegeben Position zu bewegen; Rückgabewert der (zur Bewegung verwendeten) Methode `MoveAbsolute` wird zurückgegeben

► **virtual BOOL MoveByPosition(long)** **PUBLIC**
erhöht `lPosition` um das Offset (Parameter in **Encoderschritten**); Rückgabewert: stets `TRUE`

bei `TC_Drive` überschrieben, um die Methode `MoveRelative` der abgeleiteten Klassen aufzurufen und deren Rückgabewert zurückzugeben

► **BOOL MoveToAngle(double)** **PUBLIC**
übersetzt den Parameter (*Absolutposition* in **Nutzereinheiten**) in eine *interne Antriebsposition* (Methode `Translate`) und fährt diese an (Methode `MoveToPosition`); siehe [3]: `mlMoveToDistance` und `mMoveToDistance`
Rückgabewert: `TRUE` ↔ Übersetzung und Positionierung erfolgreich

► **BOOL MoveByAngle(double)** **PUBLIC**
ermittelt die aktuelle *Absolutposition* und addiert das Offset (Parameter in **Nutzereinheiten**); rechnet diese in eine *interne Antriebsposition* um und fährt sie an; siehe [3]: `mMoveByDistance`
Rückgabewert: `TRUE` ↔ Ermittlung, Übersetzung und Positionierung erfolgreich

► **BOOL SetAngleWidth(double)** **PUBLIC**
übersetzt den Parameter (*Absolutposition* in **Nutzereinheiten**) in eine *interne Antriebsposition* und passt diese ins Intervall [`wPositionMin` … `wPositionMax`] ein; Anschließend wird der Wert wieder in **Nutzereinheiten**, umgerechnet, um in `dAngleWidth` gespeichert zu werden. siehe [3]: `mSetValue( Width, …)`

► **virtual BOOL SetSpeed(double)** **PUBLIC**
hier nur Rückgabe von TRUE; siehe [3]: `mSetValue( Speed, … )`

bei `TC_DRIVE` überschrieben; rechnet den Parameter (**Nutzereinheiten/ Sekunde**) in (**Encoderschritten/ Sekunde**) um, speichert ihn in `dwVelocity` und setzt ihn als aktuelle Geschwindigkeit; Rückgabewert: stets `TRUE`

► **inline void SetCalibrationState(BOOL state)** **PUBLIC**
setzt `bIsCalibrated` auf den als Parameter übergebenen Wert

► **void SetCorrectionState(BOOL)** **PUBLIC**
wenn der Parameter `TRUE` ist und (`bCorrection == TRUE` und `bIsCalibrated == TRUE`) gilt, wird `eCorrStatus` auf `CorrPolynom` gesetzt; sonst immer auf `CorrLinear`;
siehe [3]: `mSetCorrectionState`

► **BOOL SetAngleRange(void)** **PUBLIC**

übersetzt die *internen Antriebspositionen* `lPositionMin` und `lPositionMax` in *Absolutpositionen* und speichert diese als `dAngleMin` bzw. `dAngleMax`; Rückgabewert: stets `TRUE`

► **virtual BOOL SetLimit(DWORD)** **PUBLIC**

hier nur Rückgabe von TRUE

bei `TC_812` und `TC_832` wird `dwRemoveLimit` der Wert des Parameters zugewiesen; Bei `TC_812`, wird zusätzlich ein Kommando (siehe `ExecuteCmd`) ausgeführt.
Rückgabewert: `FALSE` ↔ Kommando nicht erfolgreich ausgeführt

► **virtual BOOL SetLine(int, BOOL)** **PUBLIC**

hier nur Rückgabe von TRUE; siehe [3]: `mSetLine`

überschrieben bei `TC_812` um den Ausgangsport des Controllers C-812 anzusprechen; Der erste Parameter (`channel` ∈ [`1 ... 16`]) wählt einen Kanal. Der zweite Parameter (`BOOL`) regelt die Spannung des Ports: `0` (`FALSE`) oder `5` (`TRUE`) Volt. Rückgabewert: `FALSE` ↔ Kommando nicht erfolgreich ausgeführt

► **virtual BOOL SetHome(void)** **PUBLIC**

stellt den Zustand her, als ob das Programm beendet und neu gestartet wurde (keine *Relative Null*, *Antriebsposition* `0`, usw.); Wird nur intern zum Kalibrieren und nach dem *Referenzpunktlauf* verwendet. Rückgabewert: stets `TRUE`

► **BOOL PushSettings(void)** **PUBLIC**

speichert einige ausgewählte Attribute in `Settings` ab, so dass sie mit Methode `PopSettings` wiederhergestellt werden können; siehe [3]: `mPushSettings`

bei `TC_Drive` überschrieben (gibt nur `TRUE` zurück), **die Methode ist in `TMotor` nicht als virtuell deklariert**

► **BOOL PopSettings(TMParameter)** **PUBLIC**

stellt die als Parameter übergebenen Attribute wieder her und fährt die Absolutposition wieder an, die bei Methode `PushSettings` gespeichert wurde; siehe [3]: `mPopSettings`

bei `TC_Drive` überschrieben (gibt nur `TRUE` zurück), **die Methode ist in `TMotor` nicht als virtuell deklariert**

► **virtual BOOL StartLimitWatch(void)** **PUBLIC**

hier nur Rückgabe von R_OK

bei `TC_832` überschrieben – dort wird **Limit Watch** gestartet; Bei aktivem *Limit Watch* werden Antriebsinformationen in Hauptfenster ‚Steuerprogramm' ausgegeben – siehe Methode `TMain::SetWatchIndicator` . Rückgabewert: `FALSE` ↔ *LimitWatch* bereits aktiv

► **virtual BOOL StopLimitWatch(void)** **PUBLIC**

hier nur Rückgabe von R_OK

bei `TC_832` überschrieben, dort wird *Limit Watch* gestoppt**;** Rückgabewert: `FALSE` ↔ *LimitWatch* nicht aktiv

► **`virtual BOOL ActivateDrive( void )`** **PUBLIC**

hier nur Rückgabe von TRUE; siehe [3]: `mActivateDrive`

überschrieben bei `TC_Drive` (nur Rückgabe von `FALSE`), `TC_812` und `TC_832`; Bei den letzten beiden Klassen wird der Antrieb im Controller ausgewählt. Der Antrieb muss ausgewählt werden, bevor die Bewegung gestartet wird! Rückgabewert `TRUE` ↔ erfolgreich ausgewählt

► **`virtual BOOL StopDrive( BOOL )`** **PUBLIC**

hier nur Rückgabe von TRUE; siehe [3]: `mStopDrive`

überschrieben bei `TC_812` und `TC_832` um die Bewegung des Antriebs zu stoppen; Der Parameter gibt an, ob die Position eingeregelt werden soll. Rückgabewert `TRUE` ↔ erfolgreich gestoppt und ggf. eingeregelt

► **`virtual void OptimizingDlg( void )`** **PUBLIC**

hier leer; siehe [3]: `mlOptimizingDlg`

bei `TC_812` und `TC_832` implementiert – dort werden die Dialogfenster ‚DC-Controler Parameter C-812' bzw. ‚DC-Controler Parameter C-832' angezeigt; siehe [7]

► **`virtual void StartCheckScan( void )`** **PUBLIC**

hier leer; siehe [3]: `mStartMoveScan`

bei `TC_812ISA`, `TC_812GPIB` und `TC_832` überschrieben; Dort wird der Scan zum Optimieren des Anfahrverhaltens implementiert.

► **`virtual BOOL StartToIndex( long& )`** **PUBLIC**

hier nur Rückgabe von TRUE

überschrieben bei `TC_812` und `TC_832`, dort starten des *Referenzpunktlaufs*; Rückgabewert: stets `TRUE`

► **`virtual int ExecuteCmd( LPSTR )`** **PUBLIC**

hier nur Rückgabe von R_OK; siehe [3]: `mExecuteCmd`

bei `TC_812ISA` und `TC_812GPIB` überschrieben, dort wird der (als Zeichenkette übergebene) Parameter direkt an den jeweiligen Controller (als Kommandofolge) gesendet; Rückgabewert:

- `0` → `bControlBoardOK == FALSE`
- `R_FAILURE` → es wird bereits ein Kommando ausgeführt
- `R_OK` → sonst

IV.1.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	989
	‚**LOC** of **I**mplementation'*	LOCI	810
	‚**LOC** of **D**eclaration'*	LOCD	179
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	**48**
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	**77**
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	**125**
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	0
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	13
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	**28**
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	59
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	260
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	**5**
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'*	NOC	96
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	16

Tabelle 5 „ausgewählte Metriken der Klasse `TMotor`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

Anmerkungen zur Fehleranfälligkeit und -toleranz der Klasse `TMotor` findet man (wenn nötig) dem jeweiligen Member, rot hervorgehoben.

IV.2 Klasse TMList

Deklaration : M_MOTCOM.H

Implementation : MOTORS.CPP

Zur Verwaltung der angeschlossen Antriebe, wird beim Start der MOTORS.DLL das Objekt `lpMList` erzeugt und initialisiert (Datei M_LAYER.CPP) – siehe ►**Dokumentation des Istzustands**.

IV.2.1 Attribute

Beim Erzeugen eines neuen `TMotor`-Objektes, registriert sich jeder Antrieb im Objekt `lpMList`, indem er sein `nId` (je nach Achse) in einem der folgenden Attribute abgelegt:

Attribut in `TMList`			`TAxisType`
►	**`int AD_Z`**	**PUBLIC**	`Z`
►	**`int AD_Y`**	**PUBLIC**	`Y`
►	**`int AD_X`**	**PUBLIC**	`X`
►	**`int AD_Tilt`**	**PUBLIC**	`Tilt`
►	**`int AD_DC`**	**PUBLIC**	`DC`
►	**`int AD_Collimator`**	**PUBLIC**	`CC`
►	**`int AD_DF`**	**PUBLIC**	`DF`
►	**`int AD_Encoder`**	**PUBLIC**	`Encoder`
►	**`int AD_Phi`**	**PUBLIC**	`Phi`
►	**`int AD_Psi`**	**PUBLIC**	`Psi`
►	**`int AD_Rotation`**	**PUBLIC**	`Rotation`
►	**`int AD_Absorber`**	**PUBLIC**	`Absorber`
►	**`int AD_Monochromator`**	**PUBLIC**	`Monochromator`
►	**`int AD_Theta`**	**PUBLIC**	`Theta`
►	**`int AD_Omega`**	**PUBLIC**	`Omega`

Tabelle 6 „Beziehung zwischen Attribut in `TMList` und zugeordneter Bewegungs-Achse“ (Quelle: selbst)

Wenn für eine Achse (z.B. `DC`) kein Antrieb existiert, dann hat das zugeordnete Attribut (hier `AD_DC`) den Wert `-1`.

Wenn mehrere Antriebe pro Achse erzeugt werden, wird der erste Index überschreiben und es gilt nur der letzte Antrieb.

► **`TMotor** aMotor`** **PRIVATE**

eine Liste von `TMotor`-Objekten, die in Methode `InitializeModule` gefüllt wird; Sie enthält mindestens einen Eintrag (ggf. wird ein `TMotor`-Objekt erzeugt) – maximal `nMaxNumber` Einträge.

► **`Int nMaxNumber`** **PRIVATE**

entspricht der maximalen Anzahl zu verwaltender Antriebe plus eins (siehe Wertebereich von `nLastDrive`); Mehr Antriebe werden bei Methode `InitializeModule` nicht erzeugt, weitere Antriebe werden ignoriert. `nMaxNumber` wird im Konstruktor initialisiert (Standardwert: `10`).

► **`Int nLastDrive`** **PRIVATE**

Index des zuletzt in `aMotor` aufgenommenen `TMotor`-Objekts; ist stets im Intervall [`0` … `nMaxNumber`]

► **`Int nActiveDrive`** **PRIVATE**

Index eines **ausgewählten Antriebs**, dessen Referenz bei Methode `MP( void )` zurückgegeben wird; `nActiveDrive` kann bei Methode `SetAxis` gesetzt werden.

IV.2.2 Methoden

► **`TMList( int )`** **PUBLIC**

Der Konstruktor initialisiert alle (unter **IV.2.1 Attribute**) genannten Attribute. Der Parameter gibt die maximale Anzahl zu verwaltender Antriebe an (Wert wird in `nMaxNumber` gespeichert).

► **`TMList( const TMList& )`** **PRIVATE**

leer implementierter privater Copy-Konstruktor, verhindert das erzeugen von weiteren `TMList`-Objekten durch Zuweisung

► **`BOOL InitializeModule( void )`** **PUBLIC**

erzeugt und initialisiert für jeden [MOTOR<M>]-Abschnitt der HARDWARE.INI je ein entsprechendes `TMotor`-Objekt und legt dieses in `aMotor` ab; siehe `nMaxNumber` und `nLastDrive`
Rückgabewert: stets `TRUE`

Die Indizies, wie sich `TMotor` in `lpMList` registriert und wie sie die Methode `InitializeModule` vergibt, können erheblich variieren. Das kann im Programm zu schwerwiegenden Zuordnungsfehlen führen: mit Abstürzen oder das sich ein falscher Antrieb bewegt! Das ist bereits der Fall wenn:

- **ein Motorabschnitt in der ini-Datei gelöscht wird**
- **wenn ein `TMotor`-Objekt entsteht, das nicht von `lpMList` erzeugt wurde**
- **wenn mehr Antriebe erzeugt werden, als `aMotor` aufnehmen kann**

► **`~TMList( void )`** **PUBLIC**

gibt die bei Methode `InitializeModule` erzeugte Antriebsliste wieder frei

► **`BOOL SaveModuleSettings( void )`** **PUBLIC**

speichert den Großteil der Attribute der (in `aMotor` vorhandenen) Antriebe (siehe Methode `TMotor::SaveSettings`) und stoppt ihre Bewegung; siehe [3]: `mlSaveModuleSettings`
Rückgabewert: stets `TRUE`

► **`TMotor* MP( void )`** **PUBLIC**

gibt die `TMotor`-Referenz des *ausgewählten Antriebs* (der sich in `aMotor` an Position `nActiveDrive` befindet) zurück; entspricht: `aMotor[ nActiveDrive ]`

► **`TMotor* MP( int )`** **PUBLIC**

überprüft, ob der Parameter (`index`) im Intervall [`0` … `nLastDrive`] liegt:

- ja → der Zeiger des <`index`>-ten Antriebs in `aMotor` wird zurückgegeben
  (z.B. `index == 0`, dann wird der zuerst erzeugte Antrieb[6] zurückgegeben)
- nein → Rückgabewert von Methode `MP( void )`

► **`int GetAxis( void )`** **PUBLIC**

gibt den Index des *ausgewählten Antriebs* zurück; entspricht: `nActiveDrive`; siehe [3]: `mlGetAxis`

► **`int LastId( void )`** **PUBLIC**

gibt den Index des zuletzt in `aMotor` aufgenommenen `TMotor`-Objekts zurück; entspricht: `nLastDrive`

► **`int GetAxisNumber( void )`** **PUBLIC**

gibt die Anzahl der Antriebe in `aMotor` zurück; entspricht: `nLastDrive + 1`, siehe [3]: `mlGetAxisNumber`

---

[6] `aMotor[0]`

► **`static TAxisType ParsingAxis( LPSTR )`** **PUBLIC**
übersetzt die (als Parameter übergebene) "beliebige", texturelle Bezeichnung eines Antriebs in die ihm zugeordnete Achse und gibt diese zurück; Wenn die Bezeichnung nicht in **Tabelle 2** enthalten ist, wird `0` zurückgegeben. siehe **[3]**: `mlParsingAxis`

► **`BOOL SetAxis( int )`** **PUBLIC**
prüft, ob der als Parameter (`index`) übergebene Wert im Intervall [`0` … `nLastDrive`] liegt (siehe **[3]**: `mlSetAxis`)
- ja → setzt den *ausgewählten Antrieb* (`nActiveDrive`) auf `index` und gibt `TRUE` zurück
- nein → setzt `0` als *ausgewählten Antrieb* und gibt `FALSE` zurück

► **`void SetAngleDefault( void )`** **PUBLIC**
hebt für alle Antriebe in `aMotor` die *Relative Null* auf und aktualisiert die Werte `dAngleMin` und `dAngleMax` in `TMotor`; siehe **[3]**: `mlSetAngleDefault`

► **`void PositionControlDlg( void )`** **PUBLIC**
zeigt das Dialogfenster ‚Verfahren nach Encoder-Position' an; siehe **[7]** und **[3])**: `mlPositionControlDlg`

► **`void SetParametersDlg( void )`** **PUBLIC**
zeigt das Dialogfenster ‚Motor-Parameter' an; siehe **[7]** und **[3]**: `mlSetParametersDlg`

► **`void InquireReferencePointDlg( BOOL )`** **PUBLIC**
zeigt das Dialogfenster ‚Grundeinstellung anfahren' an (siehe **[7]**); Der übergebene Parameter konfiguriert das Dialogfenster (wenn alle Antriebe kalibriert werden sollen ist `task == TRUE`; sonst kann nur ein Antrieb kalibriert werden.).
siehe **[3]**: `mlInquireReferencePointDlg`

IV.2.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	445
	‚**LOC** of **I**mplementation'*	LOCI	399
	‚**LOC** of **D**eclaration'*	LOCD	46
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	19
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	13
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	32
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	2
	‚**N**umber **O**f **O**verridden **M**ethods'	NOOM (0, 10)	0
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	12
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	88
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	27
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'*	NOC	26
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	17

Tabelle 7 „ausgewählte Metriken der Klasse `TMList`" (Quelle: Together® ‚Version 6.0)

* Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

Anmerkungen zur Fehleranfälligkeit und -toleranz der Klasse `TMList` findet man (wenn nötig) dem jeweiligen Member, rot hervorgehoben.

V Attribute

► **`LPMList lpMList`** **`GLOBAL`**
definiert und initialisiert in M_LAYER.CPP, beinhaltet die Liste der zu verwaltenden Antriebe; siehe **IV.2 Klasse TMList**

VI Methoden (C-Interface zur Antriebssteuerung)

Das zur Antriebssteuerung verwendete C-Interface wurde bereits einem Reverse-Engineering Prozess unterzogen. Sehr detaillierte Informationen findet man unter **[3]**.

VII Anhang

VII.1 Verwandte Dokumente

[1] „Verhaltensspezifikation: XCTL-Steuerprogramm – Teil: Motorsteuerungs-Komponente (motors.dll)“, Version 0.9 von Stefan Lützkendorf

[2] „Komponente: Diffraktometrie/ Reflektometrie Ablauf und Einstellungen“, Designbeschreibung von Jens Ullrich und Stephan Berndt

[3] „Beschreibung einer Schnittstelle zur Motorenansteuerung: Das C-Interface des RTK-Steuerungsprogramms“, Studienarbeit von Sebastian Freund und Derrick Hepp

[4] „Motorsteuerung −> Design −> Dateiübersicht“, am 03.04.2002 von Kay Schützler erstellt

[5] „Toter Code: Ergebnisse mit SNiFF und McCabe“, am 15.09.99 von Stefan Lützkendorf erstellt

[6] „Layoutkonventionen und Steuerelemente“, Version 1.0 von Thomas Kullmann und Günther Reinecker

[7] „Windows-Ressourcen, Ressourcenübersicht“, am 01.06.2002 von Thomas Kullmann und Günther Reinecker erstellt

VII.2 Index

VII.3 Tabellen

VII.4 Abbildungen