Kapitel 3

Die Umgebungssimulation

3.1 Ziel

Hauptziel der Simulation der Motoren ist die Möglichkeit, die XCTL-Software testen zu können, ohne dass die Motoren-Hardware zur Verfügung steht bzw. diese durch ungetestete Software zu gefährden. Das ist nicht nur für den Test der Motorenkomponente notwendig, sondern auch für den Test anderer Komponenten, die Steuerungsabläufe implementieren und dabei die Motorenkomponente verwenden.

Im bestehenden XCTL-System gibt es für die Aufgabe der Motorensimulation einen einfachen Implementationsansatz: einen Motor vom Typ `TMotor`, der aber für die meisten Funktionen nur leere Implementationen anbietet und jede Bewegung ohne die durch eine solche erforderliche Zeitverzögerung durchführt. Dies ist für eine Sicht von außerhalb der Motorenkomponente z.T. ausreichend, lässt aber alle Implementationen, die tatsächlich Hardware ansteuern ebenso unberücksichtigt wie die Probleme, die durch Zeitverzögerungen und Ungenauigkeiten der Hardware entstehen.

Die Motorensimulation soll also folgende Dinge ermöglichen:

- das Betreiben und Testen der Softwareteile die direkt mit der Hardware kommunizieren ohne dass die Hardware zur Verfügung steht
- und das Simulieren des Zeit- und Verzögerungsverhaltens der Hardware.
- Außerdem soll für den Softwaretest die Kommunikation protokollierbar sein, um eine überprüfbare Ausgabe zu haben und die Kommunikation soweit möglich auf Korrektheit überprüfen zu können.

Dazu müssen müssen die Befehle zur Ansteuerung der Hardware von der zu erstellenden Simulationskomponente semantisch äquivalent umgesetzt werden. Um den Aufwand zu beschränken soll zunächst nur der Teil der Motor- bzw. Motorcontroller-Funktionen simuliert werden, der von der aktuellen Implementation verwendet wird, wobei natürlich beim Entwurf eine spätere Vervollständigung berücksichtigt werden muss. Ebenso wird natürlich nur der Teil simuliert der unmittelbar für die Software sichtbar ist.

3.2 Technische Grundlagen

Das XCTL-System unterstützt zur Zeit zwei verschiedene Motorcontroller: die Gleichstrom-Motor-Controller C-812 und C-832 der Firma *Physik Instrumente (PI) GbmH, Waldbronn*. Die grundlegende Dokumentation ist in den Dokumenten [PI93a] und [PI93b] zu finden. Ein Überblick über die Ansteuerung der Motorcontroller ist in [FH01, S. 19ff] zu finden. Der Anhang B stellt die wesentlichen Informationen, die notwendig sind, die Kommunikation zwischen Software und Hardware zu verstehen, zusammen.

3.3 Integration in das vorhandene Software-System

3.3.1 Die Ansatzstelle für die Simulation

Als erstes stellt sich bei der Realisierung die Frage: Wo genau soll die Simulation ansetzen? Antwort: Möglichst weit "Unten", d.h. möglichst weit an der Aussengrenze der Motorenkomponente. Im Idealfall liegt diese Ansatzstelle außerhalb der zu testenden Komponente, so dass diese für die Integration der Simulation nicht verändert werden muss.

Das setzt aber eine Softwareschicht zwischen Motorenkomponente und Hardware voraus, die uns zugänglich ist. Diese ist nur im Fall der IEEE488-Kommunikation mit dem C812er Controller vorhanden, bei der die eigentliche Kommunikation in einer eigenen, als DLL vorliegenden, Komponente gekapselt ist.

Daher bleibt uns nur der Weg, eine Ansatzstelle für die Simulation zu finden, die innerhalb der Motorenkomponente und möglichst dicht an der Komponentengrenze liegt, und für deren Realisierung nur wenige, gut nachvollziehbare Änderungen notwendig sind, so dass wir sicherstellen können, dass das Softwareverhalten durch diese Änderungen nicht beeinflusst wird.

Diese Stelle liegt bei der Implementation für den C812er Controller direkt vor bzw. nach dem Schreiben bzw. Lesen vom Direktspeicher. Dieses Lesen oder Schreiben erfolgt über, mit den entsprechenden Adressen initialisierte, `char`-Pointer (s. Abb 3.1).

```
char* lpIn;  // Membervariablen der
char* lpOut; // Motorklasse TC_812ISA
...
// lesen vom Direktspeicher
char c = *lpIn;
...
// schreiben auf den Direktspeicher
*(lpOut + offset) = c;
```

Abbildung 3.1: Hardwarezugriffe für C812 (Orginalsystem)

Um genau eine Ansatzstelle für das Beobachten bzw. Beeinflussen dieser Kommunikation zu bekommen, habe ich die Schreib- und Lesezugriffe in Funktionen gekapselt (Abb. 3.2) und alle Zugriffe auf den Direktspeicher durch entsprechende Funktionsaufrufe ersetzt (Abb. 3.3).

```
inline static char C812ISA_Get( char* addr )
{
   char return_val = 0;
   // Ansatzstelle vor Kommunikation
   return_val = *addr;
   // Ansatzstelle nach Kommunikation
   return return_val;
}
inline static void C812ISA_Put( char* addr, char c )
{
   // Ansatzstelle vor Kommunikation
   *addr = c;
   // Ansatzstelle nach Kommunikation
}
```

Abbildung 3.2: Hardwarezugriffswrapper für C812 (Prinzip)

```
char* lpIn;  // Membervariablen der
char* lpOut; // Motorklasse TC_812ISA
...
// lesen vom Direktspeicher
char c = C812ISA_Get( lpIn );
...
// schreiben auf den Direktspeicher
C812ISA_Put( lpOut + offset, c );
```

Abbildung 3.3: Hardwarezugriffe für C812 (neue Version)

Entsprechend wurde mit den Stellen zur Hardwarekommunikation der Implementation für den C832er Controller verfahren.

Insgesamt musste für diese Änderung der Quelltext an jeweils etwa 25 Stellen für Lese- und Schreibzugriffe modifiziert werden. Die Änderungen sind sehr einfach, und somit können wir sicher sein, das Verhalten der Komponente nicht verändert zu haben. Für zukünftige Portierungen des XCTL-Systems, bei denen die Hardwarezugriffe ein besonderes Problem sein werden, haben die Änderungen den Vorteil, dass es nur noch jeweils eine Stelle im Quelltext gibt, an denen die verschiedenen Arten der Hardwarezugriffe auftreten.

3.3.2 Die Architektur der Erweiterung

Nachdem geklärt wurde, wo die Simulation prinzipiell ansetzen kann, muss nun die Software-Architektur für die Einbindung einer Simulation geklärt werden.

Die Motoren- und die Simulationskomponente sollen voneinander unabhängig sein, und nur im Bedarfsfall eine Verbindung untereinander bekommen

- um sicherzustellen, dass die XCTL-Software an den vorhandenen Arbeitsplätzen auch ohne die installierte Simulationskomponente weiter funktioniert,

- um die Anwendung, die die Motorenkomponente verwendet, in die Lage zu versetzten über das *Ob*, die *Art und Weise* bzw. die *Auswahl einer speziellen Implementation* der Simulation zu entscheiden,

- und um eine getrennte Arbeit an den verschiedenen Software-Komponenten zu ermöglichen.

Ziel ist also eine UML-Komponentenstruktur wie im Komponentendiagramm in Abb. 3.4, in der, wie im ursprünglichen XCTL-System, eine Beziehung zwischen der Anwendung (`develop.exe`) und der Motorenkomponente (`motors.dll`) besteht, und zusätzlich eine Beziehung zwischen der Anwendung und der Simulationskomponente (`msim.dll`) existiert. Wichtig ist, dass zwischen der Motoren- und der Simulationskomponente keine Beziehung besteht.

Die Beziehung zwischen Anwendung und Motorenkomponente ist eine *link-time*-Beziehung, d.h. wird durch den Binder in der Produktionsphase hergestellt. Die Anwendung ist ohne das Vorhandensein der Motorenkomponente nicht lauffähig. Dagegen soll zwischen Anwendung und Simulationskomponente eine *run-time*-Beziehung bestehen, also eine Beziehung die erst zur Laufzeit durch die Anwendung hergestellt wird, und auch nur dann, wenn sie benötigt wird. Dadurch ist die Anwendung auch ohne das Vorhandensein der Simulationskomponente lauffähig.

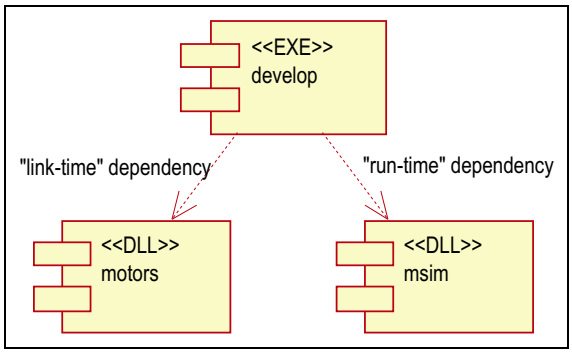


Abbildung 3.4: Angestrebte Komponentenstruktur

In der Realisation einer solchen Struktur müssen die einzelnen Komponenten bestimmte Aufgaben übernehmen bzw. Dienste anbieten.

Die Motorenkomponente muss zusätzlich zu ihren bisherigen Diensten die Möglichkeit bieten, festzulegen, ob simuliert werden soll, und eine Verbindung zu einer Simulation herstellen.

Die Simulationskomponente bietet eine Implementation einer Simulation an, die dem Zustand der Motorenkomponente entsprechend konfigurierbar ist, und die, nach Herstellung der Verbindung, die Kommunikation mit der Motorenkomponente übernimmt. Sie kann auch zusätzliche Dienste anbieten, wie z.B. das Protokollieren der Software/Hardware-Kommunikation oder eine Statusanzeige der simulierten Hardware.

Die Anwendung hat dann die Aufgabe, die beiden Komponenten zu initialisieren, zu konfigurieren und die Verbindung zwischen ihnen herzustellen.

3.3.3 Die modifizierte Schnittstelle der Motorenkomponente

Übersicht

Um die oben genannten Aufgaben zu erfüllen, musste die in `m_layer.h` definierte Schnittstelle der Motorenkomponente erweitert werden[1].

Im Idealfall einer objektorientierten Motoren-Schnittstelle könnte die Motorenkomponente ein Interface definieren, das von einer Klasse einer Simulationskomponente zu implementieren wäre, die dann zur Laufzeit bei der Motorenkomponente zu registrieren wäre (s. Abb. 3.5).

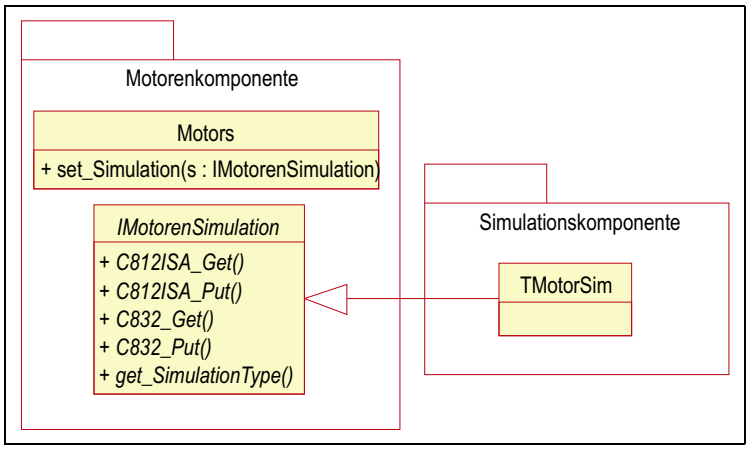


Abbildung 3.5: Mögliche OO-Schnittstelle für Motorsimulation (Prinzip)

Die Motoren-Schnittstelle im XCTL-System ist aber eine C-Schnittstelle, weshalb eine C-konforme Erweiterung notwendig war (s. Abb. 3.6). Diese enthält 4 Funktionstypen die jeweils den Typ einer callback-Funktion definieren sowie 4 Funktionen zum Registrieren von entsprechenden Implementationen dieser Funktionstypen. Außerdem enthält sie eine Funktion für die Festlegung des erforderlichen Simulationsmodus.

Die Motorenkomponente unterstützt 3 Modi der Simulation:

Normalmodus
: Das ist der Standardmodus, in dem sich die Motorenkomponente wie im bisherigen XCTL-System verhält. Es wird versucht direkt mit der Hardware zu kommunizieren. Eventuell registrierte callbacks werden ignoriert.

  Wird dieser Modus verwendet, ohne dass die Hardware vorhanden und entsprechend konfiguriert ist, wird die Motorenkomponente, wie bisher, Fehler melden. Es besteht aber weiterhin die Option, Motoren vom Typ `TMotor` zu verwenden.

Simulationsmodus
: In diesem Modus werden alle Hardware-Lese- und -Schreiboperationen an die entsprechenden, zu registrierenden callback-Funktionen delegiert. Üblicherweise treiben diese callbacks eine Simulation.

[1] Eine vorläufige Beschreibung der Schnittstelle ist in [FH01, Anhang A, S. 137-168] zu finden.

Werden keine callbacks registriert, reagiert die Motorenkomponente wie im Normalmodus.

Vergleichs- oder Protokollmodus
: In diesem Modus wird, wie im Normalmodus, direkt mit der Hardware kommuniziert. Parallel dazu werden jedoch die callback-Funktionen wie im Simulationsmodus aufgerufen, aber die dort generierten Werte ignoriert.

  Dieser Modus hat den Zweck, eine Möglichkeit zu bieten, die tatsächliche Kommunikation der Software mit der Hardware zu beobachten; entweder um diese zu protokollieren, oder aber um eine eventuelle Simulation zu testen, indem sie parallel betrieben wird und die von ihr generierten Werte mit denen von der Hardware gelieferten verglichen werden.

Details

Im folgenden Abschnitt werden die einzelnen Typen und Funktionen der Erweiterung der in `m_layer.h` definierten C-Schnittstelle der Motorenkomponente beschrieben.

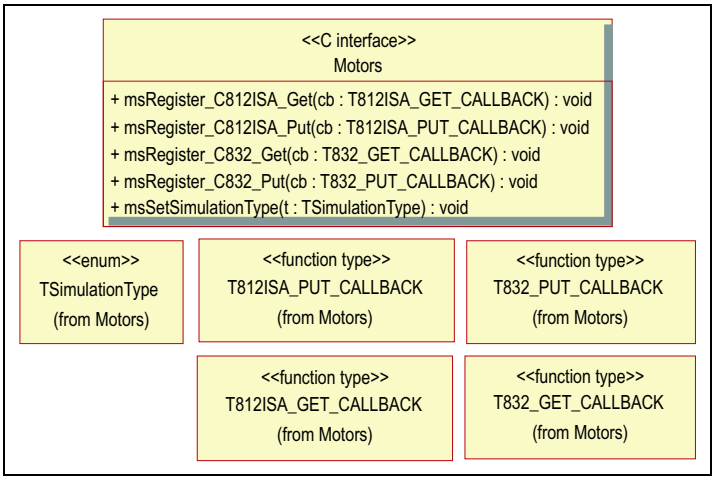


Abbildung 3.6: Erweiterung des C-Interfaces der Motorenkomponente

Um im Namensschema der Schnittstelle zu bleiben, wurden die Funktionen mit dem Präfix `ms` versehen, was als "Motoren Simulation" zu lesen ist.

Bei allen Funktionen ist zu beachten, dass sie die Art und Weise, wie die Hardware angesprochen wird, bestimmen. D.h., dass alle derartigen Einstellungen *vor* der eigentlichen Initialisierung der Motorenkomponente (durch den Aufruf von `mlInitializeMotorsDLL`) erfolgen müssen, da während dieser Initialisierug bereits die Hardware angesprochen wird. Es ist *nicht* vorgesehen, dass die Einstellungen zum Simulationsmodus nach der Initialisierung verändert werden.

Funktion: `msSetSimulationType`

Signatur
: `void WINAPI msSetSimulationType(TSimulationType t)`

Beschreibung
Informiert die Motorenkomponente über den gewünschten Simulationsmodus.

Parameter
`t==no_simulation` – stellt den Normalmodus ein (Voreinstellung)

`t==simulation_only` – stellt den Simulationsmodus ein

`t==test_simulation` – stellt den Vergleichs- oder Protokollmodus ein

Funktionstype: `T812ISA_GET_CALLBACK`

Signatur
```
char WINAPI (*T812ISA_GET_CALLBACK) (char* addr, char hw_value)
```

Beschreibung
Typ für callback-Funktion für die Leseoperation vom C812er Controller über die 'ISA'-Schnittstelle.

Parameter
`addr` – Speicheradresse von der gelesen werden soll.

`hw_value` – Hardware-Vergleichswert. Im Fall, dass der Simulationstyp auf `test_simulation` gesetzt wurde, stellt dieser Parameter den von der Hardware tatsächlich gelieferten Wert zur Verfügung.

Rückgabewert
Simuliertes Leseergebnis. Das von der callback-Funktion gelieferte Ergebnis wird von der Motorenkomponente verwendet, als wäre es von der Hardware erzeugt worden. Im Fall, dass der Simulationstyp auf `test_simulation` gesetzt wurde, wird der Rückgabewert ignoriert.

Funktionstype: `T812ISA_PUT_CALLBACK`

Signatur
```
void WINAPI (*T812ISA_PUT_CALLBACK) (char* addr, char put)
```

Beschreibung
Typ der callback-Funktion für die Schreiboperation an den C812er Controller über die 'ISA'-Schnittstelle.

Parameter
`addr` – Speicheradresse auf die geschrieben werden soll.

`put` – Der zu schreibende Wert.

Funktionstype: `T832_GET_CALLBACK`

Signatur
```
int  WINAPI (*T832_GET_CALLBACK) (unsigned port, int hw_value)
```

Beschreibung
Funktionstyp des callback für die Leseoperation vom C832er Controller.

Parameter
`port` – I/O-Adresse von der gelesen werden soll.

`hw_value` – Hardware-Vergleichswert. Im Fall, dass der Simulationstyp auf `test_simulation` gesetzt wurde, stellt dieser Parameter den von der Hardware tatsächlich gelieferten Wert im niederwertigen Byte zur Verfügung.

Rückgabewert
Simuliertes Leseergebnis. Das von der callback-Funktion gelieferte Ergebnis wird von der Motorenkomponente verwendet als wäre es direkt von der angegebenen I/O-Adresse mit tt inp gelesen worden. Im Fall, dass der Simulationstyp auf `test_simulation` gesetzt wurde wird der Rückgabewert ignoriert.

Funktionstype: `T832_PUT_CALLBACK`

Signatur
```
void WINAPI (*T832_PUT_CALLBACK) (unsigned port, int put)
```

Beschreibung
Funktionstyp des callback für die Schreiboperation an den C832er Controller.

Parameter
`port` – I/O-Adresse an die der zu übergebene Wert geschrieben werden soll.

`put` – Der zu schreibende Wert.

Funktion: `msRegister_C812ISA_Get`

Signatur
```
void WINAPI msRegister_C812ISA_Get(T812ISA_GET_CALLBACK cb)
```

Beschreibung
Registriert eine callback-Funktion für die Leseoperation vom C812er Controller über die 'ISA'-Schnittstelle.

Parameter
`cb` – Pointer auf callback-Funktion oder `NULL` um eine Registrierung rückgängig zu machen.

Funktion: `msRegister_C812ISA_Get`

Signatur
```
void WINAPI msRegister_C812ISA_Put(T812ISA_PUT_CALLBACK cb)
```

Beschreibung
Registriert eine callback-Funktion für die Schreiboperation an den C812er Controller über die 'ISA'-Schnittstelle.

Parameter
`cb` – Pointer auf callback-Funktion oder `NULL` um eine Registrierung rückgängig zu machen.

Funktion: `msRegister_C832_Get`

Signatur
```
void WINAPI msRegister_C832_Get(T832_GET_CALLBACK cb)
```

Beschreibung
: Registriert eine callback-Funktion für die Leseoperation vom C832er Controller.

Parameter
: `cb` – Pointer auf callback-Funktion oder `NULL` um eine Registrierung rückgängig zu machen.

Funktion: `msRegister_C832_Put`

Signatur
: `void WINAPI msRegister_C832_Put(T832_GET_CALLBACK cb)`

Beschreibung
: Registriert eine callback-Funktion für die Schreiboperation an den C832er Controller.

Parameter
: `cb` – Pointer auf callback-Funktion oder `NULL` um eine Registrierung rückgängig zu machen.

3.3.4 Die Änderungen im Anwendungsrahmen

Entsprechend der oben beschriebenen Komponentenstruktur muss die Anwendung mindestens die Initialisierung der Simulationskomponente übernehmen und deren Verbindung zur Motorenkomponente herstellen. Die konkrete Implementation der Motorensimulation bietet zusätzlich die Möglichkeit, eine Textbox mit Statusinformationen der simulierten Motoren zu füllen. Um dies nutzen zu können, muss die Anwendung ein Fenster mit einer solchen Textbox implementieren und dieses der Simulation bei der Initialisierung zur Verfügung stellen.

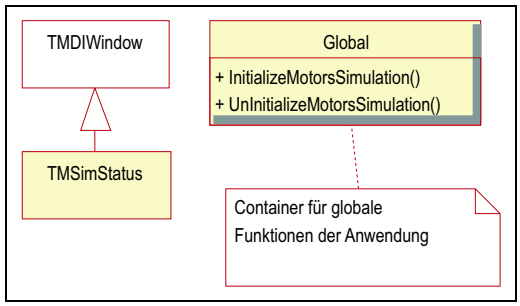


Abbildung 3.7: Die Erweiterungen im Anwendungsrahmen

Zu diesem Zweck ist die Anwendung um eine sehr einfache Klasse namens `TMSimStatus` für das Statusfenster und zwei globale Funktionen für die Initialisierung und die Beendigung der Simulationskomponente erweitert worden (s. Abb. 3.7).

Diese Erweiterungen befinden sich alle in dem Modul `msimstat.[h|cpp]`, das einen Umfang von 20 bzw. 170 LOC hat. Als letztes mussten die Initialisierungsfunktionen in den, in `m_main.cpp` implementierten, Initialisierungsablauf der Anwendung integriert werden, was mit dem Einfügen von zwei Zeilen in diesem Modul durchgeführt wurde.

Im folgenden werden die Erweiterungen aus `msimstat.[h|cpp]` beschrieben.

Funktion: `InitializeMotorsSimulation`

Signatur

```
void InitializeMotorsSimulation(const char* ini_file)
```

Beschreibung

Diese Funktion initialisiert und konfiguriert die Simulationskomponente entsprechend den Konfigurationsparametern der Sektion `MOTORSIM` (s. S. 71).

Parameter

`ini_file` – Dateiname der Konfigurationsdatei, in der die Konfigurationsparameter gesucht werden und mit deren Information auch die Simulationskomponente versorgt wird. Normalerweise ist dies auch die Konfigurationsdatei der Anwendung, deren Name, nach Initialisierung der `splib`-Komponente, `GetCFile()` (bzw. in der neuen Version `GetHWFile()`) liefert.

Näheres zu den Konfigurationsparametern, die die Simulation steuern ist weiter unten zu finden.

Die Funktion muss *vor* der Initialisierung der Motorenkomponente durch `mlInitializeMotorsDLL` aufgerufen werden.

Funktion: `UnInitializeMotorsSimulation`

Signatur

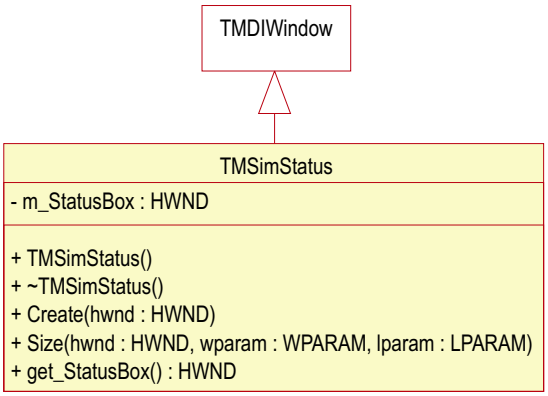
```
void UnInitializeMotorsSimulation()
```

Beschreibung

Diese Funktion dient der Beendigung der Simulation und der Löschung eventuell bei der Initialisierung angelegter Objekte.

Die Funktion muss *nach* dem letzten Aufruf einer Funktion der Motorenkomponente aufgerufen werden.

Klasse: `TMSimStatus`



Beschreibung

Die Klasse implementiert das Statusfenster für die Motorensimulation als *MDI child window*.

Die verwendete Motorensimulation bietet die Möglichkeit, eine Textbox mit Statusinformationen der simulierten Motoren zu füllen. Das Simulationsfenster besteht im Moment nur aus eben solch einer Textbox.

Für den Inhalt des Statusfensters ist die Simulationskomponente zuständig. Zur dessen Erläuterung siehe Abschnitt 3.4.6.

Methoden

Methode `get_StatusBox`

Signatur

`HWND get_StatusBox()`

Rückgabewert

Liefert das Handle der Textbox des Fensters, das in der member-Variablen `m_StatusBox` gespeichert ist.

3.4 Realisierung der Simulationskomponente

3.4.1 Vorüberlegungen

Umgebungsaspekte

Als erstes soll die Frage geklärt werden: Welche Umgebungsaspekte sind für die Software sichtbar bzw. beeinflussen ihr Verhalten?

Ort-Zeit-Verhalten Der primäre Umgebungsaspekt ist das Ort-Zeit-Verhalten der Motorenhardware. Zu einem gegebenen Zeitpunkt liefert die Motorenhardware eine Positionsangabe (Ist-Position). Die Software nimmt Einfluss auf die Position, indem sie eine Ziel- bzw. Soll-Position vorgibt. Solange die Ist-Position von der Soll-Position abweicht ist der Motor in Bewegung. Die Software beeinflusst diese Bewegung, indem sie eine Beschleunigung bzw. Dämpfung und eine maximale Geschwindigkeit festlegt.

Die vom Controller gelieferte Position hat ein Koordinatensystem mit willkürlich gesetztem und veränderbarem Nullpunkt. D.h. diese Position hat keinen Bezug zur tatsächlichen Stellung des Motors innerhalb des verfahrbaren Bereichs. Diese Stellung, und damit das Verhältnis zu den Endlagen- und Indexschaltern, beeinflusst aber das Softwareverhalten und ist in besonderer Weise sicherheitsrelevant und muss daher von der Simulation auch zur Verfügung gestellt werden. Die Information über diese Stellung erhält der Controller i.d.R. über spezielle Eingangssignale, die von den entsprechenden Schaltern bedient werden. Sie und werden der Software über Statusregister zur Verfügung gestellt.

Die Simulation wird, um beides darstellen zu können, zwei Positionen verwalten:

1. Die *Encoder-Position*, die Position die der Controller verwaltet und der Software auf Anfrage reportiert
   (eine interne Motorposition der Verhaltensspezifikation).

   Das Koordinatensystem dieser Positionen hat einen willkürlich gesetzten und veränderbaren Nullpunkt.

2. Die *physikalische Position*, die Position, die die Stellung des Motors in dem verfahrbaren Bereich widerspiegelt.

   Das Koordinatensystem dieser Positionen hat seinen festen Nullpunkt *per definitionem* an der linken Hardware-Schranke.

Ein Teilaspekt des Ort-Zeit-Verhaltens ist das Problem des Motorspiels. Auch dieses Spiel soll von der Simulation nachgebildet werden. Genauere Erläuterungen zum Motorsiel sind in der Verhaltensspezifikation zu finden.

Für die Simulation des Ort-Zeit-Verhaltens müssen also folgende Parameter verwaltet werden:

- Encoder-Ist-Position
- Encoder-Soll-Position
- physikalische Position
- aktuelle Geschwindigkeit
- Beschleunigung
- Bewegungsrichtung

E/A-Interaktion Der zweite Umgebungsaspekt ist der Bereich der Interaktion bzw. Kommunikation zwischen Software und Motorcontroller. Hierbei verhält sich der Controller ähnlich einem Zustandsautomaten, der nur unter bestimmten Bedingung Lese- oder Schreibaktionen erlaubt, die auch nur in bestimmten protokollarischen Abläufen zu gültigen Befehlen bzw. Ergebnissen führen.

Dieser Aspekt ist aber nur schwer zu verallgemeinern und soll daher bei den Abschnitten zu den einzelnen Controllern erläutert werden. Als Beispiel aber kann folgendes dienen: ein Controller akzeptierte keine Eingaben, wenn er im *busy*-Zustand ist, die Software muss vor einer Schreiboperation diesen Zustand testen, die Anzahl der notwendigen Tests ist nicht determiniert; um testen zu können, ob die Software auf solche Situationen vorbereitet ist, sollte dieser Aspekt der E/A-Interaktion simulierbar sein.

Treiber

Als nächstes steht die Frage: Wie soll die Simulation getrieben werden?

Eine erste Möglichkeit ist eine *sich selbst treibende Simulation*. Das soll heißen: die Simulation läuft parallel neben der Anwendung und aktualisiert unabhängig von dieser, sofern sie „in Bewegung“ ist, ihren Zustand, der verwendet wird um die Anforderungen der Anwendung zu bedienen; „ein Motor bewegt sich, unabhängig davon, was sie Software tut“. Diese Lösung bringt aber einige technische Probleme mit sich.

1. Die aktuelle Version des XCTL-Systems ist eine 16bit-Anwendung, geschrieben für Windows 3.1. Unter diesem Betriebssystem gibt es keine ausreichenden Konzepte für nebenläufige Programmierung wie z.B. *threads*.
2. Das aktuelle XCTL-System ist (auch wegen erstens) nicht unter der Berücksichtigung nebenläufiger Konzepte geschrieben worden und auch nach einer Portierung nach Win32 wird es schwer sein solche einzubringen.
3. Nebenläufige Programme sind deutlich komplizierter und damit fehleranfälliger.

Eine Alternative ist deshalb eine *Anforderungs-getriebene Simulation*. D.h. die Simulation bestimmt ihren Zustand nur dann, wenn die Anwendung eine Anforderung stellt. Das scheint zuerst problematisch in Bezug auf das Ort-Zeit-Verhalten. Aber wenn man die oben genannten Parameter und zusätzlich die Zeit, zu der diese galten, zur Verfügung hat, lassen sich daraus zu einem späteren Zeitpunkt die aktuellen Parameter berechnen. (zu Details s. S. 43)

Mit dieser Variante besteht zwar das Problem, dass ein Motor sich nur dann „bewegt“, wenn die Anwendung ihn nach seinem Zustand fragt bzw. auf diesen

Einfluss nimmt. Da aber das regelmäßige Abfragen des Zustandes ein wesentlicher Bestandteil einer Motoren-überwachenden Software ist, kann das in Kauf genommen werden.

3.4.2 Implementationsübersicht

Die Simulationskomponente simuliert einen C812er Controller, der bis zu vier Motoren verwalten kann und einen C832er Controller, der einen oder zwei Motoren steuert. Dementsprechend besteht die Komponente aus zwei Paketen, jeweils eines für einen Controllertyp. Jedes Paket implementiert eine Klasse die für den Controller steht (`MC_812`, `MC_832`) und eine Klasse für die Motoren (`MC_812::Motor`, `MC_832::Motor`).

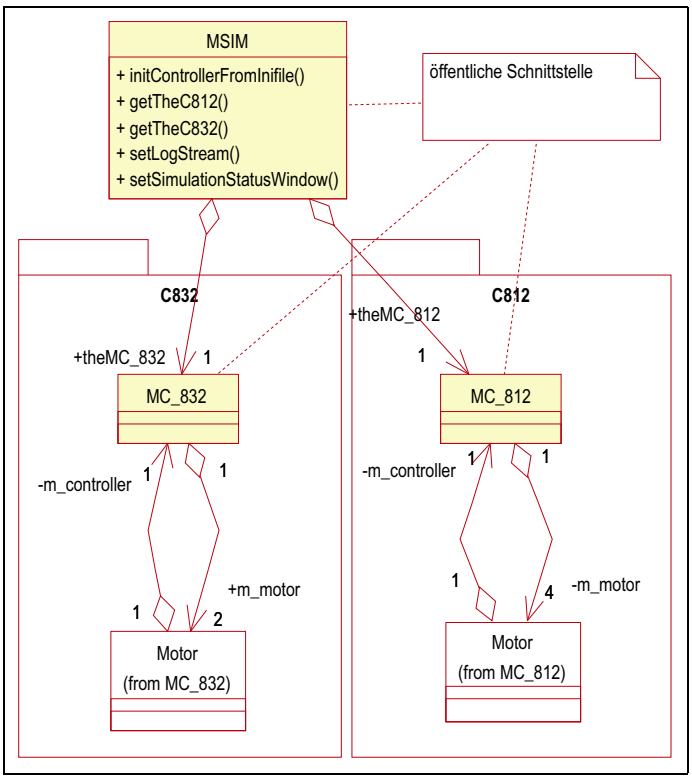


Abbildung 3.8: Übersicht über die Pakete der Simulationskomponente

Die beiden Pakete unterscheiden sich im Wesentlichen nur in der Implementation der E/A-Kommunikation, da der C812er Controller über lesbare, durch eine formale Grammatik beschreibbare Zeichenketten gesteuert wird, der C832er aber über Registerinhalte. Außerdem differiert der Funktionsumfang, den die Controller anbieten, etwas, was sich dann auch in der Implementation der Simulation niederschlägt.

Die Gemeinsamkeiten der entstandenen Implementationen legen nahe, dass es für den Fall weiterer zu simulierender Controller sinnvoll sein kann, eine allgemeinere Klassenstruktur zu suchen, in der größere Teile für die Implementation neuer Controller wiederverwendet werden könnten. Im Rahmen dieser Arbeit

aber, in der die Simulation nur mittelbar Thema ist, konnte dieser Aufwand nicht mehr aufgebracht werden.

Neben den beiden Controller-Paketen implementiert die Simulationskomponente noch einige globale Funktionen, die in Abb. 3.8 in der *Pseudoklasse* `MSIM` zusammengefasst wurden, die die globalen Funktionen der Komponente umfasst und damit für die Komponente selbst steht. Die Komponente hält zwei globale Objekte (`theMC_812`, `theMC_832`), eines für jeden Controller, die sie über entsprechende `get`-Methoden zur Verfügung stellt.

Als letztes implementiert die Simulationskomponente noch einige Callback-Funktionen, entsprechend den Definitionen in der modifizierten Motorenkomponente, für die Kommunikation mit der Motorenkomponente (s. Abschnitt 3.4.5).

3.4.3 Paket C812

Das Paket C812 implementiert die Simulation des C812er Controllers. Die Klassenstruktur dieses Paketes ist sehr einfach (s. Abb. 3.9). Es gibt eine Klasse für den Controller (`MC_812`) und eine Klasse für die von diesem verwalteten Motoren (`MC_812::Motor`). Ein C812er Controller kann bis zu 4 Motoren steuern. Daher besitzt die Controllerklasse Verweise auf 4 Objekte der Motorenklasse.

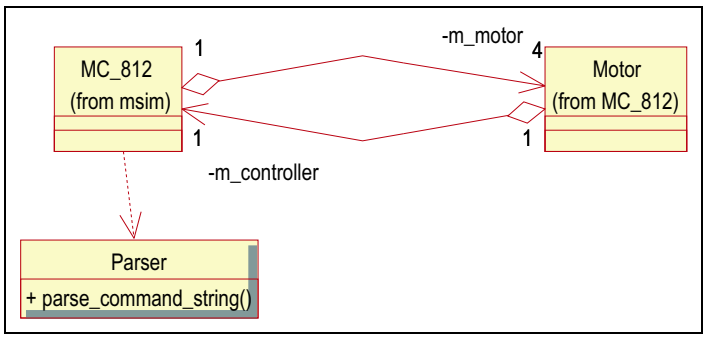
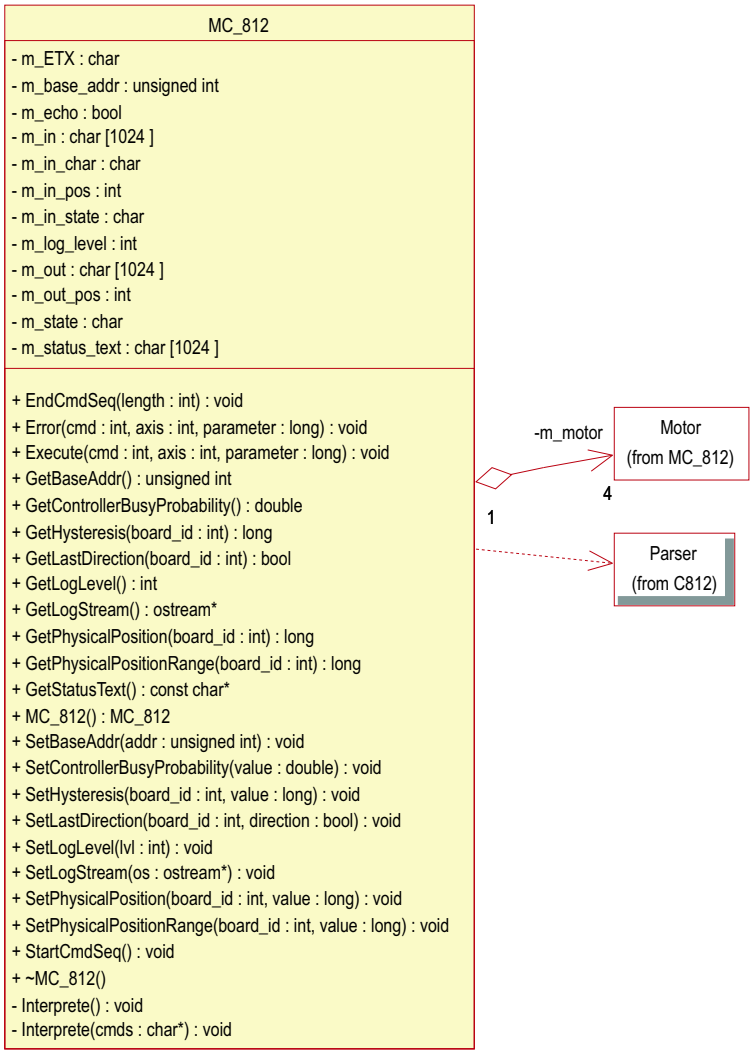


Abbildung 3.9: Klassen des Pakets C812 (Übersicht)

Die Controllerklasse `MC_812` ist für den Umgebungsaspekt E/A-Interaktion, also die Kommunikation mit der hardwarenutzenden Software, verantwortlich und die Motorenklasse `MC_812::Motor` realisiert den Umgebungsaspekt Ort-Zeit-Verhalten.

Außerdem enthält das Paket ein in Abb. 3.9 als *utility class* dargestellten Parser. Dieser ist mit den Werkzeugen lex und yacc, genauer flex und bison, generiert worden. Der Parser ist keine echte Klasse, sondern ein abstraktes Datenobjekt (Balzert, I, S. 643) mit genau einer Zugriffsoperation. (Zu spät für diese Arbeit, habe ich lex- und yacc-Versionen gefunden, die auch OO-Parser generieren.)

Klasse: MC_812



Beschreibung

Klasse für den C812er Controller.

Neben den im Diagramm aufgelisteten Hauptmethoden gibt es für jedes Motorkommando eine private Methode und einige Hilfsmethoden, die im Quelltext kommentiert sind.

Ein Objekt dieser Klasse verwaltet genau vier Objekte der Klasse `MC_812::Motor`, die für die vom Controller gesteuerten Motoren stehen.

Attribute

Attribut `m_ETX`

Beschreibung

Aktuelles *end of text*-Zeichen. Kann durch Motorkommando `CM` gesetzt werden. Voreinstellung ist 0x03.

Attribut `m_base_addr`

Beschreibung
Basisadresse des *dual port RAM* (s. Eigenschaft `BaseAddr`).

Attribut `m_echo`

Beschreibung
Flag, dass festlegt ob der Motor im Echomode ist oder nicht.

Attribut `m_in`

Beschreibung
Inputpuffer. Speichert den gerade geschriebenen Kommandostring.

Attribut `m_in_pos`

Beschreibung
Schreibposition im Inputpuffer.

Attribut `m_in_char`

Beschreibung
Zuletzt an Mailboxaddresse 1 geschriebener Wert.

Attribut `m_in_state`

Beschreibung
Mailbox-Flag. `true`, falls letzte Schreiboperation auf Mailboxaddresse 1 erfolgte.

Attribut `m_out`

Beschreibung
Ausgabepuffer. Speichert die Ausgabe des letzten Kommandostrings.

Attribut `m_out_pos`

Beschreibung
Lesepostion im Ausgabepuffer.

Attribut `m_state`

Beschreibung
Statusbyte des Controllers.

Attribut `m_status_text`

Beschreibung
Statustext der den Zustand der Motoren beschreibt (s. Eigenschaft `StatusText`).

Attribut `m_log_level`

Beschreibung
Legt Protokollumfang fest (s. Eigenschaft `LogLevel`).

Methoden

Methode `Put`

Signatur
`void Put( char* address, char put )`

Beschreibung
Methode für Schreiboperation auf Controller. Je nach Adresse wird der Inputpuffer gefüllt, oder ein *direct access*-Zugriff durchgeführt. Wird ein *carriage return* in den Inputpuffer geschrieben, wird die Kommandointerpretation angestoßen (s. `Interprete()`).

Parameter
`address` – Speicheradresse, an die geschrieben wird
`put` – der geschriebene Wert

Methode `Get`

Signatur
`char Get( char* addr )`

Beschreibung
Methode für Leseoperation auf Controller. Je nach Adresse wird ein Zeichen aus dem Outputpuffer zurückgegeben, oder ein Byte aus einem *direct access*-Register geliefert.

Parameter
`address` – Speicheradresse, von der gelesen wird

Rückgabewert
der gelesene Wert

Methoden zur Kommandoauswertung Die folgenden Methoden dienen der Auswertung eines Kommadostrings. `Interprete` ist die Einstiegsmethode, die den Parser aufruft, der wiederum die Methoden `Execute`, `Error` und `EndCmdSeq` rückruft, weshalb diese Methoden öffentlich sein müssen. Zum Zusammenspiel der Methoden s. Abb. 3.13.

Methode `Interprete`

Signatur
`void Interprete( )`

Beschreibung
Wertet den aktuellen Inputpuffer mit Hilfe des Parsers aus.

Methode `Execute`

Signatur
`void Execute( int cmd, int axis, long parameter )`

Beschreibung
Methode, die vom Parser aufgerufen wird, wenn im Inputpuffer ein vollständiges Kommando gelesen wurde. Die Methode delegiert den Aufruf entsprechend dem ersten Parameter an eine der Kommandomethoden (s.u.).

Parameter
`cmd` – Kennung des auszuführenden Kommandos

`axis` – Kennung des Motors für den das Kommando bestimmt ist; erlaubte Werte sind: 0, falls keine Achsenangabe im Kommando vor lag, 1 bis 4 sonst.

`parameter` – Parameter des Kommandos

Methode `Error`

Signatur
```
void Error( int cmd, int axis, long parameter )
```
Beschreibung
Methode, die vom Parser aufgerufen wird, wenn beim Parsen des Inputpuffers ein Fehler aufgetreten ist, also die Kommandosequenz lexikalisch oder grammatisch falsch ist. Das Auftreten eines Fehlers wird protokolliert.

Parameter
s. `Execute`

Methode `EndCmdSeq`

Signatur
```
void EndCmdSeq( int length )
```
Beschreibung
Methode die vom Parser aufgerufen wird, wenn das Ende der Kommandosequenz im Inputpuffer gefunden wurde.

Parameter
`length` – Anzahl der Kommandos in der zuletzt abgearbeiteten Kommandosequenz.

Kommandomethoden

Signatur
```
void EF( )
void CN( long param )
void AB( int axis )
void MA( int axis, long param )
...
```
Beschreibung
Die Controllerklasse implementiert für jedes Motorkommando eine gleichnamige Methode. Eine Übersicht über die Kommandos bietet der Anhang B.1. Dort findet sich in Abb. B.5 auch eine Auswahl der relevanten Kommandos.

Parameter
`axis` – Parameter bei Methoden für Kommandos, die sich auf einen Motor beziehen. Werte von 1 bis 4 geben den Zielmotor des Kommandos an. Ein Wert von 0 gibt an, dass eine optionale Achsenangabe weggelassen wurde und sich das Kommando auf alle Motoren bezieht.

`param` – Hiermit wird der Parameter des Motorkommandos übergeben.

Eigenschaften

Eigenschaft `BaseAddr`

Signatur
```
void SetBaseAddr( unsigned )
unsigned GetBaseAddr( )
```

Beschreibung
Basisadresse des für die Kommunikation verwendeten *dual port RAM*.

Eigenschaft `StatusText`

Signatur
```
const char* GetStatusText( )
```
Beschreibung
Text der den Zustand der 4 simulierten Motoren beschreibt. Ist für die Verwendung im Zusammenhang mit dem Statusfenster vorgesehen.

Eigenschaft `LogLevel`

Signatur
```
void SetLogLevel( int value )
int  GetLogLevel( )
```
Beschreibung
Zahl die den Umfang des Protokolls festlegt. Bei Werten größer als 0 werden Informationen über die abgearbeiteten Kommandos und den mit den Rückgabewerten gefüllten Outputpuffer sowie Fehler protokolliert.
Beispiel
```
<cmd-seq>
  <cmd c='C812ISA'>set torque 2, 110</cmd>
  <cmd c='C812ISA'>set gain 2, 65236</cmd>
  <error c='C812ISA'>SG: gain outside valid
                          value range [1..254]</error>
  <cmd c='C812ISA'>set dynamic gain 2, 3000</cmd>
  <out>{ETX}</out>
</cmd-seq>
```

Eigenschaft `LogStream`

Signatur
```
void SetLogStream( ostream* )
ostream* GetLogStream( )
```
Beschreibung
Outputstrom auf den die Prokolleinträge geschrieben werden soll.

Motoreigenschaften Die folgenden Eigenschaften sind keine Eigenschaften des Controllers sondern eines der verwalteten Motoren. Daher haben alle zugehörigen `get/set`-Methoden den Parameter `board_id`; eine Zahl zwischen 1 und 4, die einen der Motoren identifiziert.

Eigenschaft `Hysteresis`

Signatur
```
void SetHysteresis( int board_id, long value )
long GetHysteresis( int board_id )
```
Beschreibung
Spiel des zu simulierenden Motors.

Eigenschaft `LastDirection`

Signatur

```
void SetLastDirection( int board_id, bool value )
bool GetLastDirection( int board_id )
```

Beschreibung

Fahrtrichtung des Motors. Notwendig um die Simulation des Motorspiels realisieren zu können.

Eigenschaft `PhysicalPositionRange`

Signatur

```
void SetPhysicalPositionRange( int board_id, long value )
long GetPhysicalPositionRange( int board_id )
```

Beschreibung

Größe des verfahrbaren Bereichs, d.h. der Abstand zwischen den zu simulierenden Endlagenschaltern in Encoderschritten.

Eigenschaft `PhysicalPosition`

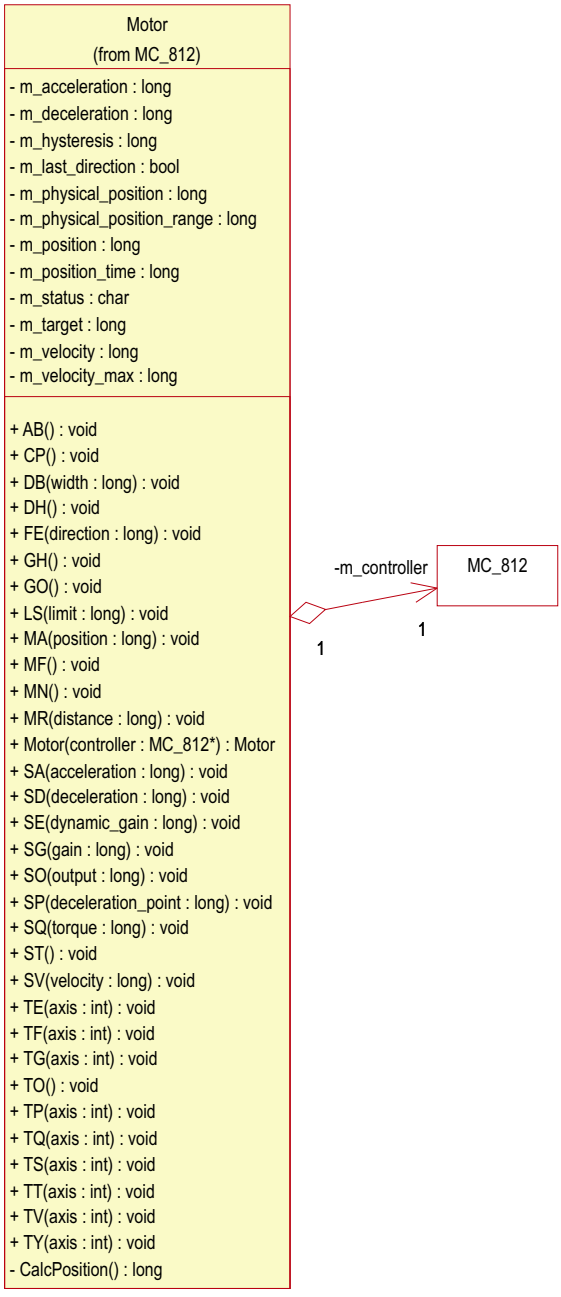
Signatur

```
void SetPhysicalPosition( int board_id, long value )
long GetPhysicalPosition( int board_id )
```

Beschreibung

Stellung des simulierten Motors innerhalb des verfahrbaren Bereichs; genauer: Abstand der Motorposition vom linken Endlagenschalter.

Klasse: `MC_812::Motor`



Beschreibung

Private Klasse für einen vom C812er Controller verwalteten Motor.

Die Klasse implementiert für jedes Motorkommando eine Methode; zur Einbindung dieser Methoden in die Kommandoverarbeitung s. Abb. 3.13. Eine Liste der Kommandos mit kurzen Erläuterungen ist in Abb. B.5 zu finden. Das Ort-Zeit-Verhalten wird im Wesentlichen durch die private Methode `CalcPosition` realisiert.

Außerdem besitzt die Klasse noch einige private Hilfsmethoden, die im Quelltext kommentiert sind, sowie einige Attribute die zur Speicherung von Motorparametern dienen, die zwar über Kommandos gesetzt werden können, aber von der aktuellen Implementation nicht verwendet werden, wie z.B. Parameter zur Steuerung des Motorstroms.

Attribute

Attribut `m_acceleration`

Beschreibung
Motorparameter Beschleunigung. Wird durch `SA`-Kommando gesetzt.

Attribut `m_deceleration`

Beschreibung
Motorparameter Entschleunigung. Wird durch `SD`-Kommando gesetzt.

Attribut `m_hystersis`

Beschreibung
Simuliertes Motorspiel (s. Eigenschaft `Hysteresis` der Klasse `MC_812`).

Attribut `m_last_direction`

Beschreibung
Bewegungsrichtung der letzten bzw. aktuellen Bewegung.

Attribut `m_position`

Beschreibung
Simulierte Motorposition zum in `m_position_time` gespeicherten Zeitpunkt (s. `CalcPosition`).

Attribut `m_position_time`

Beschreibung
Zeitpunkt der letzten Positionsbestimmung in Millisekunden seit Initialisierung (s. `CalcPosition`).

Attribut `m_physical_position`

Beschreibung
Simulierte physikalische Motorposition zum in `m_position_time` gespeicherten Zeitpunkt (s. `CalcPosition`).

Attribut `m_physical_position_range`

Beschreibung
Größe des verfahrbaren Bereichs (s. Eigenschaft `PhysicalPositionRange` der Klasse `MC_812`).

Attribut `m_velocity`

Beschreibung
Simulierte Motorgeschwindigkeit zum in `m_position_time` gespeicherten Zeitpunkt (s. `CalcPosition`).

Attribut `m_max_velocity`

Beschreibung
Motorparameter Maximale Geschwindigkeit. Wird durch `SV`-Kommando gesetzt.

Attribut `m_target`

Beschreibung
Zielposition.

Attribut `m_status`

Beschreibung
Statusregister des Motors, s. Abb. B.4.

Methoden

Kommandomethoden Für jedes Motorkommando ist eine gleichnamige Methode implementiert. Die Methoden überprüfen und speichern ihre Parameter und protokollieren eventuelle Wertebereichsverletzungen. Dann führen sie, falls notwendig, eine Positionsbestimmung durch (s. Methode `CalcPosition`). Bewegungskommandos wie `MA` oder `AB` setzten die Zielposition und lösen damit die „Bewegung“ aus. Abschließend wird das Statusregister aktualisiert.

Positionsbestimmung

Methode `CalcPosition`

Signatur
`long CalcPosition()`

Beschreibung
Die Methode bestimmt die aktuelle Position und die aktuelle Geschwindigkeit. Sie verwendet dazu die Position und Geschwindigkeit, die bei der letzten Bestimmung gespeichert worden sind, und die Zeitdifferenz, die seit der letzten Bestimmung vergangen ist, sowie die Parameter „maximale Geschwindigkeit“, „Beschleunigung“ und „Entschleunigung“. Außerdem testet sie, ob eventuell Hardwareschranken erreicht wurden.

Die durchgeführten Berechnungen sind in Abb. 3.11 und 3.12 schematisch dargestellt. In der ersten Abbildung wird die Berechnung von Istposition und Istgeschwindigkeit dargestellt. Dabei wird nach drei Phasen der Bewegung unterschieden (s. Abb. 3.10), der Beschleunigungsphase, der Phase mit konstanter Geschwindigkeit und der Verlangsamungsphase, in denen unterschiedliche Gleichungen bzw. Parameter zu verwenden sind. Wird zwischen der letzten und der aktuellen Positionsbestimmung ein Phasenwechsel festgestellt, so wird die neue Position (bzw. Geschwindigkeit) aus dem Mittel der Positionen bestimmt, die errechnet werden können, wenn 1. der Phasenwechsel bei der vorherigen Position bzw. 2. wenn der Phasenwechsel bei der aktuellen Position läge.

In der zweiten Abbildung wird der Test auf das Erreichen der Zielposition bzw. das einer Hardwareschranke dargestellt. Das Ziel gilt als erreicht, wenn die neu berechnete Position auf oder hinter der Zielposition liegt, bzw. die neu berechnete Geschwindigkeit kleiner gleich Null ist.

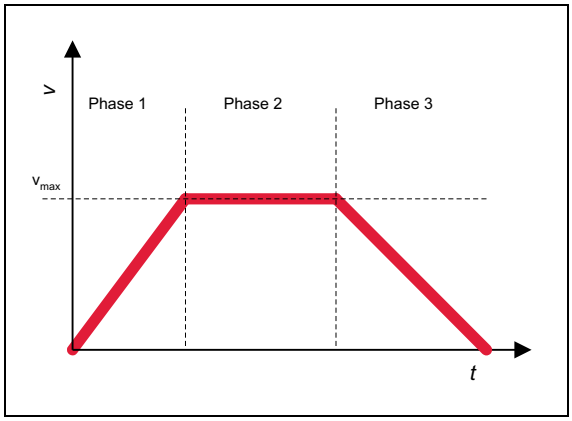


Abbildung 3.10: 3 Phasen der Motorbewegung

Verlässt die neu berechnete Position den verfahrbaren Bereich, d.h. wurde eine Hardwareschranke getroffen, wird eine neue Zielposition eingestellt, die um den im Parameter `RemoveLimit` einstellbaren Abstand vor der Hardwareschranke liegt.

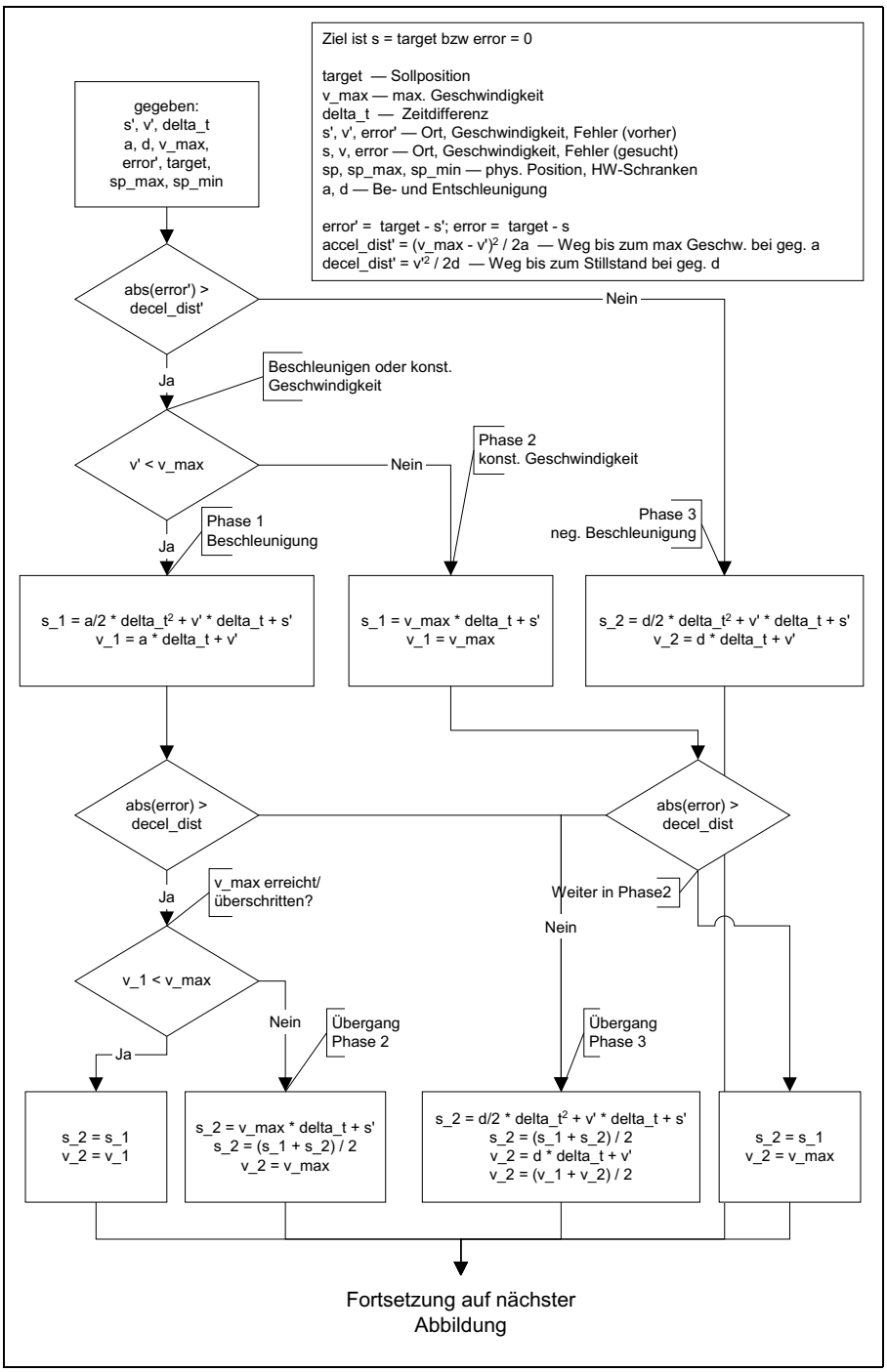


Abbildung 3.11: Berechnung der Istposition (Teil 1)

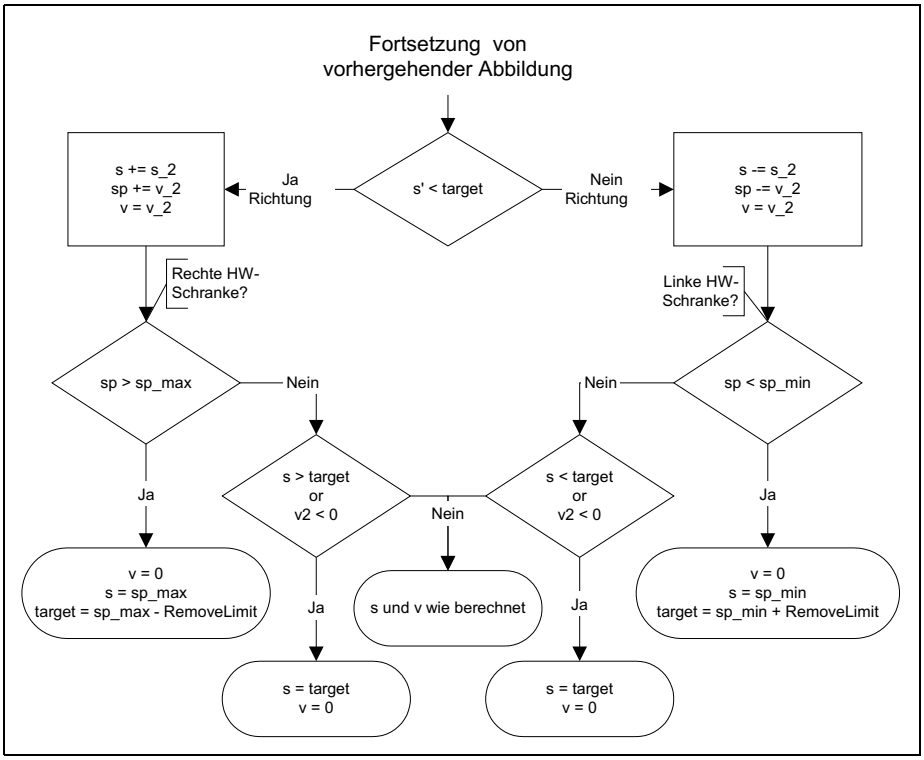


Abbildung 3.12: Berechnung der Istposition (Teil 2)

Das Sequenzdiagramm in Abb. 3.13 veranschaulicht die Grundzüge des Zusammenspiels der Klassen im Paket C812 bei der Interpretation eines Kommandostrings.

Eine Clientsoftware (im Beispiel die Simulationskomponente) sendet durch wiederholten Aufruf der `Put`-Methode zeichenweise einen Kommandostring (für Details s. Anhang B.1). Ein Kommando ist stets durch ein *carriage return*-Zeichen abzuschließen. Dieses löst die Verarbeitung des Kommandostrings aus. In dieser Implementation heisst das, dass der String an den Parser weitergereicht wird, der für jedes gefundenen Kommando die `Execute`-Methode der Controllerklasse aufruft, sowie am Ende des Strings die `EndCmdSeq`-Methode.

Die `Execute`-Methode leitet die Aufrufe an die entsprechenden Kommandomethoden weiter, z.B. das Kommando MR an die Methode `MR`. Ist das Kommando ein Motorkommando, wird von der Kommandomethode bei einem bzw. an allen Motoren die entsprechende Methode aufgerufen. Ist das Kommando ein Reportkommando, schreibt die Kommandomethode des Motors das Reportergebnis in den Ausgabepuffer.

Am Ende der Kommandoabarbeitung wird das *data available*-Bit des Statusregisters gesetzt. Den Inhalt des Ausgabepuffers kann die Clientsoftware über wiederholten Aufruf der `Get`-Methode auslesen. Das Ende des Puffers wird durch ein *end of text*-Zeichen (ETX) gekennzeichnet.

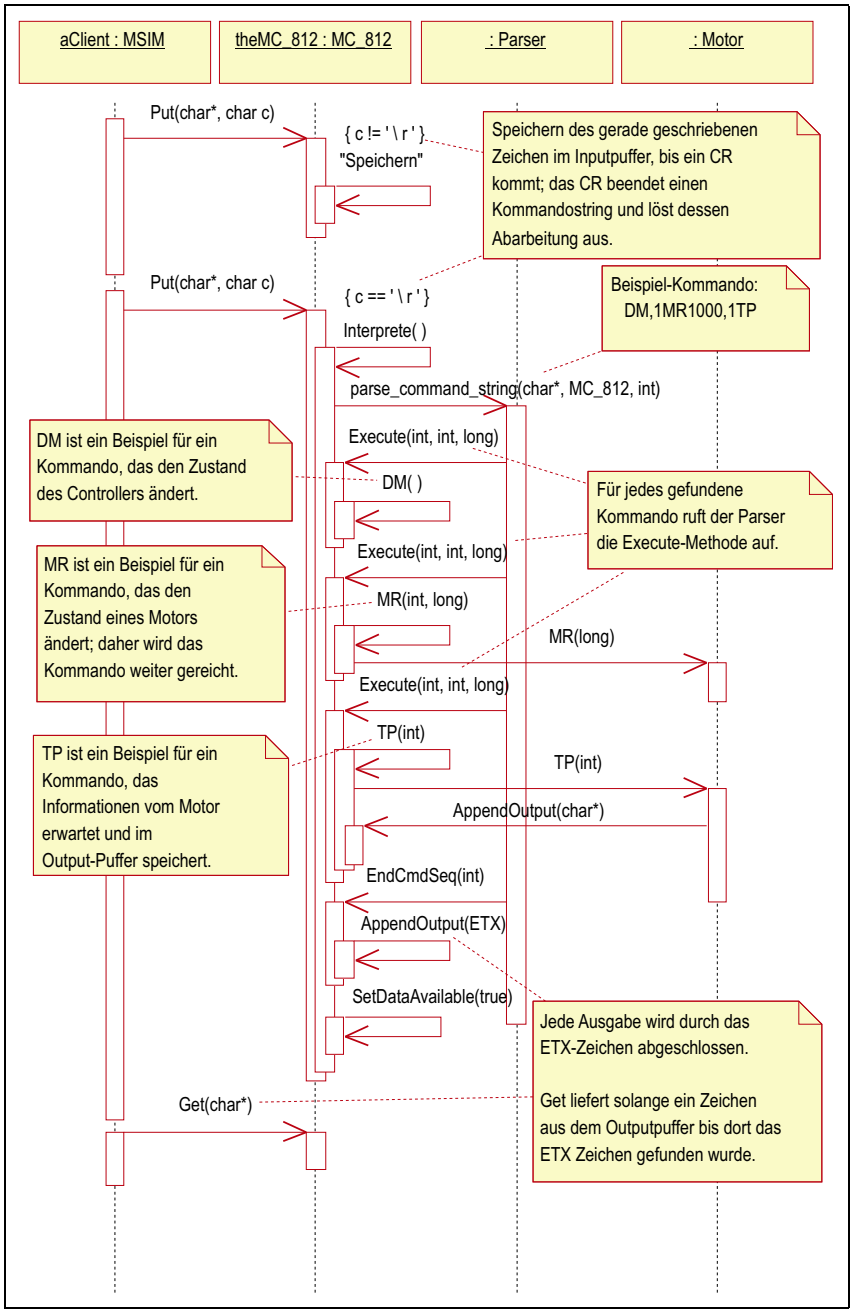


Abbildung 3.13: Sequenzdiagramm „E/A-Interaktion“

3.4.4 Paket C832

Das Paket C832 implementiert die Simulation des C832er Motorcontrollers. Wie im Paket C812 gibt es eine Klasse für den Controller (`MC_832`) sowie eine Klasse für die vom Controller verwalteten Motoren (`MC_832::Motor`).

Da die Kommunikation mit dem C832er Controller aber nicht über eine formale Sprache vermittelt wird, wie beim C812er, konnte der Zustandsautomat nicht mit Hilfe von Werkzeugen generiert werden, sondern wurde selbst implementiert. Als Hilfsmittel für die Implementation von Teilzuständen wurde die Klasse `MC_832::Command` eingeführt, von der sowohl Klassen für Reportkommandos als auch für Steuerkommandos abgeleitet sind.

Abb 3.14 bietet eine Übersicht über die Klassen des Pakets und ihre Beziehungen.

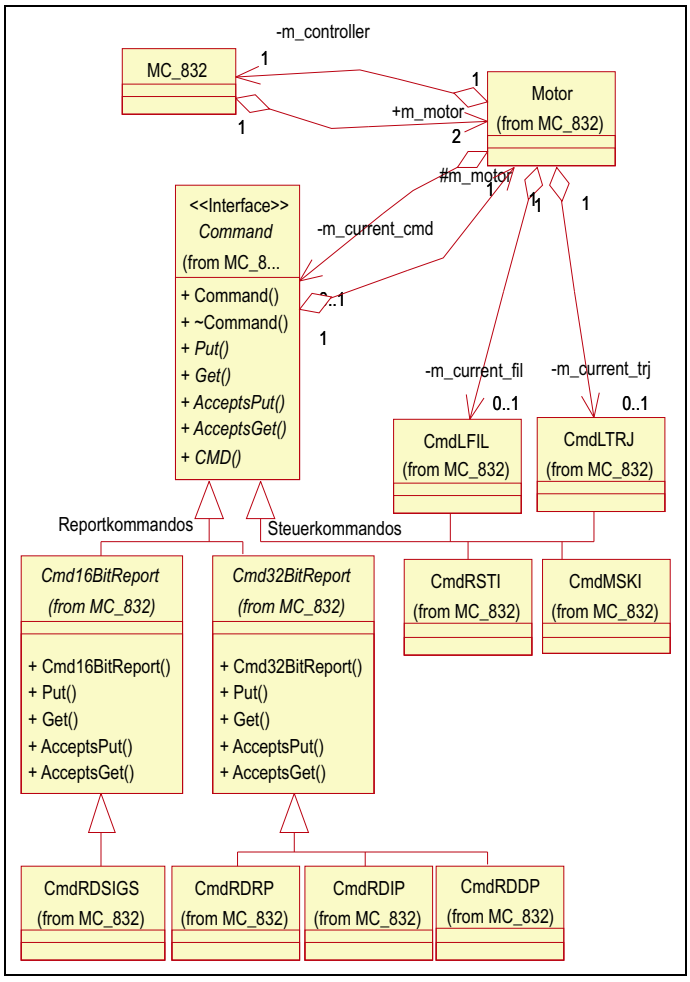


Abbildung 3.14: Klassen des Pakets C832 (Übersicht)

Ein Controllerobjekt befindet sich stets in einem von 5 Hauptzuständen, die von der Klasse `MC_812` verwaltet werden (s. dazu und zum folgenden Abb. 3.15):

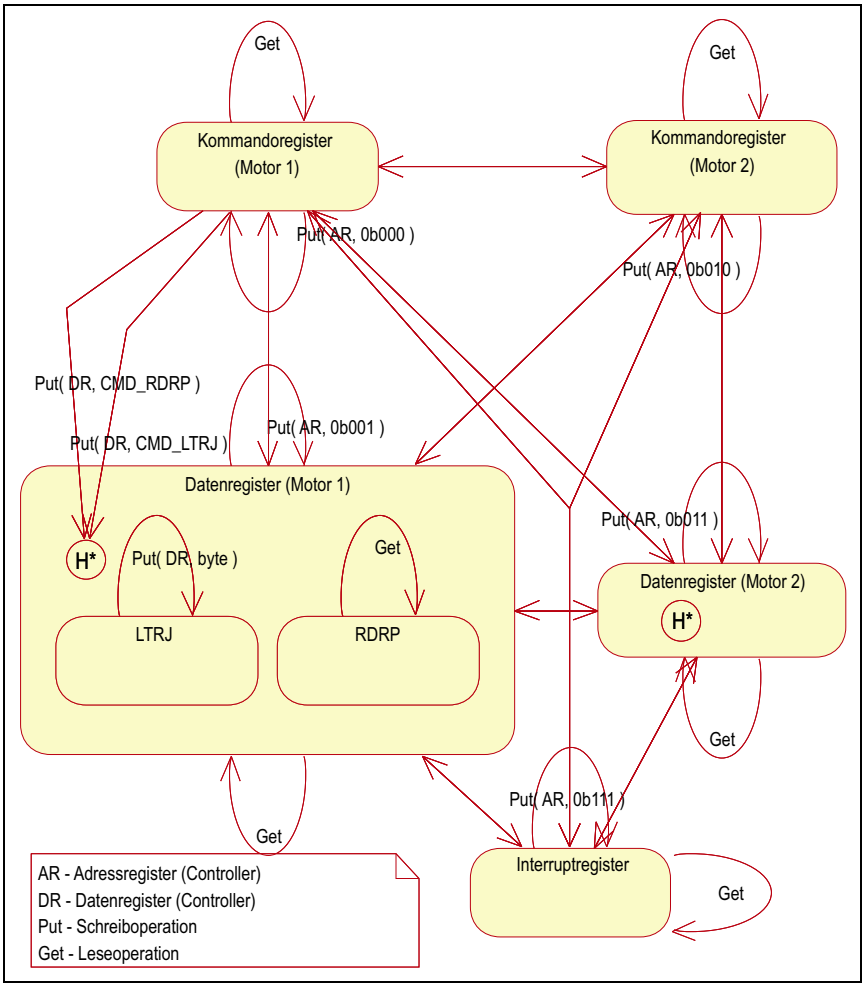


Abbildung 3.15: Die Hauptzustände des C832er Controllers

Kommandoregister (Motor 1)
: die nächste Schreiboperation ist ein Kommando für Motor 1
: die nächste Leseoperation liefert den Status des Motors 1

Datenregister (Motor 1)
: die nächste Schreiboperation ist ein Datenbyte für Motor 1
: die nächste Leseoperation liefert ein Datenbyte vom Motor 1

Kommandoregister (Motor 2)
: die nächste Schreiboperation ist ein Kommando für Motor 2
: die nächste Leseoperation liefert den Status des Motors 2

Datenregister (Motor 2)
: die nächste Schreiboperation ist ein Datenbyte für Motor 2
: die nächste Leseoperation liefert ein Datenbyte vom Motor 2

Interruptregister
: die nächste Leseoperation liefert den Inhalt des Interruptregisters

Zwischen den Hauptzuständen wird gewechselt, in dem in das Adressregister ein Byte mit entsprechend gesetzten Bits geschrieben (`Put`) wird (s. Abb. B.7). Das Lesen von Daten (`Get`) ändert an den Hauptzuständen nichts.

Die `Datenregister`-Zustände haben Unterzustände, die in Abb. 3.15 – nur für den Zustand `Datenregister (Motor 1)` und nur beispielhaft – dargestellt sind, und haben ein *Gedächtnis* (eine *History*) für ihren Zustand bzw. den ihrer Unterzustände.

Hat ein Zustand mit Unterzuständen ein *Gedächtnis*, ermöglicht dies das Zurückkehren in den zuletzt aktiven Unterzustand. Verlässt z.B. das System den Hauptzustand `Datenregister (Motor 1)` mit dem aktiven Unterzustand `LTRJ` durch einen Wechsel in den Hauptzustand `Kommandoregister (Motor 1)` (z.B. zum Auslesen der Statusregisters) und kehrt später wieder in den Zustand `Datenregister (Motor 1)` zurück, wird sofort wieder der Unterzustand `LTRJ` anstelle des (nicht dargestellten) Startzustandes eingenommen.

Das Gedächtnis eines Zustandes kann manipuliert werden. Empfängt z.B. das System im Zustand `Kommandoregister (Motor 1)` ein Byte das dem *command code* des Kommandos `LTRJ` entspricht, wird das Gedächtnis des Zustandes `Datenregister (Motor 1)` derart manipuliert, dass beim nächsten Wechsel in diesen Zustand der Unterzustand `LTRJ` aktiv ist. In diesem Unterzustand werden eine Reihe von geschrieben Bytes erwartet.

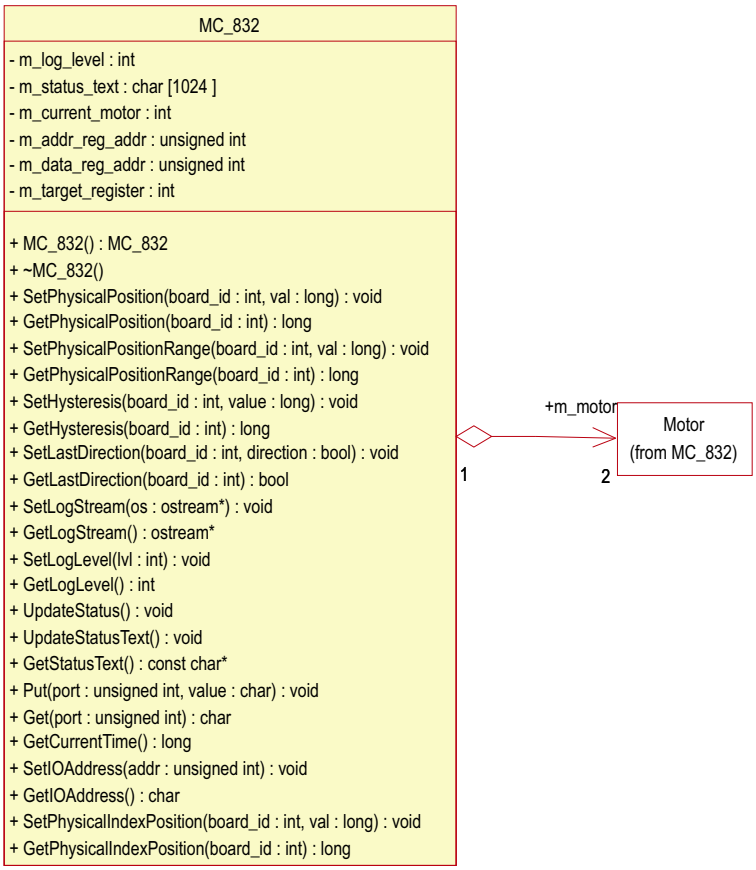
Für die Verwaltung der Unterzustände der `Datenregister`-Zustände und deren Gedächtnisses ist die Klasse `MC_812::Motor` verantwortlich. Die Unterzustände selbst und deren Unterzustände werden durch die Klasse `Command` bzw. deren Ableitungen realisiert. Das Gedächtnis wird in der `Motor`-Klasse im Wesentlichen durch die Referenz auf ein Objekt mit dem `Command`-Interface realisiert.

Die *Kommandoklassen*, d.h. die von `Command` abgeleiteten Klassen zur Implementation der genannten Unterzustände, sind alle einander sehr ähnlich. Deshalb wird in diesem Kapitel nur exemplarisch die Klasse `CmdLTRJ` beschrieben. Die anderen Kommandoklassen sind analog implementiert.

Die Abb. 3.17 stellt das Zusammenspiel der Klassen des Pakets exemplarisch an der Verarbeitung einer Kommandosequenz zum Bewegen eines Motors dar. Dazu wird von der nutzenden Komponente zuerst ein `LTRJ`-Kommando, z.B. zum Festlegen der Zielposition, gesendet, gefolgt von einem `STT`-Kommando, das die Bewegung auslöst. Zum besseren Verständnis kann das Beispiel aus B.2 herangezogen werden.

Abb. B.6 listet diejenigen Kommandos auf, die von der hier implementierten Motorensimulation realisiert werden und erläutert sie kurz.

Klasse: MC_832



Beschreibung

Klasse für die Simulation des C832er Controllers. Ein Objekt dieser Klasse verwaltet genau 2 Objekte der Klasse `MC_832::Motor`, die für die vom Controller gesteuerten Motoren stehen. Die Klasse verwaltet außerdem die Hauptzustände eines Controllers.

Attribute

Attribut `m_current_motor`

Beschreibung

Legt den Zielmotor der nächsten Schreib- oder Leseoperation fest. Bestimmt zusammen mit `m_target_register` den Hauptzustand des Controllers.

Attribut `m_target_register`

Beschreibung

Legt das Zielregister der nächsten Schreib- oder Leseoperation fest. Bestimmt zusammen mit `m_current_motor` den Hauptzustand des Controllers.

Attribut `m_addr_reg_addr`

Beschreibung
I/O-Adresse des Adressregisters (s. Eigenschaft `IOAddress`).

Attribut `m_data_reg_addr`

Beschreibung
I/O-Adresse des Datenregisters (s. Eigenschaft `IOAddress`).

Attribut `m_log_level`

Beschreibung
Legt den Protokollumfang fest (s. Eigenschaft `LogLevel`).

Attribut `m_status_text`

Beschreibung
Text der den Status der Motoren wiedergibt (s. Eigenschaft `Statustext`).

Methoden

Methode `Put`

Signatur
```
void MC_832::Put( unsigned address, byte value)
```
Beschreibung
Methode für die Schreiboperation auf dem Controller. In Abhängigkeit von den Parametern wird entweder in einen neuen Hauptzustand gewechselt oder der geschriebene Wert an den durch den aktuellen Zustand festgelegten Motor weitergeleitet.

Parameter
`address` – I/O-Adresse auf die geschrieben wird. Muss entweder dem Wert der Eigenschaft `IOAddress` (s.u.) entsprechen oder dem Wert plus 1. Das entspricht dem Schreiben ins Adressregister bzw. Datenregister.
`value` – Das zu schreibende Byte.

Methode `Get`

Signatur
```
byte MC_832::Get( unsigned address )
```
Beschreibung
Methode für die Leseoperation vom Controller.

Parameter
`address` – wie bei `Put`-Methode (s.o.).

Eigenschaften

Eigenschaft `IOAddress`

Signatur
```
void SetIOAddress( unsigned value )
unsigned GetIOAddress( )
```
Beschreibung
I/O-Adresse des simulierten Controllers. Standardwert ist 210_{16}.

Eigenschaft `StatusText`

Signatur
```
const char* GetStatusText( )
```
Beschreibung
Text der den Zustand der 2 simulierten Motoren beschreibt. Ist für die Verwendung im Zusammenhang mit dem Statusfenster vorgesehen.

Eigenschaft `LogLevel`

Signatur
```
void SetLogLevel( int value )
int  GetLogLevel( )
```
Beschreibung
Legt den Umfang des Protokolls fest. Bei Werten größer als 0 werden ...
Beispiel
`TODO`

Eigenschaft `LogStream`

Signatur
```
void SetLogStream( ostream* )
ostream* GetLogStream( )
```
Beschreibung
Ausgabestrom in den das Protokoll geschrieben werden soll.

Motoreigenschaften Die folgenden Eigenschaften sind Eigenschaften eines der zwei verwalteten Motoren. Daher haben die `set/get`-Methoden einen Parameter namens `board_id`, der entweder 0 oder 1 sein muss und damit den ersten oder den zweiten Motor identifiziert.

Eigenschaft `Hysteresis`

Signatur
```
void SetHysteresis( int board_id, long value )
long GetHysteresis( int board_id )
```
Beschreibung
Spiel des zu simulierenden Motors.

Eigenschaft `LastDirection`

Signatur
```
void SetLastDirection( int board_id, bool value )
bool GetLastDirection( int board_id )
```
Beschreibung
Fahrtrichtung des Motors. Notwendig um die Simulation des Motorspiels realisieren zu können.

Eigenschaft `PhysicalPositionRange`

Signatur
```
void SetPhysicalPositionRange( int board_id, long value )
long GetPhysicalPositionRange( int board_id )
```

Beschreibung
Größe des verfahrbaren Bereichs, d.h. der Abstand zwischen den zu simulierenden Endlagenschaltern in Encoderschritten.

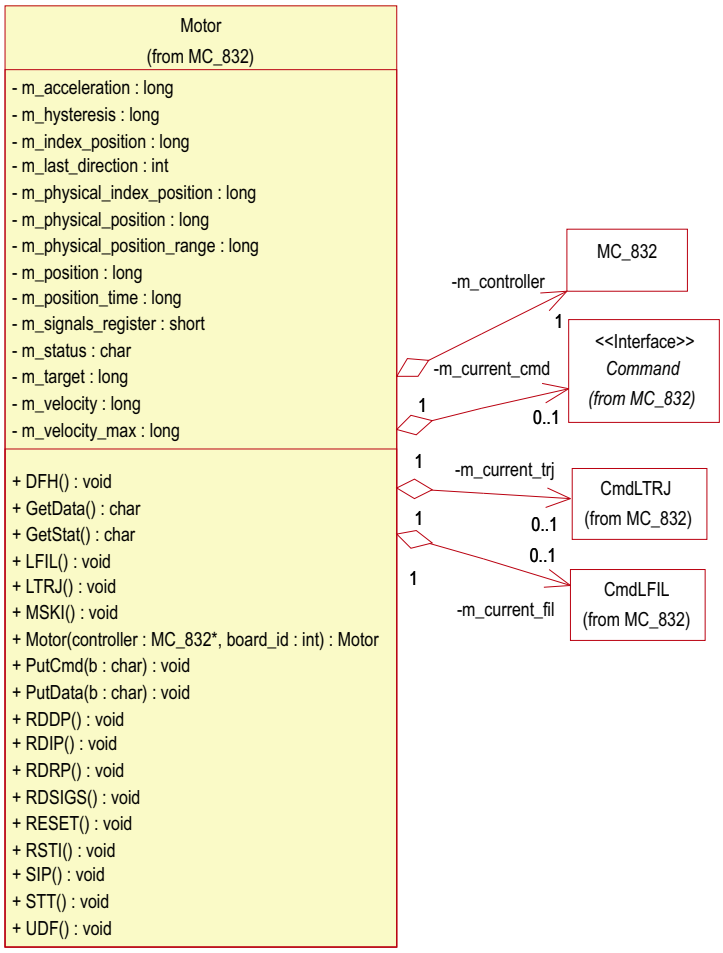
Eigenschaft `PhysicalPosition`

Signatur
```
void SetPhysicalPosition( int board_id, long value )
long GetPhysicalPosition( int board_id )
```
Beschreibung
Stellung des simulierten Motors innerhalb des verfahrbaren Bereichs; genauer: Abstand der Motorposition vom linken Endlagenschalter.

Eigenschaft `PhysicalIndexPosition`

Signatur
```
void SetPhysicalIndexPosition( int board_id, long value )
long GetPhysicalIndexPosition( int board_id )
```
Beschreibung
Abstand des Indexschalters vom linken Endlagenschalter.

Klasse: `MC_832::Motor`



Beschreibung

Klasse für die vom C832er Controller verwalteten Motoren.

Ein Objekt dieser Klasse hält eine Referenz auf ein Objekt mit dem `Command`-Interface. Dieses steht für das aktuelle Kommando und verwaltet dessen Abarbeitungszustand. Außerdem hält ein Objekt je eine Referenz auf ein `CmdLTRJ`- und ein `CmdLFIL`-Objekt, die jeweils ein vollständig abgearbeitetes Kommando darstellen und als Speicher der mit diesen Kommandos geladenen Parameter dient.

Attribute

Attribut `m_acceleration`

Beschreibung

Motorparameter Beschleunigung. Wird durch `LTRJ/STT`-Kommando gesetzt.

Attribut `m_hysteresis`

Beschreibung

Simuliertes Motorspiel (s. Eigenschaft `Hysteresis` der Klasse `MC_832`).

Attribut `m_last_direction`

Beschreibung
: Richtung der letzten bzw. der aktuellen Bewegung (s.a. Eigenschaft `LastDirection` der Klasse `MC_832`).

Attribut `m_position`

Beschreibung
: Simulierte Motorposition zum in `m_position_time` gespeicherten Zeitpunkt (s. `CalcPosition`).

Attribut `m_position_time`

Beschreibung
: Zeitpunkt der letzten Positionsbestimmung in Millisekunden seit Initialisierung (s. `CalcPosition`).

Attribut `m_physical_position`

Beschreibung
: Simulierte physikalische Motorposition zum in `m_position_time` gespeicherten Zeitpunkt (s. `CalcPosition`).

Attribut `m_physical_position_range`

Beschreibung
: Größe des verfahrbaren Bereichs (s. Eigenschaft `PhysicalPositionRange` der Klasse `MC_832`).

Attribut `m_physical_index_position`

Beschreibung
: Abstand des Indexschalters von der linken Hardwareschranke (s. Eigenschaft `PhysicalIndexPosition` der Klasse `MC_832`).

Attribut `m_index_position`

Beschreibung
: Position des Indexschalters in Encoderschritten. Wird nach `SIP`-Kommando gesetzt und mit `RDIP` ausgelesen.

Attribut `m_velocity`

Beschreibung
: Simulierte Motorgeschwindigkeit zum in `m_position_time` gespeicherten Zeitpunkt (s. `CalcPosition`).

Attribut `m_max_velocity`

Beschreibung
: Motorparameter Maximale Geschwindigkeit. Wird durch `LTRJ/STT`-Kommando gesetzt.

Attribut `m_target`

Beschreibung
: Zielposition.

Attribut `m_status`

Beschreibung
Statusregister des Motors, s. Abb. B.9.

Attribut `m_signals_register`

Beschreibung
Signalregister des Motors, s. Abb. B.9.

Methoden

Methode `Konstruktor`

Signatur
```
Motor( MC_832* controller, int board_id )
```
Parameter
`controller` – Pointer auf den Controller der das Motorobjekt verwaltet.
`board_id` – Zahl die festlegt ob der Motor der erste oder der zweite der vom Controller verwalteten Motoren ist. Erlaubt Werte sind 0 und 1.

Methode `CalcPosition`

Signatur
```
long CalcPosition()
```
Beschreibung
Die Methode bestimmt die aktuelle Position und die aktuelle Geschwindigkeit. Sie verwendet dazu die Position und Geschwindigkeit die bei der letzten Bestimmung gespeichert worden sind und die Zeitdifferenz die seit der letzten Bestimmung vergangen ist, sowie die Parameter „maximale Geschwindigkeit“, „Beschleunigung“ und „Entschleunigung“. Außerdem testet sie, ob eventuell Hardwareschranken erreicht wurden.

Die Berechnungen sind analog zu die denen der gleichnamigen Methode der `Motor`-Klasse des Pakets C812 (s.o.). Der einzige Unterschied besteht in einer anderen Reaktion auf das Erreichen der Endlagenschalter. Der C832 setzt in diesem Fall nur eine Statusflag, setzt aber seine Bewegung unverändert fort.

E/A-Methoden Die folgenden Methoden dienen der vom Controller getriebenen Ein- und Ausgabekommunikation.

Methode `PutData`

Signatur
```
void PutData( byte b )
```
Beschreibung
Methode zum Senden eines Datenbytes an einen Motor. Wird vom Controller zur Realisierung einer Schreiboperation im Zustand „Datenregister (Motor 1/2)“ verwendet.

Methode `PutCmd`

Signatur
```
void PutCmd( byte b )
```

Beschreibung
Methode zum Senden eines *command codes* an einen Motor. Wird vom Controller zur Realisierung einer Schreiboperation im Zustand „Kommandoregister (Motor 1/2)“ verwendet. Die Methode delegiert den Aufruf entsprechend dem Parameter an eine der Kommandomethoden (s.u.)

Methode `GetData`

Signatur
```
byte GetData( )
```
Beschreibung
Methode zum Lesen eines Bytes vom Datenregister eines Motors. Wird vom Controller zur Realisierung einer Leseoperation im Zustand „Datenregister (Motor 1/2)“ verwendet.

Methode `GetStat`

Signatur
```
byte GetStat( )
```
Beschreibung
Liefert den Inhalt des Statusregisters eines Motors. Wird vom Controller zur Realisierung einer Leseoperation im Zustand „Kommandoregister (Motor 1/2)“ verwendet.

Kommandomethoden Die Motoren-Klasse implementiert für jedes der simulierten Kommandos eine Methode, die für dessen Verarbeitung verantwortlich ist. Der Name der Methode entspricht jeweils der Abkürzung des Kommandos (s. Abb. B.6). Die Kommandomethoden können in drei Gruppen gegliedert werden: Reportkommandos, Steuerkommandos mit Parametern und Steuerkommandos ohne Parameter.

Methoden für Reportkommandos

Signatur
```
void RDRP( )
void RDDP( )
void RDIP( )
void RDSIGS( )
```
Beschreibung
Die Methoden bestimmen den jeweils zu reportierenden Wert, legen ein Objekt der dazugehörigen Kommandoklasse an und speichern eine Referenz auf diesen im Attribut `m_current_cmd`. Z.B. bestimmt die Methode `RDRP` als erstes die aktuelle Position, legt eine Objekt der Klasse `CmdRDRP` an, initialisiert dieses mit der aktuellen Position und speichert eine Referenz auf dieses Objekt in `m_current_cmd`.

Methoden für Steuerkommandos mit Parametern

Signatur
```
void LTRJ( )
void LFIL( )
void MSKI( )
void RSTI( )
```

Beschreibung

Die durch diese Methoden realisierten Kommandos erfordern Parameter. Deshalb kann die eigentliche Aktion noch nicht ausgeführt werden. Die Methoden legen daher nur ein neues Kommandoobjekt an und speichern dies im Attribute `m_current_cmd`.

Die Kommandos `LTRJ` und `LFIL` sind *kumulativ*, d.h. die Parameter mehrerer Kommandos werden gespeichert und zusammen durch ein nachfolgendes `STT`- bzw. `UDF`-Kommando abgearbeitet. Darum initialisieren die `LTRJ`- und `LFIL`-Methode zusätzlich ihre Kommandoobjekte mit den aktuell geladenen Parametern, die in den Objekten der Variablen `m_current_trj` bzw. `m_current_fil` gespeichert sind.

Methoden für Steuerkommandos ohne Parameter

Signatur

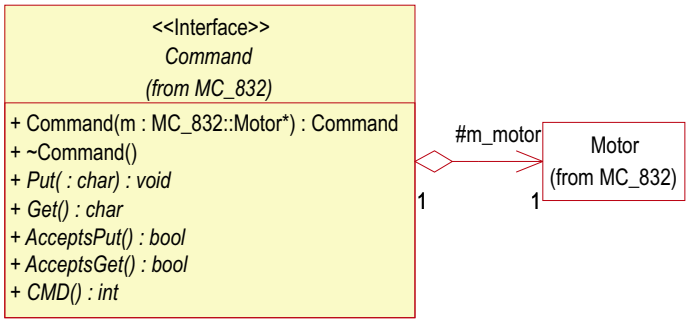
```
void RESET( )
void DFH( )
void STT( )
void UDF( )
void SIP( )
```

Beschreibung

Die Kommandos die durch diese Methoden realisiert werden, erwarten weder Parameter noch liefern sie eine Ausgabe, sondern werden sofort ausgeführt. Da sie gewissermaßen „zustandslos“ sind, gibt es auch keine Kommandoobjekte für diese Kommandos.

Die einzige Methode die etwas umfangreicher ist, ist die `STT`-Methode. Sie wertet die aktuellen Bewegungsparameter im aktuellen `CmdLTRJ`-Objekt aus und „löst die entsprechende Bewegung aus“.

Klasse: `Command`



Beschreibung

Abstrakte Basisklasse der Kommandoklassen, die für die Verwaltung des Abarbeitungszustandes der Motorkommandos des C832er Controllers und eventuell für deren Durchführung zuständig sind. Ein Kommando hat eine Referenz auf den Motor zu dem das auszuführende Kommando gehört, um Einfluss auf dessen Zustand nehmen zu können.

Methoden

Methode `AcceptsPut`

Signatur
```
virtual bool AcceptsPut( ) = 0
```

Beschreibung
Abstrakte Methode über die bestimmt werden kann, ob der Abarbeitungszustand des Kommandos eine Schreiboperation zulässt oder nicht.

Rückgabewert
`true`, falls eine Schreiboperation zulässig ist, `false` sonst.

Methode `AcceptsGet`

Signatur
```
virtual bool AcceptsGet( ) = 0
```

Beschreibung
Abstrakte Methode über die bestimmt werden kann, ob der Abarbeitungszustand des Kommandos eine Leseoperation zulässt oder nicht.

Rückgabewert
`true`, falls eine Leseoperation zulässig ist, `false` sonst.

Methode `Put`

Signatur
```
virtual void Put( byte ) = 0
```

Beschreibung
Die abstrakte Schreiboperation. Sie darf nur aufgerufen werden, wenn `AcceptsPut` `true` liefert.

Methode `Get`

Signatur
```
virtual byte Get( ) = 0
```

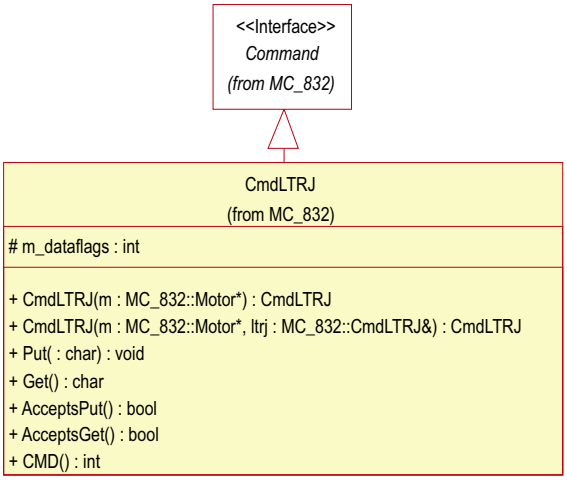
Beschreibung
Die abstrakte Leseoperation. Sie darf nur aufgerufen werden, wenn `AcceptsGet` `true` liefert.

Methode `CMD`

Signatur
```
virtual int  CMD() = 0
```

Beschreibung
Liefert den *command code* des implementierten Kommandos zu Identifikationszwecken.

Klasse: `CmdLTRJ`



Beschreibung

Klasse für die Abarbeitung des `LRTJ`-Kommandos des C832er Motorcontrollers.

Die Klasse implementiert, neben den Konstruktoren, ausschließlich die Methoden des `Command`-Interfaces. Das Kommando liefert keine Ausgabe, also liefert die `AcceptsGet`-Methode stets `false`, d.h. die `Get`-Methode darf nicht verwendet werden.

In Abb. 3.16 wird der durch die Klasse implementierte Zustandsautomat dargestellt. Ein `LTRJ`-Kommando erwartet als Parameter stets am Anfang ein 16bit-Datenword (*control word*), dass byteweise gesendet wird. Darum heißt der Startzustand `control word (byte 1)`, was zu lesen ist als: das nächste geschriebene Byte wird als Byte 1 des *control words* interpretiert.

Das *control word* bestimmt, ob und welche Parameter für das `LTRJ`-Kommando zu erwarten sind. So kann z.B. das Kommando zum Stoppen verwendet werden und es werden keine Parameter gesendet. In diesem Fall würde das System vom Zustand `control word (byte 0)` direkt in den Endzustand übergehen. In einem anderen Fall könnten die Parameter Beschleunigung (*acceleration*) und Maximalgeschwindigkeit (*velocity*) gesendet werden. In solchem Fall würde das System vom Zustand `control word (byte 0)` in den Zustand `acceleration (byte 3)` übergehen und nach dem Zustand `acceleration (byte0)` in den Zustand `velocity (byte 3)`. D.h. die empfangenen Bytes würden nacheinander als Byte 3 bis 0 des Beschleunigungsparameters und als Byte 3 bis 0 des Geschwindigkeitsparameters interpretiert werden. Überall, wo von einem Zustand mehrere Kanten wegführen, wird die Entscheidung mit Hilfe des *control words* gefällt.

Im Attribute `m_dataflags` wird gespeichert, welche Parameter durch die letzten `LTRJ`-Kommandos tatsächlich gelagen wurden.

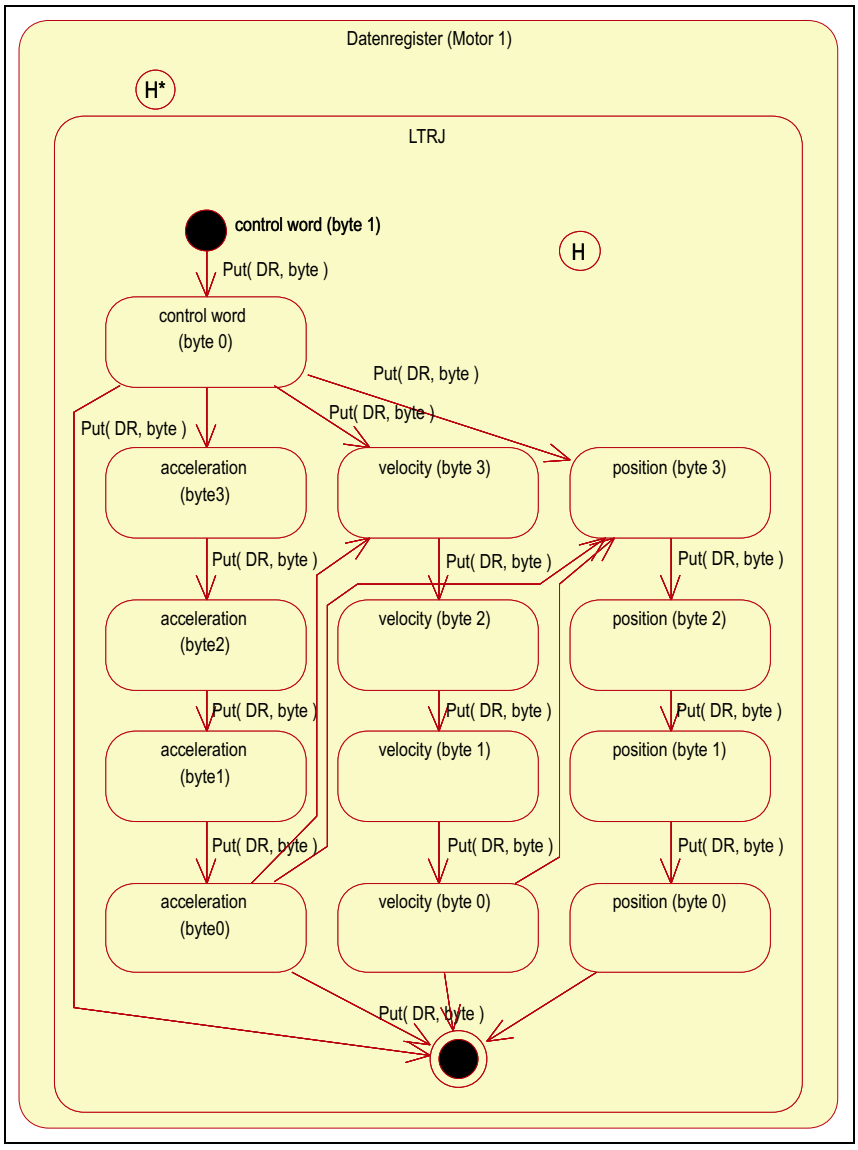


Abbildung 3.16: Zustandsdiagramm für Abarbeitung eines LTRJ-Kommandos

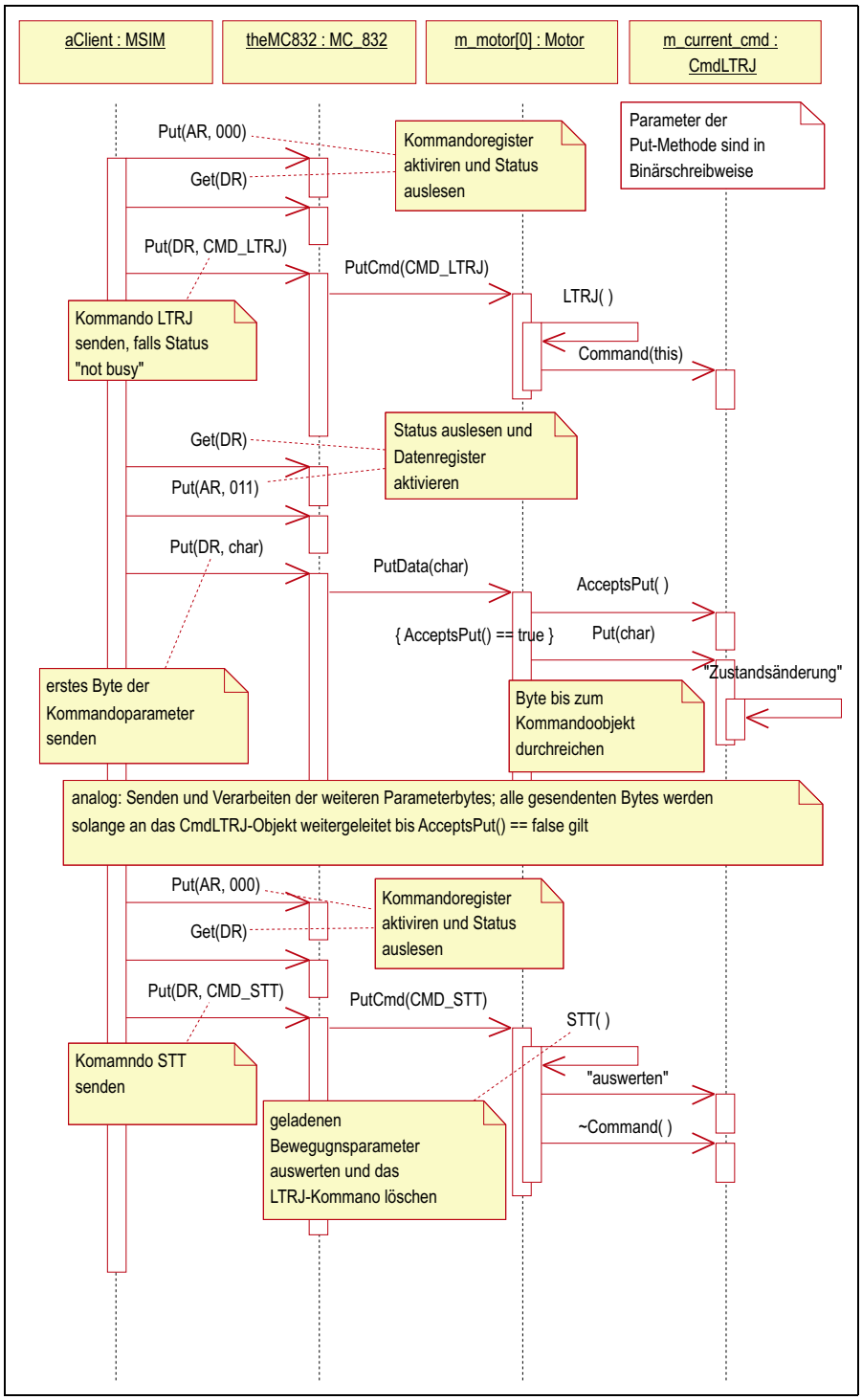


Abbildung 3.17: Sequenzdiagramm für die Abarbeitung eines `LTRJ`- und eines `STT-Kommandos`

3.4.5 Callback-Implementationen

Die Simulationskomponente implementiert einige Callback-Funktionen, die verwendet werden können, die Komponente bei der Motorenkomponente zu registrieren (s. Abschnitt 3.3.3). Die Callbacks werden alle über den `EXPORTS`-Abschnitt der Moduldefinitionsdatei (`msim.def`) exportiert, so dass sie mit der Windows-Funktion `GetProcAddress` importiert werden können, um so das in Abschnitt 3.3.2 beschriebene *run-time-linking* realisieren zu können. Für ein Beispiel s. Abb 3.18. Um Probleme mit der Groß- und Kleinschreibung beim Im- und Export zu verhindern, sind alle Namen der Funktionen in Großbuchstaben.

```
HINSTANCE hLib = LoadLibrary("msim.dll");
msRegister_C832_Put( (T832_PUT_CALLBACK)
        GetProcAddress(hLib, "SILENT_C832_PUT_CALLBACK") );
```

Abbildung 3.18: Verwendung der Callback-Implementationen (Beispiel)

Callbacks ohne Protokollierung

Die erste Gruppe der Callback-Implementierungen macht nichts weiter, als die Lese- oder Schreibanforderungen an die Simulationsobjekte weiter zu reichen.

Funktion: `SILENT_C812ISA_GET_CALLBACK`

Signatur

```
char WINAPI SILENT_C812ISA_GET_CALLBACK ( char*, char )
```

Beschreibung

Callback vom Typ `C812ISA_GET_CALLBACK`.

Funktion: `SILENT_C812ISA_PUT_CALLBACK`

Signatur

```
char WINAPI SILENT_C812ISA_PUT_CALLBACK ( char*, char )
```

Beschreibung

Callback vom Typ `C812ISA_PUT_CALLBACK`.

Funktion: `SILENT_C832_GET_CALLBACK`

Signatur

```
int  WINAPI SILENT_C832_GET_CALLBACK ( unsigned, int )
```

Beschreibung

Callback vom Typ `C832_GET_CALLBACK`.

Funktion: `SILENT_C832_PUT_CALLBACK`

Signatur

```
void WINAPI SILENT_C832_PUT_CALLBACK ( unsigned, int )
```

Beschreibung

Callback vom Typ `C832_PUT_CALLBACK`.

Callbacks mit Protokollierung

Die zweite Callback-Gruppe reicht, wie die erste, alle Lese- und Schreibanforderungen an die Simulationsobjekte weiter. Zusätzlich protokolliert sie bei einem LogLevel größer als 1 jede dieser Anforderungen.

Die Protokollierung erfolgt, um eine gute Weiterverarbeitung in einem Testrahmen zu ermöglichen, in XML-Notation. Dabei werden Schreiboperationen durch ein `<output>`-Element, und Leseoperationen durch ein `<input>`-Element dargestellt, die jeweils auf einer extra Zeile stehen.

Für die Schreiboperationen werden verschiedene Callbacks für die verschiedenen Simulationsmodi (s. S. 25) angeboten.

Funktion: VERBOSE_C812ISA_GET_CALLBACK_NS

Signatur

```
char WINAPI VERBOSE_C812ISA_GET_CALLBACK_NS ( char*, char )
```

Beschreibung

Protokollierender callback vom Typ `C812ISA_GET_CALLBACK` für den Protokollmodus. Protokolliert werden der Controllertyp (`C812ISA`), der Operationstyp (`get`) und die Zieladresse der Operation (`0x025f0800` im Bsp.) sowie der Hex-Code und eventuell eine lesbare Übersetzung des von der Hardware erzeugten Ergebnisbytes (`41 = A` im Bsp.).

Beispiel

```
<output>C812ISA get 0x025f0800 </output>
<input src='hw'>41 = A</input>
```

Funktion: VERBOSE_C812ISA_GET_CALLBACK_SO

Signatur

```
char WINAPI VERBOSE_C812ISA_GET_CALLBACK_SO ( char*, char )
```

Beschreibung

Protokollierender callback vom Typ `C812ISA_GET_CALLBACK` für den Simulationsmodus. Protokolliert werden der Controllertyp (`C812ISA`), der Operationstyp (`get`) und die Zieladresse der Operation (`0x025f0800` im Bsp.) sowie der Hex-Code und eventuell eine lesbare Übersetzung des von der Simulation erzeugten Ergebnisbytes (`fc` im Bsp.).

Beispiel

```
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
```

Funktion: VERBOSE_C812ISA_GET_CALLBACK_TS

Signatur

```
char WINAPI VERBOSE_C812ISA_GET_CALLBACK_TS (char*, char )
```

Beschreibung

Protokollierender callback vom Typ `C812ISA_GET_CALLBACK` für den Testmodus. Protokolliert werden der Controllertyp (`C812ISA`), der Operationstyp

(get) und die Zieladresse der Operation (0x025f0800 im Bsp.). Außerdem werden der Hex-Code und eventuell eine lesbare Übersetzung sowohl des von der Simulation als auch des von der Hardware erzeugten Ergebnisbytes protokolliert. Letztere werden verglichen und das Ergebnis des Test als Erfolg (pass) oder Fehlschlag (fail) verzeichnet.

Beispiel

```
<output>C812ISA get 0x025f0800 </output>
<input src='sim'>fc</input>
<input src='hw'>fc</input>
<sim-test>pass</sim-test>
```

Funktion: VERBOSE_C812ISA_PUT_CALLBACK

Signatur

```
void WINAPI VERBOSE_C812ISA_PUT_CALLBACK ( char*, char put )
```

Beschreibung

Protokollierender callback vom Typ C812ISA_PUT_CALLBACK. Protokolliert werden der Controllertyp (C812ISA), der Operationstyp (put), die Zieladresse der Operation (im Bsp. 0x025f03fc) sowie der Hex-Code und eventuell eine lesbare Übersetzung des geschriebenen Bytes (im Bsp. 45 = E).

Beispiel

```
<output>C812ISA put 0x025f03fc 45 = E</output>
```

Funktion: VERBOSE_C832_GET_CALLBACK_SO

Signatur

```
int  WINAPI VERBOSE_C832_GET_CALLBACK_SO ( unsigned, int )
```

Beschreibung

Protokollierender callback vom Typ C832_GET_CALLBACK. Protokolliert werden der Controllertyp (C832), der Operationstyp (get) und der I/O-Port von dem gelesen werden soll (im Bsp. 201) sowie die Binärdarstellung des von der Simulation generierten Leseergebnisses (im Bsp. 10000100).

Beispiel

```
<output>C832 get 201</output>
<input src='sim'>10000100</input>
```

Funktion: VERBOSE_C832_GET_CALLBACK_NS

Signatur

```
int  WINAPI VERBOSE_C832_GET_CALLBACK_NS ( unsigned, int )
```

Beschreibung

Protokollierender callback vom Typ C832_GET_CALLBACK. Protokolliert werden der Controllertyp (C832), der Operationstyp (get) und der I/O-Port von dem gelesen werden soll (im Bsp. 201) sowie die Binärdarstellung des von der Hardware generierten Leseergebnisses (im Bsp. 10000100).

Beispiel

```
<output>C832 get 201</output>
<input src='hw'>10000100</input>
```

Funktion: VERBOSE_C832_GET_CALLBACK_TS

Signatur

```
int  WINAPI VERBOSE_C832_GET_CALLBACK_TS ( unsigned, int )
```

Beschreibung

Protokollierender callback vom Typ `C832_GET_CALLBACK`. Protokolliert werden der Controllertyp (`C832`), der Operationstyp (`get`) und der I/O-Port von dem gelesen werden soll (im Bsp. `201`) sowie die Binärdarstellung sowohl des von der Hardware generierten als auch des von der Simulation erzeugten Leseergebnisses. Außerdem werden die letzten beiden Werte verglichen und das Ergebnis des Vergleichs als Erfolg (`pass`) oder Fehlschlag (`fail`) notiert.

Beispiel

```
<output>C832 get 201</output>
<input src='hw'>10000100</input>
<input src='sim'>10000101</input>
<sim-test>fail</sim-test>
```

Funktion: VERBOSE_C832_PUT_CALLBACK

Signatur

```
void WINAPI VERBOSE_C832_PUT_CALLBACK ( unsigned, int )
```

Beschreibung

Protokollierender callback vom Typ `C832_PUT_CALLBACK`. Protokolliert werden der Controllertyp (`C832`), der Operationstyp (`put`), der I/O-Port auf den geschrieben werden soll und die Binärdarstellung des geschrieben Bytes (`00011110` im Bsp.).

Beispiel

```
<output>C832 put 201 00011110</output>
```

Hilfsfunktionen

Funktion: INITCONTROLLERFROMINIFILE

Signatur

```
void WINAPI initControllerFromInifile( const char* inifile )
```

Beschreibung

Standardinitialisierung für Motorenkomponente entsprechend den Informationen der Konfigurationsdatei.

Damit die Motorsimulation eine bestimmte Testsituation realisiert, muss sie vor der Initialisierung der Motorenkomponente entsprechend konfiguriert werden. In den meisten Fällen wird es notwendig sein, die Simulation so zu konfigurieren, dass sie die Situation realisiert, die in der Konfigurationsdatei

abgespeichert ist. Diese Aufgabe übernimmt diese Funktion. Sie verwendet folgende Konfigurationsparameter um die entsprechenden Motoreigenschaften zu setzten:

Konfigurationsparameter	**Motoreigenschaften**
Hysteresis	Hysteresis
Upwards	LastDirection
RamAddr	BaseAddr (C812)
IOAddr	IOAddress (C832)
hwswDistance − PositionMin + DeltaPosition	PhysicalPosition
IndexPosition	PhysicalIndexPosition (C832, falls IndexLine=1))
PositionMax − PositionMin + 2 hwswDistance	PhysicalPositionRange

Die Eigenschaften, die nicht aus den Konfigurationsparametern der `MotorX`-Abschnitte bestimmt werden können, werden mit Hilfe neuer Parameter des `MOTORSIM`-Abschnittes ermittelt. Zu diesen (`IndexPosition`, `hwswDistance`) s. im Abschnitt Konfigurationsparameter S. 71.

3.4.6 Das Statusfenster

Zur Visualisierung des Zustandes der simulierten Motoren bietet die Simulationskomponente im aktuellen Zustand an, eine Textbox mit entsprechenden Informationen zu füllen. Der veränderte Anwendungsrahmen (s. Abschnitt 3.3.4) implementiert eine Fensterklasse, die eine solche Textbox enthält, und die in Abb. 3.19 dargestellt ist. Für den Inhalt dieser Textbox ist die Simulation verantwortlich. Im folgenden soll dieser Inhalt erläutert werden.

Die Statusinformation gliedert sich in 2 Hauptteile. Der erste Teil enthält Statusinformationen über die vier Motoren, die der simulierte C812er Controller verwalten kann, und der zweite enthält die zwei Motoren des C823er Controllers.

Der Status eines C812er Motors besteht aus zwei Zeilen. In der ersten steht ein Kleiner- oder ein Größer-Zeichen, das die aktuelle Bewegungsrichtung des Motors symbolisiert und eine Motornummer gefolgt von der Encoder-Position, der physikalischen Motorposition und dem Positionfehler (alle drei in Encoderschritten). Zu den Positionsbegriffen s. Abschnitt 3.4.1. Der Positionsfehler ist die Differenz zwischen der Soll- und der Ist-Encoder-Position. Die zweite Zeile enthält die Binärdarstellung des 8bit-Statusregisters für den Motor (zum Statusregister s. Abb. B.4).

Die Statusinformation für einen C832er Motor ist analog zu der des C812er aufgebaut, mit dem Unterschied, dass das die zweite Zeile das 16bit große Signalregister zeigt und das am Ende der zweiten Zeile ein Ausrufezeichen erscheint, wenn der Motor den verfahrbaren Bereich verlassen hat, d.h. ein Endlagenschalter angesprochen hätte (zum Signalregister s. Abb. B.9).

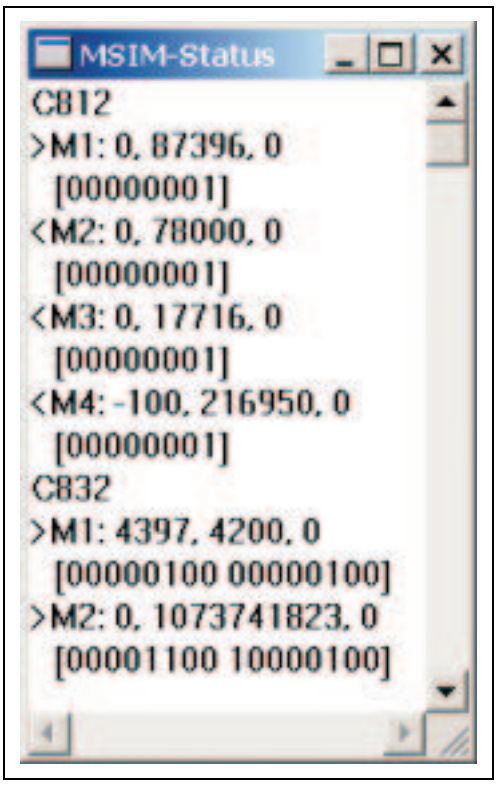


Abbildung 3.19: Das Status-Fenster

3.4.7 Dateiübersicht

Dateiname	Kommentar	LOC
msim.h		36
msim.cpp		580
msim.ide	Borland-Projektdatei (16bit dll)	–
msim.def	Moduldefinitionsdatei (16bit dll)	35
msim.rc	Resourcedatei (nur Versions-Resource, 16bit dll)	–
msim32.dsp	MS Visual C++ Projektdatei (32bit dll)	–
msim32.dsw	MS Visual C++ Workspacedatei (32bit dll)	–
msim32.rc	Resourcedatei (nur Versions-Resource, 32bit dll)	–
msimutil.h		96
C812		
mc_812.h		516
mc_812.cpp		1843
cmd.h		89
parser.l	Lexik	197
parserl.c	Lexer, flex-generiert	(2058)
parser.y	Grammatik	186
parsery.h	Parser, bison-generiert	(13)
parsery.c	Parser, bison-generiert	(989)
parser.cpp	Parser, cpp-Wrapper für parsery.c	10
bison.simple	bison-Hilfsdatei	(692)
C832		
mc_832.h		143
mc_832.cpp		1293
motor.h		607
		Σ 5631

Abbildung 3.20: Dateien der Simulationskomponente

3.5 Konfigurationsparameter für die Steuerung der Motorensimulation

Die Motorensimulation wird über einige Konfigurationsparameter in der Sektion `[MOTORSIM]` des ini-Files gesteuert, die im Folgenden beschrieben werden. Abb. 3.21 zeigt einen beispielhaften Ausschnitt aus einem Konfigurationsfile. Es ist zu berücksichtigen, dass die `Type`-Parameter der `[MotorX]`-Abschnitte die Werte `C-812ISA` oder `C-832` haben müssen.

```
[MOTORSIM]
SimulationType=simulation_only
LogLevel=0       ; kein Protokoll
LogFile=test.log
StatusWindow=1   ; Statusfenster Ein
dll=msim.dll
```

Abbildung 3.21: Beispielhafte `MOTORSIM`-Sektion

`SimulationType`
: Legt den Simulationsmodus fest (s. S. 25).
Werte: `[no_simulation|simulation_only|test_simulation]`
`no_simulation` stellt den Normalmodus ein (Voreinstellung)
`simulation_only` stellt den Simulationsmodus ein
`test_simulation` stellt den Vergleichs- oder Protokollmodus ein

`LogFile`
: Name der Datei, in die die Kommunikation mit der Hardware protokolliert werden soll.
Voreinstellung: `"msim.log"`.

`LogLevel`
: Legt fest, ob bzw. wie ausführlich das Protokoll sein soll.
Werte: `[0|1|2]`
0 – kein Protokoll (Voreinstellung)

`StatusWindow`
: Legt fest, ob das Statusfenster angezeigt werden soll.
Werte: `[0|1]`
0 – kein Statusfenster (Voreinstellung)

`dll`
: Legt den Namen der DLL fest, in der die Implementation der Simulation zu finden ist. Voreinstellung ist `"msim.dll"`. Die DLL wird mit `LoadLibrary` geladen, d.h. falls nur der Dateiname angegeben wird, wird die Bibliothek in folgenden Verzeichnissen gesucht: im aktuellen Verzeichnis, im Windows- und im System-Verzeichnis, im Verzeichnis von `develop.exe` und im Pfad.

`IndexPosition`
: Legt den (zu simulierenden) Abstand zwischen der linken Hardwareschranke und dem Indexschalter in Encoderschritten fest.
Werte: `[0..]`
Voreinstellung: 5000

`hwswDistance`
: Legt den (zu simulierenden) Abstand zwischen Hardwareschranke (Endlagenschalter) und Softwareschranke in Encoderschritten fest. Bei positiven Werten liegen die simulierten Endlagenschalter hinter den Softwareschranken, bei negativen Werten davor. Dient dem Test ungültiger Softwareschranken.
Voreinstellung: 4100

`PositionOffset`
: Legt eine Verschiebung der tatsächlichen Motorposition gegenüber der aus der Konfigurationdatei ermittelbaren fest. Dient dem Test mit Konfigurationsparametern, die im Bezug auf die Motorposition ungültig sind. Sollte größer oder gleich als `RemoveLimit` sein.
Voreinstellung: 0

3.6 Test der Simulation

Um das Kapitel zur Motorensimulation abschließen zu können, muss noch überprüft werden, in wieweit die Simulation der Motoren der Realität entspricht, bzw. ob sie ihr hinreichend nahe kommt.

Dazu müssen die echte Motorenhardware und die Motorensimulation parallel betrieben und deren Antworten auf die Softwareanforderungen verglichen werden. Der Test besteht also darin, die Motorenkomponente im Vergleichsmodus zu betreiben (s.o. S. 25), sie mit der Simulationskomponente zu verbinden, einige grundlegende Steuerabläufe durchzuführen und das entstandene Protokoll auszuwerten.

Ort-Zeit-Verhalten Als erstes wurden die Protokolle hinsichtlich des Umgebungsaspektes Ort-Zeit-Verhalten ausgewertet. Dazu wurden die von Hardware und Simulation generierten Positionsangaben extrahiert und in Diagrammen dargestellt.

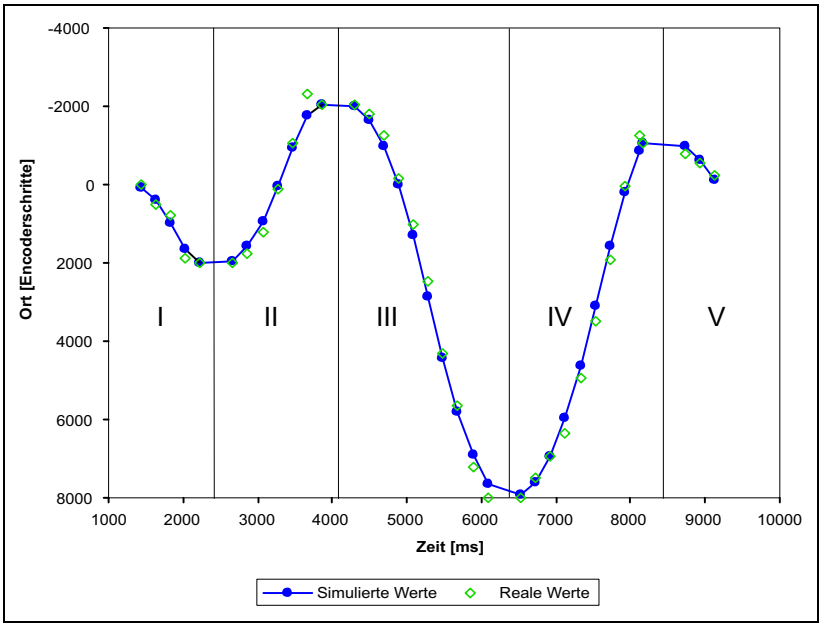


Abbildung 3.22: Ort-Zeit-Diagramm 1

Abb. 3.22 zeigt das Ort-Zeit-Diagramm einer Abfolge von fünf Bewegungen. Darin werden als dunkle Punkte die von der Simulationskomponente generierten Positionsangaben dargestellt und durch Linien verbunden. Die hellen Vierecke sind die vom C812er Motorcontroller gelieferten Positionen. Die fünf Abschnitte des Diagramms (I-V) begrenzen jeweils die fünf Bewegungen. Im Abschnitt I startet der Motor bei der Position 0 und erhält die Anweisung nach „rechts“ auf Position 2000 zu fahren, im Abschnitt II wird der Motor von der Position 2000 nach „links“ auf die Position -2000 gefahren usw.

Es können, v.a. im Abschnitt III, deutlich die 3 Bewegungsphasen (Beschleunigung, konstante Geschwindigkeit und Bremsung) unterschieden werden. Die Linie, die die Punkte der simulierten Positionswerte verbindet, nähert sich der

einer idealen Bewegungskurve an, was auf die Berechnung dieser Werte mit Hilfe theoretischer Ort-Zeit-Gesetze zurückzuführen ist. Die realen Werte liegen selten auf den simulierten. Der mittlere Positionfehler liegt bei 190 Encoderschritten. Aber der realen Bewegungstendenz folgt die Simulation gut. Die auffälligsten Abweichungen sind in der Bremsphase festzustellen. Der echte Motor beginnt später zu bremsen, fährt z.T. über die Zielposition hinaus und bremst insgesamt wesentlich abrupter.

Abb. 3.23 zeigt das Ort-Zeit-Diagramm einer Bewegung, die den Motor an die linke Hardwareschranke führt. Der Motor startet bei der Position 0 und erhält ein Kommando zu einer Position weit hinter der linken Hardwareschranke zu fahren. Der Endlagenschalter spricht bei ca. -24000 an. Im Abschnitt I beschleunigt der Motor, im Abschnitt II hat er die maximale Geschwindigkeit erreicht und hält diese bei. Im dritten Abschnitt spricht der Endlagenschalter an, der Motor bremst sehr abrupt (Abschnitt III ist deutlich kürzer als Abschnitt I) und fährt in Abschnitt IV um eine konfigurierbare Strecke zurück (in diesem Fall um 4000 Schritte).

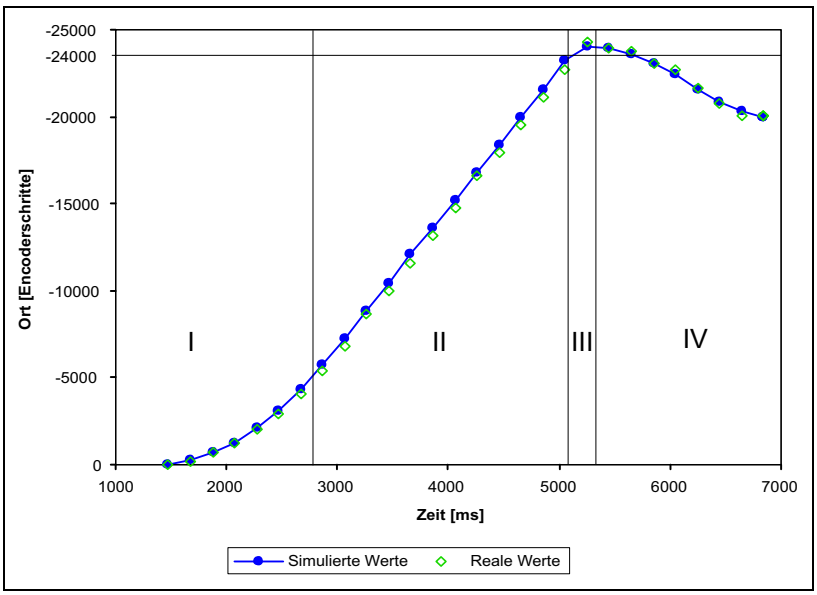


Abbildung 3.23: Ort-Zeit-Diagramm 2

Auch im zweiten Beispiel liegen die Positionswerte der Simulation und der Realität in ausreichendem Maß beieinander. Die gleichen Ergebnisse wurden mit weiteren Tests erzielt, so dass zusammenfassend festgestellt werden kann, dass die Simulation beim Aspekt Ort-Zeit-Verhalten die Umwelt adäquat wiedergibt.

E/A-Interaktion Zum zweiten wurde der Interaktionsaspekt untersucht. Dabei wurden alle Antworten der echten und der simulierten Hardware auf die Softwareanforderungen auf Gleichheit getestet und auftretende Unterschiede untersucht. Unterschiede waren in zwei Arten von Situationen zu beobachten:

1. *systematisch beim Auslesen von Positions- bzw. Positionsfehlerangaben.* Dieser Fall entspricht dem oben zum Ort-Zeit-Verhalten Gesagten. Die Po-

sitionsangaben werden von der nutzenden Software byteweise ausgelesen. Differenzen in den Positionsangaben zwischen Simulation und Hardware führen dazu, dass auch zumindest die zwei niederwertigen Bytes jeweils von einander abweichen.

2. *unsystematisch nach dem Ausführen verschiedener Kommandos.* Dieser Unterschied bei den Antworten tritt nur beim Auslesen des Statusbytes nach dem Ausführen einiger Kommandos auf; z.B. setzt der echte Controller sein *busy*-Byte etwas später zurück als die Simulation oder setzt das *data available*-Flag etwas später. Das ist auf Differenzen in der Verarbeitungsgeschwindigkeit zwischen Controller und Rechner zurückzuführen, die sicher bei der Verwendung schnellerer Rechner noch größer werden. Diese Verzögerungen treten aber unsystematisch auf und können daher von der Simulation nicht systematisch nachgebildet werden, so dass diese Unterschiede nicht zu verhindern sind.

Die auftretenden Unterschiede bei der E/A-Interaktion sind entweder Unterschiede im Rahmen akzeptabler Abweichungen bei Positionsangaben oder unsystematische Differenzen, die die Simulation nicht abfangen kann. In allen anderen Situationen liefert die Simulation Ergebnisse, die zu den realen identisch sind. D.h. dass auch für den Aspekt der E/A-Interaktion die Simulation den Anforderungen genügt.