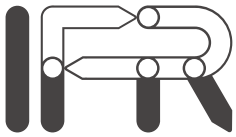
Institut für Regelungstechnik

TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

Prof. Dr.-Ing. W. Schumacher
Prof. Dr.-Ing. M. Maurer
Prof. em. Dr.-Ing. W. Leonhard



Automatisierungstechnisches Praktikum

Regelung eines fahrerlosen Transportsystems mithilfe des Echtzeitbetriebssystems QNX

Datum: 7. Juli 2010

Inhaltsverzeichnis

1 Einleitung **1**

2 Systemaufbau des fahrerlosen Transportfahrzeugs **2**

3 Echtzeitbetriebssysteme am Beispiel von QNX **5**
3.1 Prozesse 5
3.1.1 Prozesszustände und prioritätengesteuerte Prozessorzuteilung 6
3.1.2 Interprozess-Kommunikation 8
3.2 Threads 11
3.3 Schlossvariablen 12
3.4 Timer 12

4 Modellbildung **14**

5 Versuchsdurchführung **18**
5.1 Hausaufgabe: Reglerimplementierung 18
5.2 Interprozess-Kommuniaktion und Hardware-Zugriff unter QNX 19
5.2.1 Interprozess-Kommunikation 20
5.2.2 Interprozess-Kommunikation über ein Netzwerk 21
5.2.3 Timer-Programmierung 22
5.2.4 Zugriff auf die Peripherie 23
5.2.5 Implementierung einer einfachen Regelung 24
5.3 Reglerentwurf 26
5.4 Versuchsfahrten mit zeitdiskreter Reglerauslegung 29

1 Einleitung

Autonome Fahrzeuge werden heute vielfach eingesetzt, sei es z. B. im industriellen Umfeld zur Automatisierung von Transportvorgängen oder im wissenschaftlichen Bereich zur Erforschung unwirtlicher Regionen. So unterschiedlich die Anforderungen an diese Systeme, ihre Komplexität und ihre Einsatzgebiete auch sein mögen, so gleichermaßen identisch sind sie in ihrem prinzipbedingten Aufbau, denn sie verfügen alle über:

- eine Umfeldsensorik zur Ermittlung der Position im Raum
- eine Steuereinheit mit einer Bahnplanung, die aus den Daten der Umfeldsensorik ermittelt, wie das vorgegebene Ziel erreicht werden kann
- eine Aktorik zur Erreichung des Ziels

Wie sich dieser scheinbar banale Aufbau praktisch darstellt, soll anhand dieses Laborversuchs demonstriert werden. Dazu wurde am Institut für Regelungstechnik ein einfaches, aber voll funktionsfähiges fahrerloses Transportfahrzeug (FTF) aufgebaut, das im Folgenden vorgestellt werden soll.

2 Systemaufbau des fahrerlosen Transportfahrzeugs

Die beiden folgenden Abbildungen zeigen das Fahrzeug ohne Verkleidung, sodass die verschiedenen Komponenten des FTF leicht ersichtlich sind. Vor der Antriebseinheit ist der Sensor angebracht, der die Umfeldsensorik des Fahrzeugs darstellt. Dabei wird mit einem Leitdraht, der von einem Wechselstrom durchflossen wird, eine Spur vorgegeben. Dieser Strom induziert in einer Messspule im Sensor einen Strom, aus dessen Stärke der Abstand berechnet wird.

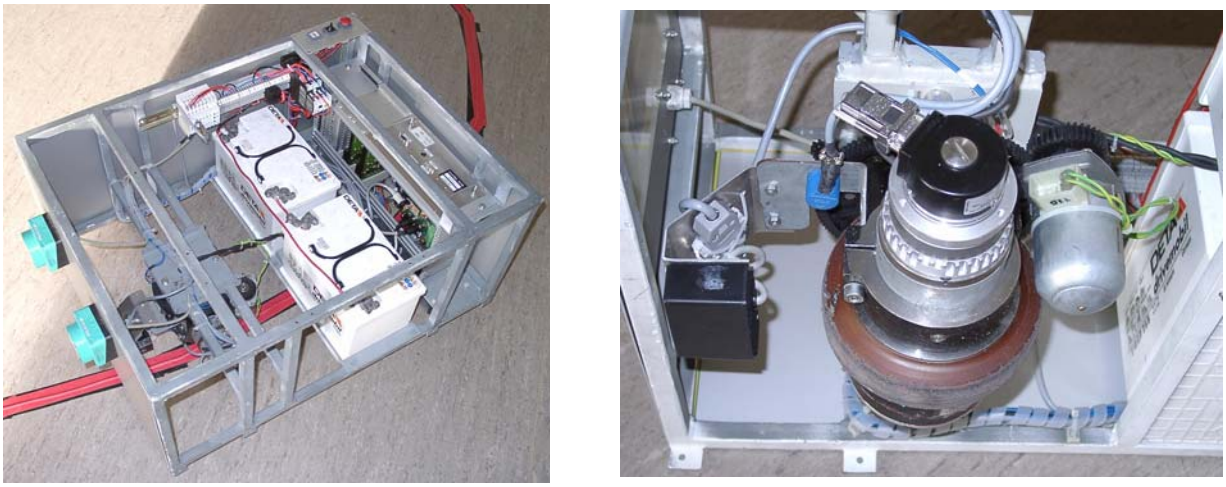
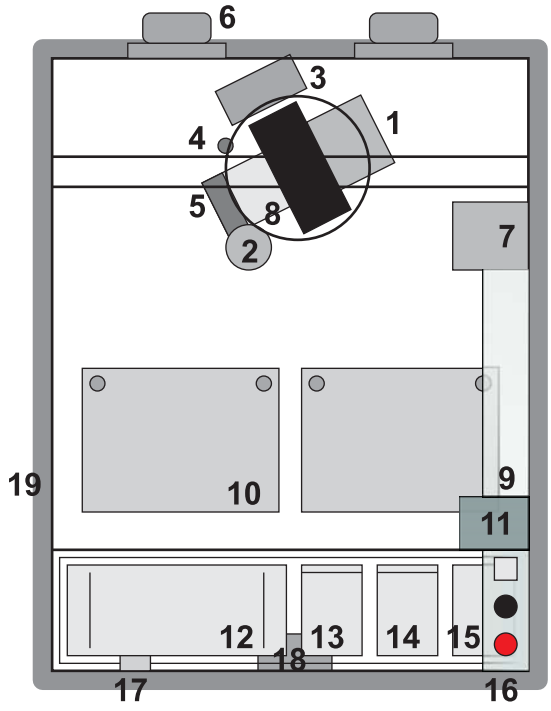


Bild 2.1: FTF ohne Verkleidung (Blick von oben und Antriebseinheit)

Der Systemaufbau des Fahrzeugs ist in der Abbildung 2.2 schematisch dargestellt. Die einzelnen Komponenten sollen im Anschluss kurz vorgestellt und ihr Zusammenwirken verdeutlicht werden. Sie umfassen neben Sensorik, Steuerung und Aktorik, auch ein Sicherheits- und Not-Aus-System.

Insbesondere sei hier auf die Antriebseinheit und der umgebenden Sensoren hingewiesen. Der Leser möge sich die Lage der einzelnen Komponenten anhand der schematischen Darstellung und der obigen Abbildung verdeutlichen.



1 *Antriebsmotor*
2 *Lenkmotor*
3 *Vorderer Mikrolenksensor*
4 *Lenkpotentiometer*
5 *Inkrementalgeber*
6 *Ultraschallsensoren*
7 *Trans-O-Prox zur Auswertung der Ultraschallsensoren*
8 *Elektromagnetbremse*
9 *Klemmleiste*
10 *Gelbatterie*
11 *Schütz*
12 *Industrie-PC*
13 *PWM-Modul zur Ansteuerung des Lenkmotors*
14 *PWM-Modul zur Ansteuerung des Antriebsmotors*
15 *DC/DC-Wandler zur Energieversorgung*
16 *Hauptschalter, Start-Taster und Not-Aus*
17 *Anschlussbuchse zum Laden der Batterien*
18 *Hinterer Mikrolenksensor*
19 *Bumper (Not-Aus)*

Bild 2.2: Bestandteile des FTF

- **Antriebsmotor:** Ein permanent erregter Gleichstrommotor mit einer Leistung von 0,13kW/24V kann das Fahrzeug maximal mit einer Geschwindigkeit von $1\,\mathrm{ms}^{-1}$ antreiben.

- **Lenkmotor:** Über den Lenkmotor kann das Fahrzeug gesteuert werden. Dabei wird die komplette Antriebseinheit inklusive des Antriebsmotors geschwenkt.

- **Mikrolenksensor:** Das Fahrzeug besitzt zwei Mikrolenksensoren am vorderen und hinteren Ende, die den Abstand zum Leitdraht messen. Der hintere Sensor ist dabei für Rückwärtsfahrt vorgesehen und soll an dieser Stelle nicht weiter interessieren.

- **Lenkpotentiometer:** Mit dem Schwenken der Antriebseinheit wird über ein Zahnrad ein kleines Präzisionspotentiometer verändert, aus dessen Stellung der Lenkwinkel ermittelt werden kann.

- **Inkrementalgeber:** Gleichermaßen wichtig wie der Lenkwinkel ist auch die gefahrene Geschwindigkeit. Diese wird mit einem Inkrementalgeber gemessen, der direkt an der Motornabe angebracht ist.

- **Ultraschallsensoren:** Zur Erkennung von Hindernissen sind an der Vorderseite zwei

Ultraschallsensoren angebracht, die den Bereich vor dem Fahrzeug abtasten und den Abstand zu möglichen Hindernissen messen.

- **Elektromagnetbremse:** Im Falle eines Not-Aus zieht die Bremse an und hält das Fahrzeug unmittelbar an. Dies erfolgt entweder bei Betätigung des Not-Aus-Schalters oder bei Verformung des Bumpers, der um das Fahrzeug herum angebracht ist.
- **Energieversorgung:** Zwei 10Ah/12V Gel-Batterien sorgen für die nötige Energieversorgung des Fahrzeugs, das im Fahrbetrieb etwa 4A Strom zieht. Diese Spannung wird über den DC/DC-Wandler stabilisiert und der Steuereinheit zugeführt.
- **Industrie-PC:** Auf dem Fahrzeug ist ein sehr kompakt aufgebauter Computer (Pentium 150MHz) untergebracht, der unter dem Echtzeitbetriebssystem QNX die Steuerung des Fahrzeugs übernimmt. Alle Sensorsignale werden auf ihm zusammengeführt und ausgewertet, entsprechende Stellsignale anschließend ausgegeben. Dazu ist der Rechner mit einem zusätzlichen IO-Modul, einem sogenannten IP-Modul ausgerüstet.
- **PWM-Module:** Die Motoren werden mit einer PWM-Spannung angetrieben. Die PWM-Signale des PCs müssen folglich entsprechend verstärkt werden.
- **Hauptschalter, Start- und Not-Aus-Taster:** Über den Hauptschalter wird das Fahrzeug gestartet und ein Steuerstromkreis eingeschaltet, sodass der Rechner hochfährt. Durch Betätigung des Start-Tasters wird der Antriebsstromkreis zugeschaltet. Die PWM-Module sind nun betriebsbereit und die Elektromagnetbremse ist gelöst. Erst jetzt kann das Fahrzeug losfahren. Mittels des Not-Aus-Schalters oder des Bumpers wird nur dieser Antriebsstromkreis unterbrochen und das Fahrzeug wird gestoppt, der Rechner und die Steuerung bleiben in Betrieb.

Auf die einzelnen Komponenten soll nicht näher eingegangen werden, denn die genaue Kommunikation mit der Peripherie und deren Ansteuerung ist nicht Teil dieses Praktikum, sondern es können vorhandene Routinen verwendet werden, die die entsprechenden Hardware-Aufrufe darstellen.

Damit aber nun das Fahrzeug dem Leitdraht entlang fahren kann, muss eine Regelung implementiert werden. Der Fahrzeug-Rechner ist dazu über ein Wireless-LAN mit einem Terminal verbunden. Beide Rechner kommunizieren im Betrieb laufend miteinander, sodass am Terminal das FTF gestartet, gestoppt und der momentane Zustand überwacht werden kann. Dazu ist auf beiden Rechnern das Echtzeitbetriebssystem QNX installiert, das im Anschluss mit seinen wesentlichen Merkmalen kurz erläutert werden soll.

3 Echtzeitbetriebssysteme am Beispiel von QNX

Wird ein PC zur Regelung und Steuerung eingesetzt, so werden hohe Anforderungen in Bezug auf Stabilität und Determiniertheit gestellt, da es sonst zu katastrophalen Folgen kommen könnte. Dazu muss das Betriebssystem verschiedene Mechanismen bereitstellen, die wichtigsten sollen im Folgenden am Beispiel von QNX vorgestellt werden. Einen tieferen Einstieg vermittelt z. B. die Vorlesung „Prozessinformatik“. Nicht-Echtzeitfähige Betriebssysteme verfügen zum Teil über gleiche oder ähnliche Mechanismen, können aber ein deterministisches Zeitverhalten bei hoher Stabilität aufgrund ihrer Architektur nicht (immer) gewährleisten.

Den Begriff der Determiniertheit definiert die DIN 44300 folgendermaßen:

> „Realzeitbetrieb ist der Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten einmalig oder in regelmäßigen Abständen anfallen.“

Damit diese Techniken und Mechanismen bei der Software-Entwicklung auch vom Programmierer genutzt werden können, ist die Standard-C-Bibliothek erweitert worden und umfasst alle benötigten Funktionen zur Ausführung der entsprechenden (Kernel-)Aufrufe.

3.1 Prozesse

Die anstehenden Aufgaben werden von Programmen, die im Folgenden als Prozesse bezeichnet werden, bearbeitet. Diese Prozesse kooperieren zum Teil miteinander, z. B. bei der Weiterverarbeitung von Daten, konkurrieren aber allesamt miteinander, z. B. um vorhandene Betriebsmittel wie die CPU. Die Organisation der einzelnen Prozesse erfolgt durch das

Betriebssystem, das in seinem Kernel einem Verwalter, den sogenannten *Dispatcher* oder *Scheduler* besitzt. Dieser schaltet zwischen den einzelnen Prozessen um (*Context-Switch*) und entscheidet, welcher Prozess als nächstes ausgeführt werden soll.

3.1.1 Prozesszustände und prioritätengesteuerte Prozessorzuteilung

Der Scheduler verwaltet sämtliche Prozesse, indem er ihnen verschiedene Zustände zuordnet und zu jedem Zustand eine Warteschlange führt, in der alle Prozesse mit dem jeweiligen Zustand eingereiht sind. Diese lauten:

- *existent:* Ein Prozess ist existent, wenn er der Prozessverwaltung bekannt ist, jedoch noch keinen anderen Zustand durchlaufen hat.
- *ready:* Ein Prozess ist bereit, wenn er den Prozessor nutzen möchte und nur auf die Zuteilung wartet.
- *running:* Ein Prozess ist laufend, wenn ihm der Prozessor zugewiesen ist und er ihn benutzt.
- *blocked:* Wartet ein Prozess auf das Eintreffen eines Ereignisses oder die Freigabe eines Betriebsmittels, so ist er blockiert.[1]
- *terminated:* Nach Ausführung der letzten Anweisung wird der Prozess beendet.

Die Prozesszustände werden vom Scheduler im Kernel laufend überwacht und bei einem Wechsel neu zugeordnet. Dazu wird die Rechenzeit in sogenannte Zeitscheiben diskretisiert. Bei jedem Ablauf einer Zeitscheibe oder bei asynchronen Ereignissen (Unterbrechungsanforderung, laufender Prozess beendet sich, etc.) wird dieser aktiv, kontrolliert die Prozesse und entscheidet, welcher Prozess seinen Zustand ändert und wem eventuell der Prozessor neu zugeteilt wird. Wurde z. B. ein Betriebsmittel von einem Prozess freigegeben, auf das ein anderer Prozess gewartet hat, so könnte dieser nun weiterarbeiten und wird daher aus der *blocked*-Warteschlange entnommen und in die *ready*-Warteschlange eingereiht. Aus dieser Warteschlange wird dann der nächste Prozess, dem der Prozessor zugeteilt werden soll, ausgewählt. Den genauen Wechsel zwischen den Zuständen erklärt die Abbildung 3.1.

Anhand dieser *ready*-Warteschlange wählt der Scheduler im Kernel aus den bereiten Prozessen denjenigen aus, der als nächstes den Prozessor zugeteilt bekommen soll. Dazu wird jedem Prozess eine Priorität, die Auskunft über seine Dringlichkeit gibt, zugeordnet. Dies

[1] QNX unterteilt diesen Zustand entsprechend der Blockierungsursache weiter in zwanzig verschiedene, die aber an dieser Stelle nicht weiter interessieren sollen.

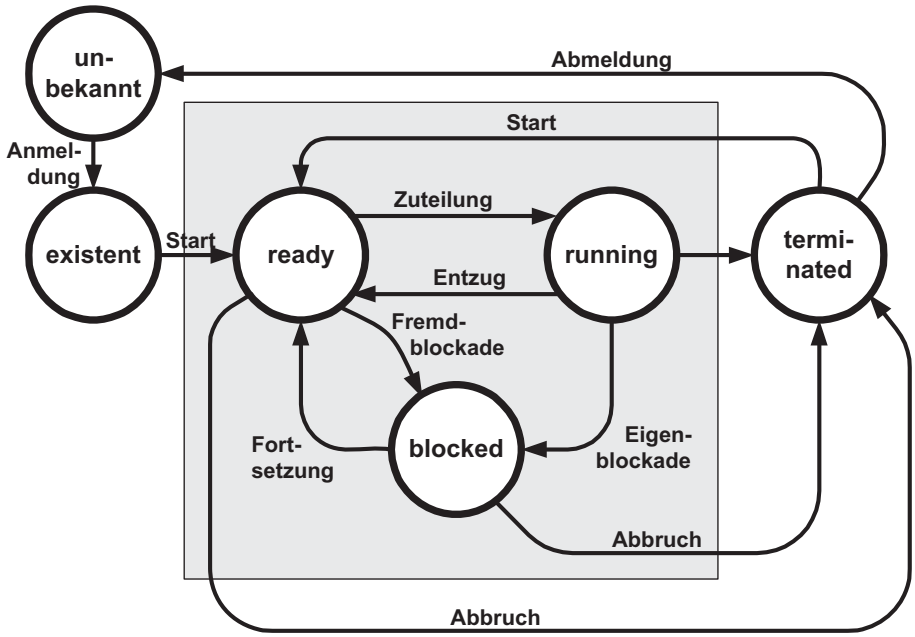


Bild 3.1: Zustandsübergänge von Prozessen

geschieht durch eine Vorabvergabe und kann sich während des laufenden Betriebes aber auch verändern.

Es gibt verschiedenste Scheduling-Algorithmen, wobei hier nur das *Round-robin*-Verfahren als wichtigstes und gebräuchlichstes anhand der Abbildung 3.2 vorgestellt werden soll: In der *ready*-Warteschlange wird zu jeder Priorität eine Liste der Prozesse, z. B. **A**, **B**, **C** geführt. Ein Prozess läuft nun solange, bis er selbst den Prozessor abgibt, von einem höherprioren Prozess unterbrochen wird oder sein Zeitbudget (Zeitscheibe) abgelaufen ist. Dann wird er an das Ende der Warteschlange eingereiht, in diesem Fall Prozess **A**, und der nächste kommt zur Ausführung (**B**). Für alle übrigen Zustände ist eine Einteilung entsprechend der Prioritäten natürlich nicht nötig.

Durch das Scheduling mit Prioritäten kann es leider auch zu Problemen, z. B. *Prioritäten-inversionen* und *Dead-Locks* kommen, die durch eine sogenannte Prioritätenvererbung minimiert werden können. Auf weitere Details soll verzichtet werden, es sei aber darauf aufmerksam gemacht, dass stets eine Restverantwortung beim Programmierer bleibt und dieser sich über die eingesetzten Techniken im Klaren sein muss.

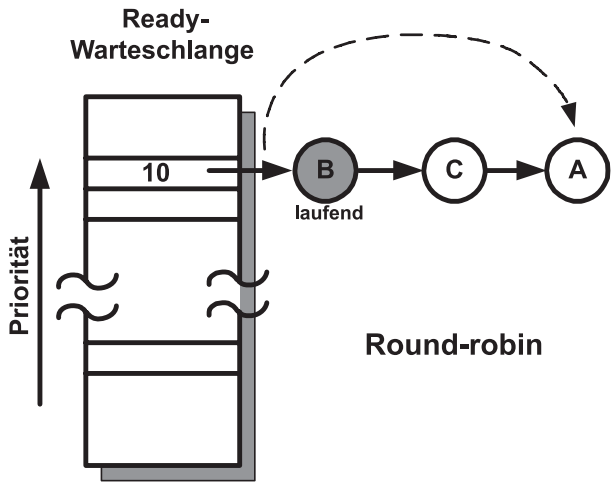


Bild 3.2: Zustandsübergänge von Prozessen

3.1.2 Interprozess-Kommunikation

Damit Prozesse miteinander kooperieren können, müssen sie Nachrichten und Informationen miteinander austauschen. Es gibt drei grundlegende Techniken, die unter dem Begriff *Interprozess-Kommunikation* zusammengefasst werden.

- *Nachrichten (Message-passing):* Ein Prozess schickt einem zweiten eine Nachricht, z. B. eine Struktur, der auf diese mit einer zweiten Nachricht antworten muss.

- *Signale (Signals):* Ein Signal kann als ein Software-Interrupt interpretiert werden, der von einem Prozess ausgelöst werden kann. Der Empfänger muss dieses Signal mit einer entsprechenden Software-Routine ähnlich einer Interrupt-Service-Routine abfangen.

- *gemeinsamer Speicherbereich (Shared memory):* Zwei Prozesse definieren einen globalen Speicherbereich, auf den beide zugreifen dürfen ähnlich einer globalen Variable. Dieser Datenaustausch empfiehlt sich insbesondere für größere Datenmengen, soll an dieser Stelle aber nicht weiter interessieren.

Diese Mechanismen funktionieren gleichermaßen auch über eine Netzwerkverbindung und unterscheiden sich in der Anwendung für den Programmierer fast überhaupt nicht von einer Ein-Rechner-Realisierung. Es ist also problemlos möglich, zwei Prozesse über ein Netz miteinander kooperieren zu lassen. Dies wird beim FTF genutzt. Über eine Wireless-LAN-Verbindung ist eine eine elegante Kommunikation zu einem Terminal-Rechner realisiert worden.

Beim *Message-passing* wird durch einen Prozess ein Nachrichtenkanal aufgebaut. Dieser Prozess wirkt dann als ein Server, an dessen Kanal sich nun beliebig viele weitere Client-Prozesse hängen und somit dem Server-Prozess eine Nachricht schicken können, wobei die Kommunikation in Form einer Sternstruktur nur zwischen Clients und Server erfolgen kann, nicht aber zwischen den Clients direkt. Dadurch entsteht eine wohldefinierte Kommunikationshierachie.

Mit dem Befehl `ChannelCreate()` wird ein Nachrichtenkanal kreiert, der dem aufrufenden Prozess, dem Server zugeordnet ist. Ein weiterer Prozess, der Client kann sich nun mit Hilfe des Befehls `ConnectAttach()` an diesen Kanal hängen, sodass eine Verbindung zwischen beiden Prozessen hergestellt ist. Ein einfacher Informationsaustausch kann bereits mit den drei Funktionen `MsgSend()`, `MsgReceive()` und `MsgReply()` geschehen. Der Client sendet durch Aufruf des Befehls `MsgSend()` seine Nachricht an den Server, der sie mit Hilfe von `MsgReceive()` empfängt. Dazu kopiert der Kernel die Nachricht aus dem Speicherbereich des Clients in den des Servers. Der Server antwortet nun, nachdem der Auftrag erledigt worden ist mit `MsgReply()` dem Client, und übergibt ihm dabei wiederum seine Information. Während der Kommunikation werden die beteiligten Prozesse zeitweilig blockiert. Das bedeutet, dass der Client (Sender) so lange angehalten ist, bis die Nachricht vom Server (Empfänger) bearbeitet und beantwortet ist. Die Funktionsaufrufe sehen im Detail wie folgt aus, wobei dem Verständnis halber einige Parameter mit ihren Standardwerten eingesetzt sind, die nur für komplizierte Aufrufe eine Bedeutung besitzen:

```
#include <sys/neutrino.h>

// return: channel_id
int ChannelCreate (0);

// return: connection_id
int ConnectAttach (int rechner_id, int prozess_id, int channel_id, 0, 0);

// return: ungleich 0 wenn Fehler
int MsgSend (int connection_id,
             const void *zeiger_auf_nachricht, int laenge_der_nachricht_in_bytes,
             const void *zeiger_auf_antwort,   int laenge_der_antwort_in_bytes);

// return: sender_id
int MsgReceive (int channel_id,
                void *zeiger_auf_nachricht, int laenge_der_nachricht_in_bytes,
                NULL);
```

```
// return: ungleich 0 wenn Fehler
int MsgReply (int sender_id, 0,
              const void *zeiger_auf_antwort, int laenge_der_antwort_in_bytes);
```

Der Ablauf der Kommunikation kann anhand der Abbildung 3.3 verdeutlicht werden.

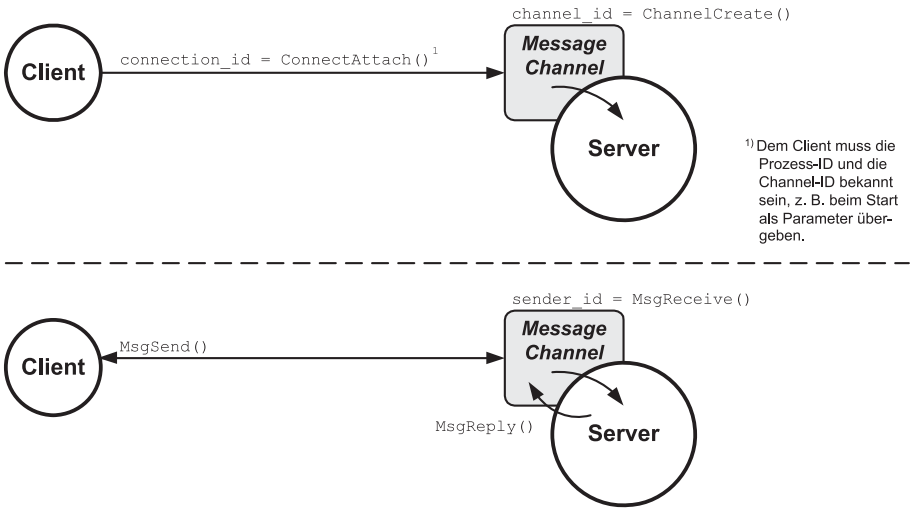


Bild 3.3: Interprozess-Kommunikation mit Message-passing

Bei der Kommunikation mittels Signalen lassen sich asynchrone Abläufe ähnlich wie bei Unterbrechungsanforderungen starten. Ein Prozess kann mit der Funktion `sigaction()` einem Signal einen bestimmten Funktionsaufruf zuordnen. Man spricht dann auch von einem *Handler* für dieses Signal. Alternativ kann im Programmfluss auch auf eines oder mehrere ausgewählte Signale gewartet werden, um sich mit einem anderen Prozess zu synchronisieren. Dazu existiert der Befehl `sigwaitinfo()`:

```
#include <sys/signal.h>

// return: signal_id
int sigwaitinfo (const sigset_t *signale, NULL);
```

Der Funktion muss eine Liste mit Signalen `signale` übergeben werden, auf die gewartet werden soll. Die Programmzeile blockiert nun solange, bis ein Signal aus dieser Liste eintrifft. Es fährt anschließend mit der nächsten Zeile im Code fort. Als Rückgabewert wird die ID des Signals übergeben. Eine vollständige Liste aller Signale im System kann z. B. mit `sigfillset()` erzeugt werden.

```
#include <sys/signal.h>

sigset_t signale;
sigfillset (&signale);
```

Diese Art der Interprozess-Kommunikation bietet sehr vielfältige Möglichkeiten, Prozesse miteinander zu synchronisieren, so stellt das Betriebssystem über sechzig Signal-IDs für verschiedenste Ereignisse zur Verfügung, u. a. gibt es die Signale `SIGUSR1` und `SIGUSR2`, die für den Anwender reserviert sind. Obwohl es noch zahlreiche weitere Funktionen gibt, sollen die vorgestellten ausreichen, um den grundlegenden Mechanismus von Signalen zu verstehen und eine einfache Kommunikation zu realisieren.

3.2 Threads

Die Modularisierung mit parallelen Prozessen lässt sich noch weiter erhöhen, indem ein Prozess wiederum in mehrere parallele Abarbeitungsstränge, sogenannte Threads unterteilt werden. Der Vorteil ist nun, dass Daten zwischen diesen Threads wie globale Variablen ausgetauscht werden können, da sie im selben Speicherbereich liegen. Dabei konkurrieren die Threads genauso um die Betriebsmittel wie zuvor die Prozesse, für sie werden die gleichen Scheduling-Verfahren angewendet.

Ein Thread lässt sich genauso wie eine Funktion schreiben. Mit dem Befehl `pthread_create()` wird er gestartet und läuft parallel zum ursprünglichen Programmfluss. Mit `pthread_join()` ist anschließend eine Synchronisation zwischen beiden möglich. Die folgende Programm-Sequenz möge dem grundsätzlichen Verständnis dienlich sein und soll an dieser Stelle genügen.

```
#include <pthread.h>
int a;
void * thread_function (void * arg) {
  a = 1;
}

int main () {
  pthread_t thread_id;
  a = 2;
  pthread_create(thread_id, NULL, thread_function, NULL);
  // thread l\"auft nun parallel zu main()
```

```
  pthread_join(thread_id, NULL);
  // Warten auf das Ende von thread()
  // Welchen Wert hat jetzt a?
}
```

3.3 Schlossvariablen

Durch den parallelen Zugriff von Threads auf globale Variablen kann es zu Inkonsistenzen kommen. Wird ein Thread während einer Modifikation an einer solchen Variablen von einem anderen unterbrochen, so findet dieser die Variable in einem nicht bekannten Zustand vor. Um solche parallelen Zugriffe zu koordinieren, sind Schlossvariablen geschaffen worden, die als eine abstrakte Datenstruktur implementiert sind. Greift nun ein Thread auf eine globale Variable, die von anderen Threads ebenfalls verwendet wird, zu, so speichert er dies in der Schlossvariablen. Ein anderer Thread muss zunächst diese Schlossvariable prüfen, bevor er auf die eigentliche Variable zugreifen darf, somit wird durch diese Schlossvariablen ein weiterer Synchronisationsmechanismus geschaffen. Es existiert eine Vielzahl solcher Variablen, z. B. Semaphore.

Im vorliegenden Beispiel kann der Wert, den die Variable a am Ende besitzt nicht angegeben werden, da nicht definiert ist, welcher Thread bei gleichen Prioritäten zuerst auf die Variable zugreifen darf.

3.4 Timer

Als letzter wichtiger Punkt sollen Timer erläutert werden. Denn erst somit ist es möglich, Aktionen zu bestimmten Zeitpunkten zu planen, sei es periodisch, z. B. die regelmäßige Abtastung in einer Regelung, oder einmalig, z. B. beim zeitversetzten Eingriff in der Prozessautomatisierung.

Unter QNX sind Timer mit den beiden Funktionen `timer_create()` und `timer_settime()` realisierbar. Der erste Befehl meldet den Timer beim Betriebssystem an, beschreibt die Aktion, die beim Ablauf des Timers gestartet werden soll, z. B. kann ein Signal ausgelöst oder ein paralleler Thread gestartet werden und definiert eine Timer-ID für ihn. Der zweite Befehl stellt die Zeiten des Timers ein und startet ihn anschließend.

```
#include <time.h>
#include <sys/siginfo.h>
```

```
// return: ungleich 0 wenn Fehler
int timer_create (CLOCK_REALTIME, struct sigevent *event, timer_t *timer_id);

// return: ungleich 0 wenn Fehler
int timer_settime (timer_t timer_id, 0,
                   struct itimerspec *zeiteinstellung_des_timers,
                   NULL);
```

Zur Zeiteinstellung des Timers werden die Datenstrukturen `struct timespec` und `struct itimerspec` benötigt. Sie sind wie folgt definiert:

```
#include <time.h>

// Definiert eine Zeit
struct timespec {
  long  tv_sec,     // Zeit in Sekunden
        tv_nsec;    // plus Zeit in Nanosekunden
}
// Definiert die Einstellung des Timers
struct itimerspec {
  struct timespec it_value,       // Zeit, wann der Timer ausloesen soll
                  it_interval;    // Periode fuer regelmaessiges Ausloesen des Timers
                                  // ansonsten 0, wenn nur einmalig.
}
```

Mithilfe der Struktur `struct sigevent` wird festgelegt, welche Aktion beim Auslösen des Timers erfolgen soll. Diese Struktur ist sehr komplex, daher sind zur Vereinfachung einige Makros definiert, mit denen die Initialisierung leicht erfolgen kann. Im einfachsten Fall kann ein Signal ausgelöst werden:

```
#include <sys/siginfo.h>

struct sigevent event;
SIGEV_SIGNAL_INIT (&event, signal_nummer);
```

4 Modellbildung

Das Fahrzeug ist recht einfach aufgebaut, daher gestaltet sich die Modellbildung für das Querverhalten problemlos. Die Geschwindigkeit des FTF wird als konstant bzw. im Verhältnis zur Abstandsdynamik als langsam veränderlich betrachtet. Die Modellbildung kann mithilfe der Abbildung 4.1, die schematisch das FTF mit Leitdraht zeigt, erfolgen.

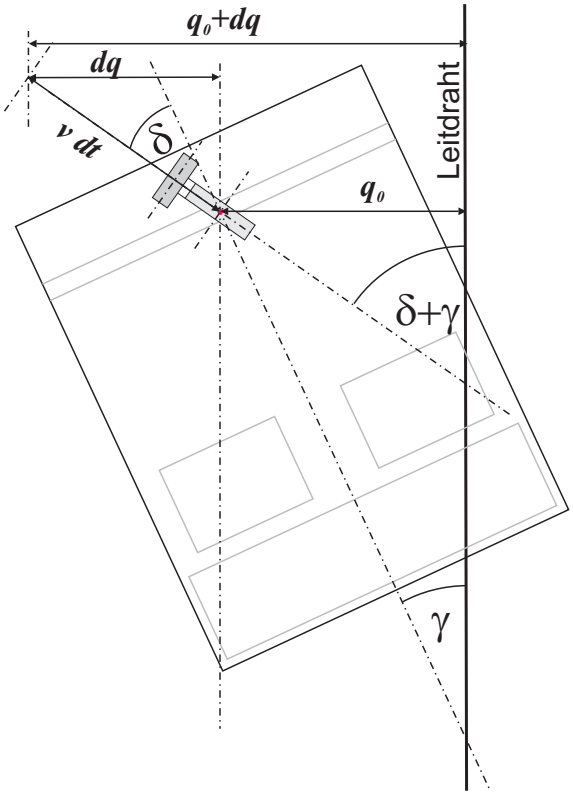


Bild 4.1: Modell der Querkinematik

Der Leitdraht stellt ein Referenzsystem dar, d. h. der Querabstand wird stets senkrecht zu ihm gemessen, er gibt die Orientierung vor. Die Lage dieses Referenzsystem im Weltkoordinatensystem ist dabei völlig ohne Belang. Relativ zum Draht ist das Fahrzeug verdreht, die Fahrzeugmittelachse schließt mit dem Leitdraht den Gierwinkel γ ein. Zusammen mit dem

Lenkwinkel δ ergibt die Summe beider Winkel den Gesamtwinkel zwischen Leitdraht und der Orientierung des Antriebsrades. Alle Winkel werden im mathematisch positiven Sinn gezählt, das bedeutet, dass die im Bild eingezeichneten Winkel alle positiv sind[1]. Betrachtet man nun den kurzen Zeitraum dt, so ergibt sich für den Querabstand q der folgende Zusammenhang:

$$dq = v \sin(\delta + \gamma) \cdot dt \tag{4.1}$$

$$\Leftrightarrow \frac{dq}{dt} = \dot{q} = v \sin(\delta(t) + \gamma(t)) \tag{4.2}$$

Integriert man diese Gleichung, wobei Lenk- und Gierwinkel als konstant über den Integrationszeitraum angenommen werden und berücksichtigt ferner den anfänglichen Querabstand, so erhält man die gesuchte Beschreibung für das Querverhalten:

$$q(t) = q_0 + \int_{\tau=0}^{t} v \sin(\delta + \gamma)\, d\tau \tag{4.3}$$

Der Gierwinkel γ wirkt hierbei als Störgröße, da eine Änderung das Fahrzeug aus der Spur führt. Man kann also eine gekrümmte Verlegung des Leitdrahts als eine Störung interpretieren. Ziel der Regelung ist der stetige Abgleich auf den Querabstand null durch Nachführung des Fahrzeugs entlang des Sollspur.

Mit der gefundenen Gleichung lässt sich das nichtlineare Blockschaltbild zeichnen und das Übertragungsverhalten der Querbewegung verdeutlichen. Dabei ist ein Proportionalitätsfaktor eingefügt, der den Winkel von Grad ins Bogenmaß überführt, was im weiteren Verlauf bei der Linearisierung wichtig ist. Die als konstant angenommene Geschwindigkeit kann

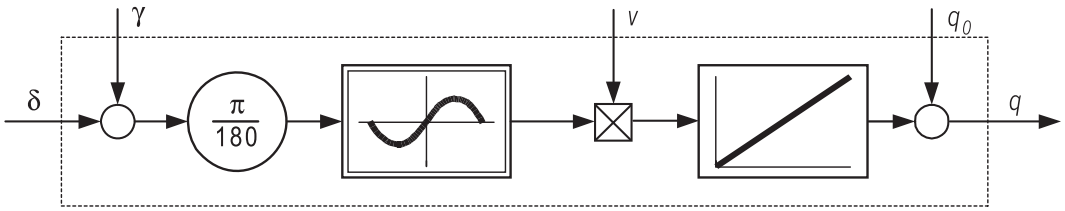


Bild 4.2: Blockschaltbild der Querkinematik

vereinfacht ebenfalls als Verstärkung angenommen und dadurch die Nichtlinearität des Produktgliedes eliminiert werden. Des Weiteren ist es offensichtlich, dass im geregelten System

[1]Der Gierwinkel wird immer vom Referenzsystem ausgehend gemessen. Die Vorzeichenfestlegung der Richtung gilt auch für den Lenkmotor, der bei einem positivem Tastverhältnis die Antriebseinheit mathematisch positiv dreht, bei negativen Tastverhältnis folglich entgegengesetzt.

das Fahrzeug der Spur folgt, also die Summe aus Lenk- und Gierwinkel $\delta + \gamma$ sehr klein sein wird. Die Sinus-Funktion kann somit recht einfach linearisiert werden, indem sie durch ihr Argument im Bogenmaß ersetzt wird. Das gesamte, linearisierte Übertragungsverhalten entspricht also einem einfachen Integrator.

Normiert man nun die Gleichung auf den Querabstand q_n, so folgt für $q_0 = 0$ die linearisierte Gleichung zu:

$$\frac{q}{q_n} = \frac{\pi\, v}{180 q_n} \int_{\tau=0}^{t} \delta\, d\tau + \underbrace{d(t)}_{\frac{\pi\, v}{180 q_n}\,\gamma\, t} \tag{4.4}$$

Die Integrationszeitkonstante lautet dann:

$$T_3 = \frac{180 q_n}{\pi\, v} \tag{4.5}$$

und die Querdynamik lässt sich mit der Übertragungsfunktion eines Integrators beschreiben:

$$G(s) = \frac{1}{T_3\, s} \tag{4.6}$$

Da die Dynamik des Stellgliedes Lenkmotor nicht zu vernachlässignen ist, wird sie der Strecke zugeordnet. Bekanntermaßen kann ein Motor in seiner Übertragungsfunktion von Spannung zu Winkel mit einem IT1-Glied charakterisiert werden, sodass die Dynamik der Strecke durch das Gesamtblockschaltbild in Abbildung 4.3 beschrieben werden kann. Die Gesamtübertragungsfunktion von PWM-Verhältnis u_{LM} als Eingang in den Lenkmotor bis zum Querabstand $\frac{q}{q_n}$ lautet dann:

$$G(s) = \frac{\frac{q}{q_n}}{u_{FM}} = \underbrace{\frac{1}{(T_1 s + 1)\; T_2 s}}_{Lenkmotor} \cdot \underbrace{\frac{1}{T_3 s}}_{Querdynamik} \tag{4.7}$$

Die Zeitkonstanten T_1 und T_2 sind aus einer Identifikation bekannt. Der Abstandssensor misst in Millimetern. Aus diesem Grund wird der Normierungsabstand q_n zu 1mm gewählt. Die Zeitkonstante T_3 hängt von der gefahrenen Geschwindigkeit ab, es zeigt sich aber aus Praxisversuchen, dass eine Annahme von $v = 0{,}1\,\mathrm{ms}^{-1}$ das gesamte Geschwindigkeitsspektrum

sehr gut abdeckt. Die Zeitkonstanten betragen somit:

$$
\begin{aligned}
T_1 &= 57{,}9\text{ms} \\
T_2 &= 0{,}7296\text{s} \\
T_3 &= 0{,}573\text{s}
\end{aligned}
$$

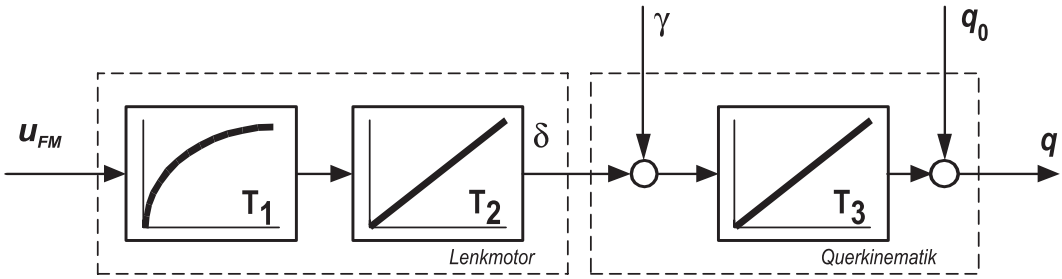


Bild 4.3: Modell der Querbewegung in einer Kurve

Der Abstandssensor ist etwa 15cm vor dem Drehpunkt der Antriebseinheit angebracht. Dadurch wird der Abstand an einem Punkt gemessen, den der Drehpunkt erst kurze Zeit später erreichen wird. Dies wirkt vorsteuernd auf die Regelung, muss aber nicht weiter berücksichtigt werden. Das Prinzip lässt sich etwa als eine imaginäre Deichsel interpretieren, an der das Antriebsrad auf dem Leitdraht entlang gezogen wird.

Die Längsdynamik zur Auslegung eines Geschwindigkeitsregler soll nicht Teil dieses Labors sein und wird daher hier nicht betrachtet. Denn für die Versuchsdurchführung reicht eine Steuerung durch eine feste Vorgabe des Tastverhältnisses des PWM-Signals aus.

5 Versuchsdurchführung

5.1 Hausaufgabe: Reglerimplementierung

Ein PI-Regler besitzt bekanntlich die kontinuierliche Übertragungsfunktion:

$$K(s) = \frac{u}{e} = V_{pi} \frac{T_i s + 1}{T_i s}$$

Da die Regelung aber aufgrund der Abtastung zeitdiskret ist, muss auch der Regler zeitdiskret sein. In einer ersten Näherung kann man bei einer Abtastperiode T die folgende Substitution durchführen:

$$s = \frac{2}{T} \frac{1 - z^{-1}}{1 + z^{-1}}$$

Hausaufgabe 1: Setzen Sie diese Substitution in die Übertragungsfunktion ein und lösen Sie sie nach der Stellgröße u auf. Wie sieht nun eine Programmsequenz aus, die einen zeitdiskreten PI-Regler implementiert? Geben Sie diese in Pseudo-Code an.

Hinweis: Durch den Ausdruck z^{-1} wird auf alte Werte aus vorhergehenden Abtastungen zurückgegriffen, z. B. bedeutet

$$u\, z^{-1} = u(t - T) \qquad \text{und} \qquad u\, z^{-2} = u(t - 2T),$$

dass zum aktuellen Zeitpunkt t auch die berechneten Stellgrößen aus der letzten $(t - T)$ und vorletzten Abtastung $(t - 2T)$ einfließen.

In der Praxis wird häufig der PI-Regler leicht modifiziert verwendet. Das Blockschaltbild in der Abbildung 5.1 zeigt diese etwas veränderte Form, die häufig etwas bessere Regelungsergebnisse liefert.

Hausaufgabe 2: Zeigen Sie, dass es sich tatsächlich um einen PI-Regler handelt, indem Sie die Differenzengleichung für die Stellgröße berechnen. Vernachlässigen Sie dazu den nichtlinearen Einfluss der Begrenzungen.

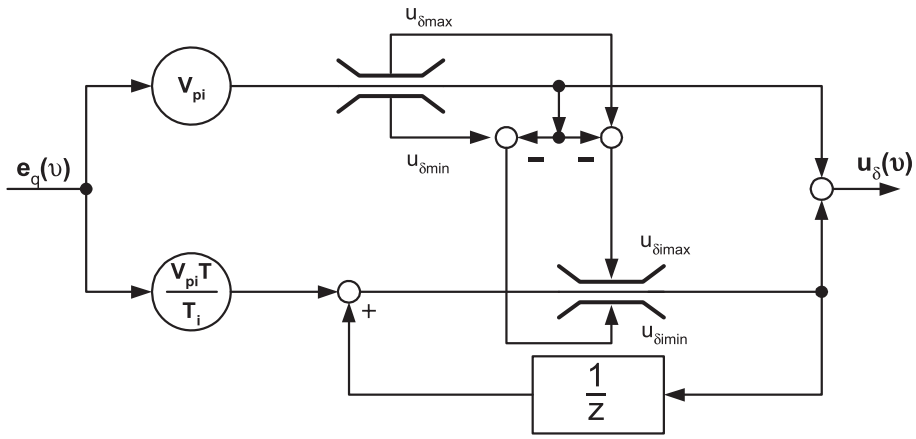


Bild 5.1: Blockschaltbild des diskreten PI-Querreglers

Hausaufgabe 3: Der PI-Regler soll nun in Pseudo-Code implementiert werden. Wie lautet die Programmsequenz unter Berücksichtigung der Stellgrößenbegrenzung für das in der Abbildung 5.1 dargestellte Blockschaltbild?

Hausaufgabe 4: Machen Sie sich mit der Reglerauslegung im Kapitel 5.3 vertraut und berechnen Sie die Kaskadenregelung entsprechend der gegebenen Anleitungsschritte.

5.2 Interprozess-Kommuniaktion und Hardware-Zugriff unter QNX

Im ersten Teil des Labors sollen zunächst die Mechanismen eines Echtzeitbetriebssytems anhand von QNX, wie sie entsprechend dem einführenden Kapitel erläutert worden sind, verdeutlicht werden. Nachdem diese vertraut sind, kann im Anschluss der Zugriff auf die Peripherie erfolgen und die Motoren und Sensoren in Betrieb genommen werden. Mit den Kenntnissen ist es dann bereits möglich, eine einfache Regelung zu implementieren und das FTF fahren zu lassen.

Starten Sie zur Versuchsdurchführung den Terminal- und den FTF-Rechner und loggen Sie sich anschließend ein:

- Terminal-Rechner
  Login: `labor`
  Passwort: `ftf`

- FTF-Rechner
  Login: `root`
  Passwort: `ftf`

Der FTF-Rechner bootet nicht automatisch in die grafische Oberfläche Photon. Diese kann aber bei Bedarf mit dem Befehl `ph` gestartet werden. Auf beiden Rechner existiert ein Verzeichnis `/net/`. In diesem Verzeichnis tauchen unter `irt52` (Terminal-Rechner) und `irt55` (FTF-Rechner) die jeweiligen Verzeichnisstrukturen der beiden beteiligten Rechner auf. Mit dem Befehl `dir` oder `l` kann die Verzeichnisstruktur angezeigt und mit `cd` manövriert werden.

Überprüfen Sie zunächst, ob die Netzwerkverbindung zwischen beiden Rechnern besteht. Öffnen Sie dazu auf dem Terminal-Rechner ein Eingabefenster und pingen Sie den FTF-Rechner mit dem Befehl `ping irt55` an.

Wechseln Sie anschließend auf dem Terminal-Rechner in ihr Home-Verzeichnis `/home/labor/` und richten Sie die Labor-Umgebung durch Aufruf des Skripts `labor` für die Versuchsdurchführung ein.[1]

5.2.1 Interprozess-Kommunikation

Wechseln Sie zunächst mit dem Befehl `cd` auf dem Terminal-Rechner in das Verzeichnis `/home/labor/ipc/`. Normalerweise befinden Sie sich beim Öffnen eines Terminals bereits in Ihrem Home-Verzeichnis `/home/labor/`. Zur Erleichterung der Programmierung existieren bereits Vorlagen, die Sie um die entsprechenden Funktionen ergänzen müssen. Die Quelldateien können durch den Befehl `make` übersetzt und gelinkt werden. Der zugrunde liegende Makefile beinhaltet zur Vereinfachung sämtliche Compiler- und Linkeranweisungen.

Aufgabe 1: Programmieren Sie einen Server-Prozess zur Berechnung der Quadratwurzel. Dazu soll ein Client eine Anfrage an den Server richten, indem er ihm eine positive Zahl schickt. Der Server antwortet daraufhin mit dem Ergebnis. Vergegenwärtigen Sie sich die Programmzeilen der beiden Vorlagen `client.c` und `server.c` und besprechen Sie untereinander, was sie jeweils bewirken. Ergänzen Sie die Vorlagen anschließend um die noch fehlenden Anweisungen.

1. Wechseln Sie in das Verzeichnis `/home/student/labor/ipc/`.

[1] Die Einrichtung der Laborumgebung erfolgt sowohl auf dem Terminal- als auch auf dem FTF-Rechner über die exisitierende Netzwerkverbindung!

2. Starten Sie die integrierte Entwicklungsumgebung *Workspace* über die Schnellstartleiste am rechten Bildschirmrand oder den Befehl `ws`. Alternativ können Sie auch die Editoren `ped` oder `jed` benutzen.
3. Öffnen Sie die Dateien `client.c` und `server.c` und ergänzen Sie die fehlenden Programmzeilen.
4. Compilieren und linken Sie die beiden Programme anschließend mit dem Befehl `make`.
5. Sollten beide Programme einwandfrei übersetzt worden sein, so öffnen Sie ein zweites Terminal, z. B. mit dem Befehl `terminal &`[2], korrigieren Sie anderenfalls Ihren Quellcode.
6. Wechseln Sie nun im zweiten Terminal in das gleiche Verzeichnis und starten Sie den Server mit `server`.
7. Starten Sie im ersten Terminal den Client `client` und geben Sie die nötigen Parameter zum Aufbau der Kommunikation ein. Verdeutlichen Sie sich dazu die jeweilige Bedeutung.
8. Beobachten Sie den Ablauf der Interprozess-Kommunikation. Beenden Sie anschließend die beiden Programme mit der Tastenkombination `STRG+C`.

5.2.2 Interprozess-Kommunikation über ein Netzwerk

In einer Erweiterung der vorhergehenden Aufgabe wird nun gezeigt, wie einfach eine Kommunikation auch über ein Netzwerk erfolgen kann und dass sich für den Anwender prinzipiell kein Unterschied zwischen einer netzwerkweiten und einer auf einem einzelnen Rechner beschränkten Kommunikation ergibt.

Aufgabe 2: Die Interprozess-Kommunikation soll netzwerkweit erfolgen. Ergänzen Sie dazu die Vorlage `client.c`. Sie können sich dazu an der Lösung der vorherigen Aufgabe orientieren:

1. Kopieren Sie zunächst den Prozess `server` aus der vorherigen Aufgabe in das Verzeichnis `/tmp/` auf dem FTF-Rechner, z. B. mithilfe des Befehls `cp`:
   `cp /net/irt52/home/labor/ipc/server /net/irt55/tmp/server`
2. Wechseln Sie in das Verzeichnis `/home/labor/ipc-network/` und editieren Sie die Vorlage.

[2] Das &-Zeichen besagt lediglich, dass nicht auf das Ende des Prozesses gewartet werden muss. Führen Sie zum Vergleich den Befehl einmal ohne das &-Zeichen aus. Dies ist auch eine Form der Interprozess-Kommunikation!

3. Compilieren und linken Sie den Client mit `make`.
4. Wurde der ausführbare Client fehlerfrei erzeugt, so starten Sie auf dem FTF-Rechner den Server `server`.
5. Starten Sie auf dem Terminal-Rechner den Client `client` und geben Sie die benötigten Kommunikationsparameter ein.
6. Führen Sie mehrere Anfragen des Clients aus und beobachten Sie die Ausgabe des Servers auf dem FTF-Rechner.
7. Beenden Sie wieder beide Programme mit der Tastenkombination `STRG+C`.

Aufgabe 3: Erkennen Sie die Veränderungen, die gegenüber der vorherigen Aufgabe am Code vom Client- und Server-Prozess durchgeführt werden mussten? Diskutieren Sie diese Modifikationen in Ihrer Gruppe.

5.2.3 Timer-Programmierung

Eine grundlegende Voraussetzung für eine zeitdiskrete Regelung ist das Abtasten zu äquidistanten Zeitpunkten. Dazu muss ein Timer programmiert werden. Wechseln Sie auf dem Terminal-Rechner in das Verzeichnis `/home/labor/timer/` und ergänzen Sie die Vorlage entsprechend der Aufgabenstellung.

Aufgabe 4: Programmieren Sie einen Timer, der in periodischen Abständen das Signal `SIGUSR1` generiert. Erzeugen Sie eine regelmäßige Ausgabe auf dem Bildschirm und verifizieren Sie anschließend Ihre Lösung.

1. Wechseln Sie in das Verzeichnis `/home/labor/timer/` und ergänzen Sie die Vorlage `timer.c`.
2. Compilieren und linken Sie Ihr Programm mit `make`.
3. Starten Sie Ihr Programm und überprüfen Sie die zeitäquidistante Ausgabe, indem Sie die eingebaute Stoppuhr in der Vorlagendatei benutzen und die Periodendauer in verschiedenen Iterationsschritten anpassen.

Aufgabe 5: Wie kann nun der Quellcode für eine einfache Regelung aussehen? Schreiben Sie ein Programm in Pseudo-Code. Gehen Sie dabei nicht zu sehr ins Detail, sondern benutzen Sie Anweisungen im Sinne von z. B.:
`Regleralgorithmus: Berechnung der neuen Stellgröße`.

5.2.4 Zugriff auf die Peripherie

Nachdem Sie die allgemeine Grundstruktur einer Regelung diskutiert haben, sollen in einem vorbereitenden Schritt zunächst die Motoren und Sensoren angesteuert werden. Über das IP-Modul, das aus einem programmierbaren Logikbaustein mit DA-/AD-Wandlern besteht und als Erweiterungskarte auf dem FTF-Rechner die I/O-Verbindung realisiert, wird auf die Hardware zugegriffen. Damit Sie sich mit der eigentlichen Programmierung der Hardware-Ansteuerung nicht beschäftigen müssen, wurde eine Klasse implementiert, die die benötigten Funktionen zur Verfügung stellt.

```
#include <IP_Modul>
#include <SysDef_FTF.h>

cIP_Modul::cIP_Modul (IP_MODUL_SLOT)
```

Die Klasse ruft bei der Instanziierung automatisch ihren Konstruktor auf, der die nötigen Initialisierungen durchführt. Mit den folgenden Member-Funktionen erfolgt der entsprechende Zugriff recht einfach.

```
float cIP_Modul::messeGeschwindigkeit ();
float cIP_Modul::messeLenkwinkel ();
cIP_Modul::steuerDrehzahlMotor ({FAHRMOTOR | LENKMOTOR}, float tastverhaeltnis);
```

Die ersten beiden Funktionen erklären sich von selbst, die dritte steuert die beiden Motoren an, wobei der jeweilige Motor angegeben werden muss. Das Tastverhältnis des PWM-Signals kann im Bereich von -100%..+100% liegen. Ein positiver Wert entspricht einer Vorwärtsfahrt bzw. einer Rechtsdrehung der Lenkung, ein negativer Wert wirkt entsprechend entgegengesetzt.

Aufgabe 6: Analog zu ihrem Pseudo-Code soll nun zunächst eine Steuerung implementiert werden. Dazu soll der Fahrmotor langsam vorwärts und rückwärts drehen. Die Wechsel sollten nicht abrupt, sondern idealerweise oszillierend erfolgen. Gleichzeitig soll der Lenkmotor die Antriebseinheit hin und her schwenken[3]. Geben Sie zu regelmäßigen Zeitpunkten die Messwerte für Lenkwinkel und Geschwindigkeit aus.[4]

[3] Für den Lenkmotor bietet es sich in diesem Versuch an, die Amplitude auf das Intervall $[-2{,}5; 2{,}5]$ (Angabe in Prozent) zu beschränken.

[4] Dies sollte nicht in jeder Abtastperiode, sondern nur etwa jede zehnte geschehen, da sonst zuviel Rechenperformance verloren geht und die Abtastintervalle evtl. nicht mehr eingehalten werden können. Außerdem ist die Ausgabe sonst auch viel zu schnell für eine Beobachtung.

1. Starten Sie die grafische Oberfläche Photon auf dem FTF-Rechner mit `ph`.
2. Öffnen Sie ein Terminal und wechseln Sie in das Verzeichnis `/usr/ftf/labor/motoren/`.
3. Vergleichen Sie Ihren Pseudo-Code mit der Vorlage `motoren.cpp`. Handelt es sich bei der Vorlage um eine Steuerung oder Regelung? Begründen Sie Ihre Antwort.
4. Ergänzen Sie die Vorlage um die fehlenden Hardware-Zugriffe und Programmzeilen.
5. Compilieren und linken Sie Ihr Programm mit `make`.
6. **Bocken Sie das FTF mithilfe des kleinen roten Kunststoffblocks auf!**
7. **Überprüfen Sie, dass die Antriebseinheit etwas in der Luft hängt!**
8. **Fahren Sie erst fort, wenn Sie sich davon überzeugt haben!**
9. Starten Sie den Antriebsstromkreis durch Drücken des Start-Buttons. Ein deutliches Anziehen des Schützes muss durch ein Klacken hörbar sein. Ist dies nicht der Fall, so ist wahrscheinlich der Not-Aus-Schalter noch eingerastet. Entrasten Sie ihn durch ein leichtes Drehen nach links.
10. Starten Sie ihr Programm, beobachten Sie die Motoren und vergleichen Sie das Verhalten mit der Ausgabe auf dem Bildschirm.
11. Stoppen Sie ihr Programm mit `STRG+C`.

Hinweis: Sollten sich nach Abbruch Ihres Programms die Motoren noch drehen, obwohl ihr Programm nicht mehr aktiv ist, so können Sie mit dem Befehl `stoppe` die Motoren anhalten und neu ausrichten.

5.2.5 Implementierung einer einfachen Regelung

Im nächsten Schritt soll nun eine einfache Regelung mit einem Proportional-Glied erfolgen. Dazu muss der Abstand der Lenkeinheit vom Leitdraht gemessen werden. Dies kann mit der folgenden Klasse geschehen:

```
#include <Mikrolenksensoren.h>
#{\'\i}nclude <SysDef_FTF.h>

cMikrolenksensoren::cMikrolenksensoren (false);
```

Die Member-Funktion `messeAbstand()` misst den Abstand des vorderen und hinteren Mikrolenksensors vom Leitdraht in Millimeter, wobei für Vorwärtsfahrt der hintere Sensor nicht interessiert und nicht weiter berücksichtigt werden soll.

```
cMikrolenksensoren::messeAbstand (float *abstand_vorne, float *abstand_hinten);
```

Aufgabe 7: Erweitern Sie nun Ihr Steuerungsprogramm aus der vorhergehenden Aufgabe zu einer Regelung. Benutzen Sie dazu die gegebene Vorlage bzw. greifen Sie auf Ihre Lösung aus der vorhergehenden Aufgabe zurück. Als Regler verwenden Sie einen einfachen P-Regler mit der Verstärkung:

$$V = 2,5$$

Die Längsbewegung soll ungeregelt bleiben, stellen Sie das Tastverhältnis des Antriebsmotors auf maximal 65 Prozent ein.

1. Wechseln Sie auf dem FTF-Rechner in das Verzeichnis `/usr/ftf/labor/regelung/`.
2. Verifizieren Sie den Code der Vorlage `regelung.cpp`. Handelt es sich um eine Regelung? Ergänzen Sie anschließend die noch fehlenden Programmzeilen.
3. Compilieren und linken Sie Ihr Programm mit `make`.
4. Richten Sie das FTF so aus, dass es an der Vorderseite mittig über dem Leitdraht platziert ist.
5. **Bocken Sie das FTF mithilfe des kleines roten Kunststoffblocks auf!**
6. **Überprüfen Sie, dass die Antriebseinheit etwas in der Luft hängt!**
7. **Fahren Sie erst fort, wenn Sie sich davon überzeugt haben!**
8. Starten Sie den Antriebsstromkreis durch Drücken des Start-Buttons.
9. Schalten Sie die Stromversorgung für den Leitdraht ein und stellen Sie einen Wechselstrom von 30mA bei einer Frequenz von 5,2kHz an der Konstantstromquelle ein.
10. Starten Sie Ihr Programm und beobachten Sie die Lenkbewegung. Wird die Antriebseinheit immer über dem Leitdraht ausgerichtet, wenn Sie das FTF seitlich etwas hin und her schieben?

Ist dies nicht der Fall, überprüfen Sie Ihren Code nochmals. Ansonsten können Sie nun das Fahrzeug fahren lassen. Stoppen Sie dazu Ihr Programm mit `STRG+C`. Sollten die Motoren noch laufen, so halten Sie sie mit dem Befehl `stoppe` an. Fahren Sie nun wie folgt fort:

1. Schließen Sie die grafische Oberfläche über den Launch-Button in der linken unteren Hälfte: `Shutdown Photon Session`.
2. Stellen Sie vom Terminal-Rechner aus eine Telnet-Verbindung her: `telnet irt55` und loggen Sie sich ein.
3. Wechseln Sie in das Verzeichnis `/usr/ftf/labor/regelung/`.
4. Entfernen Sie den kleinen roten Aufbockblock vom FTF, sowie sämtliche Kabel und Anschlüsse.
5. Schalten Sie den Antriebsstromkreis ein, falls dieser abgeschaltet wurde.
6. **Da in dieser Regelung noch keinerlei Sicherheitsabschaltungen eingebaut sind, sollte eine Person mit dem FTF mitgehen und eine Hand über dem Not-Aus-Button halten, falls eine kritische Situtation eintritt.**
7. Rufen Sie nun Ihr Regelungsprogramm auf: `regelung`
8. Halten Sie das FTF an, nachdem es einige Zeit gefahren ist, indem Sie Ihr Programm beenden.

Das Prinzip der Regelung sollte an dieser Stelle klar geworden sein. Jedoch ist der eingebaute Regler noch sehr rudimentär. Bei einer höheren Geschwindigkeit ist er nicht mehr in der Lage, das Fahrzeug auf dem Draht zu führen. Aus diesem Grund soll im nächsten Teil des Versuchs eine bessere Regelung entwickelt werden.

Aufgabe 8: Bei genauer Betrachtung ist es erstaunlich, dass bereits ein einfacher P-Regler in der Lage ist, das FTF auf dem Leitdraht zu führen. Denn anhand der durchgeführten Modellbildung lässt sich die Strecke mit einem verzögerten doppelten Integrator beschreiben. Aus den Grundlagen der Regelungstechnik ist aber bekannt, dass ein P-Regler niemals in der Lage ist, eine solche Strecke (I2T) zu stabilisieren. Warum gelingt es trotzdem? Untersuchen Sie dazu das Verhalten des FTF bei hoher Geschwindigkeit, **indem Sie das Fahrzeug wieder aufbocken** und das Tastverhältnis für das PWM-Signal auf hundert Prozent setzen.

5.3 Reglerentwurf

Die Übertragungsfunktion der Querbewegung lautet entsprechend der Modellbildung:

$$G(s) = \frac{\frac{q}{q_n}}{u_{FM}} = \underbrace{\frac{1}{(T_1 s + 1)\ T_2 s}}_{Lenkmotor} \cdot \underbrace{\frac{1}{T_3 s}}_{Querdynamik} \tag{5.1}$$

Mit den Zeitkonstanten:

$$\begin{aligned} T_1 &= 57{,}9\text{ms} \\ T_2 &= 0{,}7296\text{s} \\ T_3 &= 0{,}573\text{s} \end{aligned}$$

Es wird eine Kaskadenstruktur gewählt. Die innere Schleife regelt den Lenkwinkel, die äußere den Querabstand. Die folgende Abbildung 5.2 stellt diese Struktur dar und zeigt die beiden vorgeschlagenen Regler.

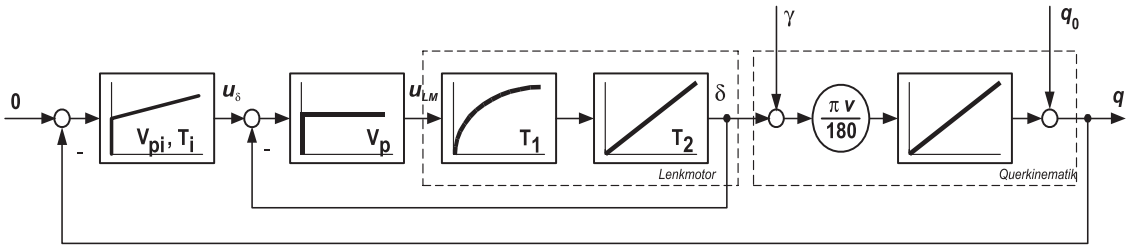


Bild 5.2: Kaskadenstruktur der Regelung

Im ersten Schritt wird die innere Kaskade ausgelegt. Der P-Regler arbeitet auf einer IT1-Strecke. Durch den Integrator ist der geschlossene Kreis im Führungsverhalten sogar stationär genau, jedoch nicht im Störübertragunsverhalten. Dies muss im Anschluss von der äußeren Kaskade übernommen werden.

Aufgabe 9: Bestimmen Sie zunächst die Übertragungsfunktion des inneren geschlossenen Kreises und berechnen Sie anschließend durch einen Koeffizientenvergleich mit der Normalform eines PT2 die Verstärkung des P-Reglers für eine Dämpfung von D=1.

Die innere Kaskade wird nun implementiert.

Aufgabe 10: Modifizieren Sie Ihr Regelungsprogramm aus dem vorherigen Kapitel so, dass die innere Kaskade implementiert wird.

1. Geben Sie eine beliebige Sollvorgabe für den Lenkwinkel vor. Idealerweise programmieren Sie eine Rechteckfunktion (T$\approx 10s$), sodass der Lenkwinkel laufend wechselt.

2. Programmieren Sie die innere Kaskade, indem Sie den Lenkwinkel messen und den Regelfehler berechnen. Zur Verifikation geben Sie den aktuellen Lenkwinkel und den Sollwinkel aus.
   `float IP_Modul::messeLenkwinkel();`

3. Compilieren und linken Sie Ihr Programm mit `make`.
4. **Bocken Sie das FTF mithilfe des kleines roten Kunststoffblocks auf.**
5. **Überprüfen Sie, dass die Antriebseinheit etwas in der Luft hängt!**
6. **Fahren Sie erst fort, wenn Sie sich davon überzeugt haben!**
7. Starten Sie den Antriebsstromkreis durch Drücken des Start-Button.
8. Starten Sie Ihr Programm und beobachten Sie die Lenkbewegung. Folgt der Lenkwinkel der Sollvorgabe.

Im zweiten Schritt wird nun die äußere Kaskade ausgelegt.

Aufgabe 11: Approximieren Sie den inneren geschlossenen Kreis durch ein PT1-Glied. Legen Sie anschließend den PI-Regler für die sich ergebende IT1-Strecke nach dem symmetrischen Optimum bei einer Dämpfung von D=1 aus.

Aufgabe 12: Erweitern Sie Ihr Regelungsprogramm um die äußere Kaskade.

1. Fügen Sie den PI-Regler entsprechend dem Blockschaltbild aus Abbildung 5.1 in Ihr Programm ein und parametrieren Sie ihn mit den Ergebnissen Ihrer Rechnung.
2. Compilieren und linken Sie Ihr Programm mit `make`.
3. **Bocken Sie das FTF mithilfe des kleines roten Kunststoffblocks auf!**
4. **Überprüfen Sie, dass die Antriebseinheit etwas in der Luft hängt!**
5. **Fahren Sie erst fort, wenn Sie sich davon überzeugt haben!**
6. Starten Sie den Antriebsstromkreis durch Drücken des Start-Buttons.
7. Schalten Sie die Stromversorgung für den Leitdraht ein und stellen Sie einen Wechselstrom von 30mA bei einer Frequenz von 5,2kHz an der Konstantstromquelle ein.
8. Starten Sie Ihr Programm und beobachten Sie die Lenkbewegung. Folgt die Antriebseinheit auch dem Leitdraht, wenn Sie das FTF etwas hin- und herschieben?

Ist dies der Fall, so können Sie das FTF erneut auf dem Leitdraht fahren lassen. Stoppen Sie dazu zunächst das Programm.

1. Schließen Sie die grafische Oberfläche über den Launch-Button in der linken unteren Hälfte: `Shutdown Photon Session`.
2. Stellen Sie vom Terminal-Rechner aus eine Telnet-Verbindung her: `telnet irt55` und loggen Sie sich ein.

3. Wechseln Sie in das Verzeichnis `/usr/ftf/labor/regelung/`.
4. Entfernen Sie den kleinen roten Aufbockblock vom FTF, sowie sämtliche Kabel und Anschlüsse.
5. Schalten Sie den Antriebsstromkreis ein, falls dieser abgeschaltet wurde.
6. **Da in dieser Regelung noch keinerlei Sicherheitsabschaltungen eingebaut sind, sollte eine Person mit dem FTF mitgehen und eine Hand über dem Not-Aus-Button halten, falls eine kritische Situtation eintritt.**
7. Rufen Sie nun Ihr Regelungsprogramm auf: `regelung`
8. Halten Sie das FTF an, nachdem es einige Zeit gefahren ist, indem Sie Ihr Programm beenden.

Die Regelung stabilisiert das FTF auf dem Leitdraht. Allerdings neigt das Fahrzeug sehr stark zum Schwingen bzw. die Regelung ist relativ schlecht. Dies liegt an der vollständigen Vernachlässigung der Zeitdiskretisierung bei der Reglerauslegung. Dies soll im folgenden Kapitel noch eingehender betrachtet werden, jedoch eine vollständige zeitdiskrete Reglerauslegung nicht mehr durchgeführt werden.

5.4 Versuchsfahrten mit zeitdiskreter Reglerauslegung

Das vorherige Kapitel hat gezeigt, dass ein kontinuierlicher Regler nicht ohne weiteres für eine zeitdiskrete Regelung verwendet werden kann. Insbesondere zeigt sich dies, wenn Rechenlaufzeiten und Sensorkommunikation nicht mehr gegenüber der Abtastperiode wie in diesem Fall vernachlässigbar sind. Diese müssen daher bei der Reglerauslegung berücksichtigt werden. Die Struktur und die Reglertypen K_1 und K_2 werden beibehalten. Lediglich die Reglerauslegung erfolgt nun auf der zu diskretisierten Strecke. Auf die Darstellung der einzelnen zu berücksichtigenden Effekte und Berechnungsschritte soll verzichtet werden, da sie den Rahmen des Labors deutlich übersteigen würden.

Man erhält für den inneren P- und den äußeren PI-Regler folgendes Ergebnis:

$$K_1(s) = 2,7144 \tag{5.2}$$

$$K_2(s) = 0,7105 \cdot \frac{2,4191s + 1}{2,4191s} \tag{5.3}$$

Überprüfen Sie die Lösung, indem Sie das FTF mit diesen Parametern in Betrieb nehmen. Starten Sie dazu auf dem Terminal-Rechner das Terminalprogramm, das über eine WLAN-Verbindung mit dem Regelungsprogramm auf dem FTF-Rechner kommuniziert. Gehen Sie dabei wie folgt vor:

1. Richten Sie das FTF mittig auf dem Leitdraht aus.
2. Schließen Sie auf dem FTF-Rechner die grafische Oberfläche über den Launch-Button in der linken unteren Hälfte: `Shutdown Photon Session`.
3. Optional können Sie sich ausloggen.
4. Starten Sie den Antriebsstromkreis durch den Start-Taster und vergewissern Sie sich, dass im Leitdraht ein Strom von 30mA bei 5,2kHz fließt.
5. Schließen Sie alle Programme auf dem Terminal-Rechner.
6. Starten Sie nun auf dem Terminal-Rechner das FTF-Terminal über den ftf-Button am rechten Bildschirmrand.
7. Wählen Sie den Kaskaden-Regler im Menü `Regelung` und stellen Sie die Parameter entsprechend der Vorgabe ein.
8. Klicken Sie auf den Start-Button oder `F5` und bestätigen Sie den Start des FTF.
9. Es kann bis zu einer Minute dauern bis sich die beiden Rechner synchronisiert haben.
10. Das FTF sollte anfahren.
11. Über den `Status`-Button oder `F6` kann das FTF während der Fahrt überwacht und beeinflusst werden.
12. Untersuchen Sie z. B. den Einfluss der Ultraschallsensoren und der Geschwindigkeitsvorgabe.
13. Stoppen Sie das Fahrzeug über den `Stopp`-Button oder `F6`.