Studienarbeit

Neuronale Echtzeitregelung eines doppeltinversen Pendels

Teil A

Simulation, Steuerung und Programm

Marius Heuler

geboren 3.11.72 in Werneck

heuler@informatik.uni-wuerzburg.de

Angefertigt am
Lehrstuhl für verteilte Systeme (Informatik III)
Bayerische Julius–Maximilians–Universität Würzburg

Betreuer: Prof. Dr.–Ing. P. Tran–Gia,
Dipl.–Inform. K. Tutschku
Dipl.–Inform. R. Müller

Inhaltsverzeichnis

1. Einführung

1.1. Thema

In dieser Arbeit geht es um ein Standardproblem der nichtlinearen Regelungstechnik, welches dazu dienen soll, verschiedene Verfahren und Algorithmen zur Steuerung zu vergleichen. Die Aufgabe der Regelung ist das Balancieren eines senkrecht stehenden Stabes, inverses Pendel genannt. Dieser Stab ist mittels eines Gelenkes drehbar auf einem Träger befestigt. Meist kann sich der Stab nur in einer Achse bewegen. Das Ziel der Regelung ist es nun, durch Kraftwirkung auf den Träger den Stab senkrecht zu halten, d.h. das inverse Pendel zu balancieren. Als nichtlinearer Regler können dabei die unterschiedlichsten Verfahren verwendet werden, wie z.B. Fuzzy-Controller oder neuronale Regler. Vergleichen kann man z.B. wie lange das Pendel durch den Regler senkrecht gehalten wird oder wie „gut“ geregelt wird. Als Güte kann man etwa die Abweichung des Pendels von der Senkrechten oder die Empfindlichkeit der Steuerung gegenüber äußeren Einflüssen betrachten.
In der Standardliteratur wird meist der in Abbildung 1.1 dargestellte Aufbau verwendet. Das Pendel ist hierbei mit dem Gelenk auf einem Wagen befestigt, der entlang einer Schiene verschoben werden kann. Zur Steuerung, also der Balancierung des Pendels, wird eine Kraft, genauer ein Kraftstoß nach links oder rechts, auf den Wagen ausgeübt.

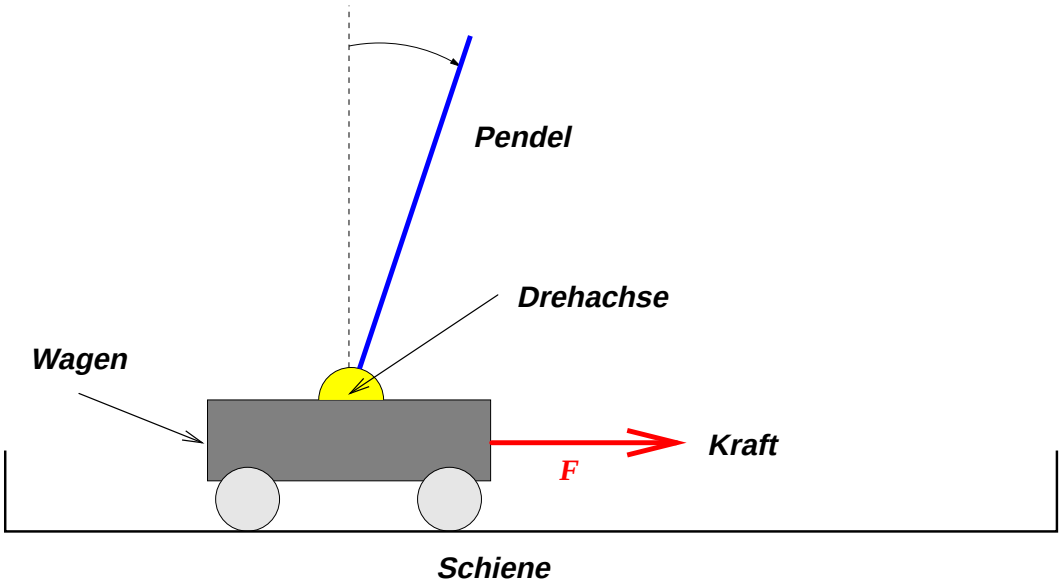


Abbildung 1: Standardaufbau mit Wagen

Der hier verwendete Aufbau unterscheidet sich etwas vom oben Beschriebenen. Statt auf einem Wagen ist der Pendelstab drehbar auf einem weiteren Stab montiert, der beweglich auf der Grundplatte befestigt ist. Diese Anordnung wird auch als doppeltinverses Pendel bezeichnet. Als Steuerungsmöglichkeit dient ein Motor, der in der Drehachse des unteren Stabes, im folgenden Antriebsarm genannt, angeflanscht ist. Der Aufbau ist schematisch in Abbildung 1.1 dargestellt.

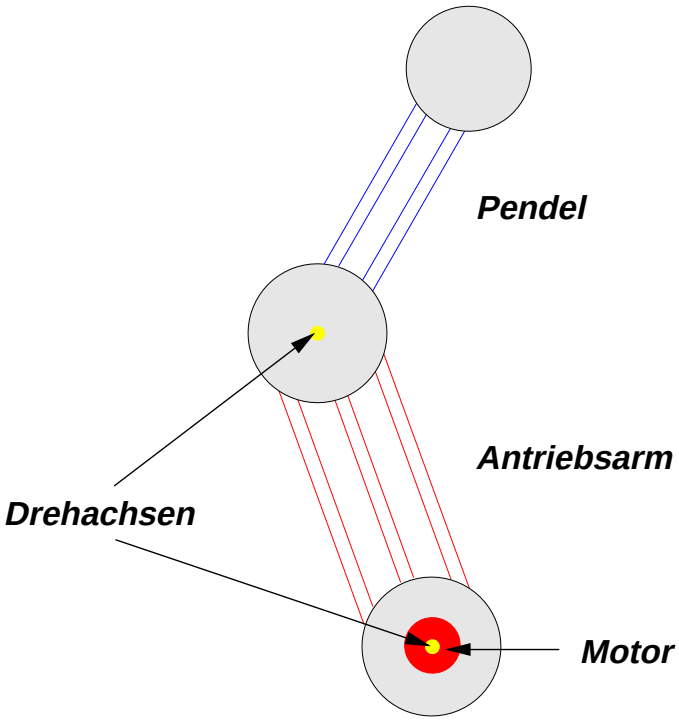


Abbildung 2: Pendelsystem - schematisch

1.2. Ziele

Als Hauptziele der beiden Studienarbeiten sind folgende Punkte formuliert. Zuerst soll ein Simulationsmodell, mit dem das Pendelsystem unabhänging von der Hardware untersucht werden kann, auf dem Rechner erstellt werden. Wenn die Hardware vorhanden ist, muß eine Schnittstelle zu dieser programmiert werden. Auf diese Grundlagen aufbauend soll ein neuronaler Regler zur Regelung der Simulation oder des Pendelsystems implementiert werden. Dieser Regler muß dabei echtzeitfähig ausgeführt werden, damit die Pendel-Hardware in Echtzeit gesteuert werden kann. Dies erfordert auch Modifikationen am verwendeten Betriebssystem und stellt wohl die größte Herausforderung in dieser Studienarbeit dar. Abschließend ist noch ein Vergleich der verschiedenen neuronalen Regler mit dem Fuzzy-Controller und Bewertung der Eignung der unterschiedlichen neuronalen Netze, d.h. eine Art „Benchmark“ für neuronale Netze, geplant.

1.3. Ausführung

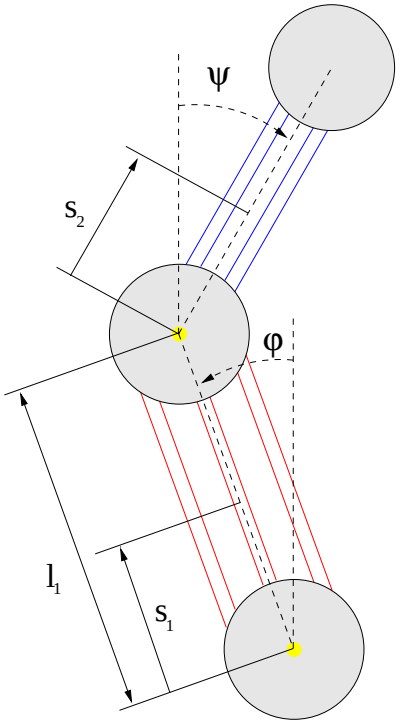
Durch die verwendete Pendel-Hardware wird als Zielsystem ein PC vorausgesetzt. Als Betriebssystem wird Linux eingesetzt, da nur Linux den in Kapitel 3.1 beschriebenen Anforderungen entspricht. Nach der Bestimmung des Zielsystems, müssen zum Erreichen obiger Ziele im Laufe der Arbeit verschiedene Teilprobleme gelöst werden. Zuerst ist eine physikalisch korrekte Beschreibung der Bewegungsgleichungen des Pendelsystems, siehe Kap. 2.1, zu formulieren. Auf deren Grundlage kann das Simulationsmodell durch Lösen des gewonnenen Differentialgleichungssystems mit numerischen Mitteln, siehe Kap. 2.2, erstellt werden. Um ein leicht bedienbares Testsystem für die Simulation und die Regelung zu erhalten, wird eine grafische Oberfläche unter dem X-Window System erstellt, siehe Kap. A. Für die Hardwaresteuerung wird ein Gerätetreiber als ladbares Kernelmodul unter Linux benötigt, siehe Kap. B, C und D. Durch geeignete Modifikationen am Linux-Kernel, siehe Kap. 3.6 und B.5, Verwendung einer Zweiprozeßsteuerung mit Interprozeßkommunikation zur Steuerung, siehe Kap. 5.1, und Erweiterung des Gerätetreibers wird die Echtzeitfähigkeit des Linux Betriebssystems sichergestellt. Die Regelung des Pendels wird durch Adaption des vorhandenen Fuzzy-Controllers für die Echtzeitsteuerung ermöglicht, Kap. 3 und Laufer (1996). Die Regelung durch ein neuronales Netz wird in Laufer (1996) untersucht.

2. Simulation des Pendelsystems

Um ein vorhandenes Hardwaresystem genauer zu untersuchen und zu verstehen, wird ein Modell des Systems erstellt. Anhand dieses Systemmodells können dann verschiedene Steuerungsmöglichkeiten verglichen werden.

2.1. Simulationsmodell

Die Simulation des Pendels beruht auf einem Modell, in dem verschiedene Energieformen einfließen, wobei jedoch die Reibung vernachlässigt wird. Die Energiegleichungen werden mit der modifizierten Lagrangeschen Gleichung zweiter Art gelöst und zu einem gekoppelten Differentialsystem zweiter Ordnung geführt. Hierbei wird das bewegte Koordinatensystem des Pendels in das stationäre Koordinatensystem des Antriebsarms transformiert. Grundlage der folgenden Berechnung ist das in Abbildung 3 gezeigte System, wobei die in der Abbildung angegebenen Parameter l_i, s_i und Θ_i bekannte Konstanten sind.



Pendelwinkel	ψ
Schwerpunktabstand	s_2
Massenträgheitsmoment des Pendels	Θ_P

Antriebsarmwinkel	φ
Armlänge	l_1
Schwerpunktabstand	s_1
Massenträgheitsmoment des Arms	Θ_A

Abbildung 3: Aufbau und Beschreibung des Pendelsystems

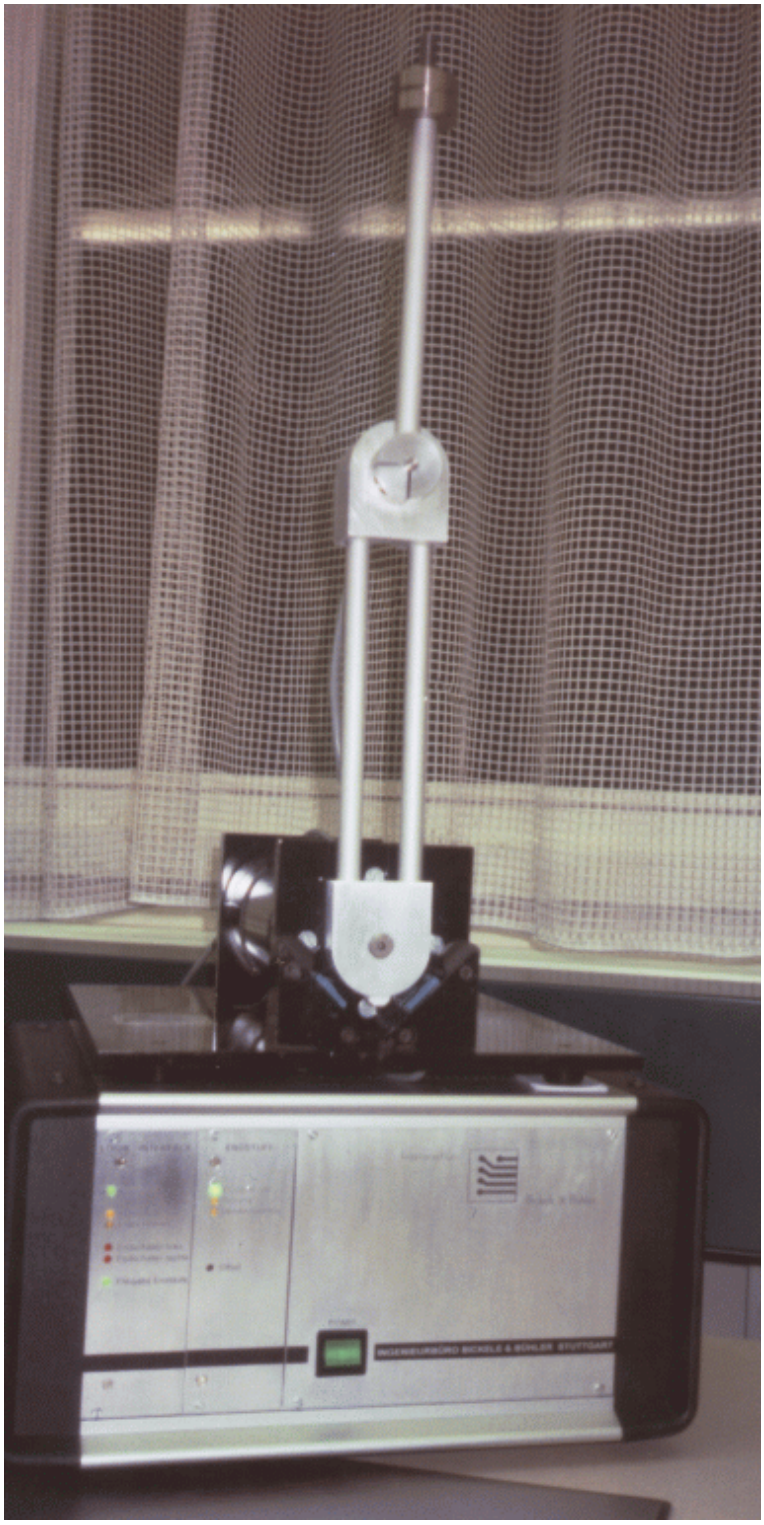


Abbildung 4: Hardware des Pendelsystems

Physikalisch gesehen handelt es sich um ein gekoppeltes Doppelpendelsystem, welches nach den Gesetzen der Massenträgheit unter Einfluß der Erdanziehung schwingt. Noch zu berücksichtigen ist dabei die Kraftwirkung durch den steuernden Motor. Im Pendelsystem werden zunächst der Pendel- und der Antriebsarm getrennt betrachtet. Die gesamte Masse eines Armes wird hierbei punktförmig konzentriert im Schwerpunkt des Armes angenommen. Daraus ergibt sich ein bestimmtes Massenträgheitsmoment, welches zusammen mit dem Schwerpunkt zur Beschreibung eines einzeln schwingenden Armes ausreicht. Um das komplette System zu modellieren, muß der Einfluß der beiden schwingenden Arme aufeinander, und zusätzlich die Kraftwirkung durch den Motor berücksichtigt werden. Dieses komplexe System kann nicht mehr durch eine Schwingungsgleichung beschrieben werden. Es ergibt sich vielmehr ein gekoppeltes Differentialgleichungssystem, welches nur numerisch lösbar ist.

Für das beschriebene Pendelsystem wird zunächst die in ihm enthaltene potentielle Energie berechnet. Sie ergibt sich zu:

$$U = (m_1 g s_1 + m_2 g l_1)\cos(\varphi) + m_2 g s_2 \cos(\psi)$$

Analog ergibt sich der Ansatz für die kinetische Energie. Hierbei bezeichnet die Summe $(\Theta_A + \Theta_M)$ das Gesamtträgheitsmoment von Arm und Motor.

$$E = \frac{1}{2}((\Theta_A + \Theta_M) + m_2 l_1) + \frac{1}{2}(\Theta_P + m_2 {s_2}^2)\dot{\psi}^2 - m_2 l_1 s_2 \dot{\psi}\dot{\varphi}\cos(\varphi + \psi)$$

Aus den so bestimmten Energieinhalten kann die *modifizierte innere Energie* L des Systems berechnet werden:

$$L = E - U$$

Zur Vereinfachung werden die folgenden Abkürzungen eingeführt:

$$\begin{aligned} K_a &= \Theta_A + \Theta_M + m_2 \cdot {l_1}^2, \\ K_b &= \Theta_P + m_2 \cdot {s_2}^2, \\ K_c &= m_2 \cdot l_1 \cdot s_2, \end{aligned} \qquad \begin{aligned} -K_d &= m_1 \cdot g \cdot s_1 + m_2 \cdot g \cdot l_1, \\ -K_e &= m_2 \cdot g \cdot s_2, \end{aligned}$$

Die Bewegungsgleichungen ergeben sich aus den Lagrange-Gleichungen:

$$\begin{aligned} \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\varphi}}\right) - \frac{\partial L}{\partial \varphi} &= M(t), \\ \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\psi}}\right) - \frac{\partial L}{\partial \psi} &= 0, \end{aligned}$$

zu

$$\begin{aligned} K_a \cdot \ddot{\varphi} &= K_c \cdot \ddot{\psi} \cdot \cos(\varphi+\psi) - K_c \cdot \dot{\psi}^2 \cdot \sin(\varphi+\psi) \\ &\quad - K_d \cdot \sin(\varphi) + M(t) \qquad (1) \\ K_b \cdot \ddot{\psi} &= K_c \cdot \ddot{\varphi} \cdot \cos(\varphi+\psi) - K_c \cdot \dot{\varphi}^2 \cdot \sin(\varphi+\psi) \\ &\quad - K_e \cdot \sin(\psi) \qquad (2) \end{aligned}$$

Szabo (1964) behandelt ein gekoppeltes Pendelsystem, welches dem hier vorliegenden Doppelpendel entspricht. Durch Substitution der Ansatzgrößen[1] können die Differentialgleichungen ineinander übergeführt werden. Eine Literaturrecherche ergab obendrein, daß das Problem des invertierten Doppelpendels auch der Problemstellung Glocke, Klöppel entspricht, welche in Müller und Magnus (1987) der Literaturliste behandelt wird und zu einem identischen Ergebnis führt. Die Gleichungen können auch hier durch Substitution der Winkel ineinander übergeführt werden.
Durch Umformen und Ineinandereinsetzen der beiden Gleichungen (1) und (2) können sie in die für Bewegungsgleichungen übliche Form gebracht werden. Es gilt dann für die beiden Winkelbeschleunigungen:

$$\begin{aligned} \ddot{\varphi} = \frac{1}{\frac{{K_c}^2}{K_b}\cos(\varphi+\psi)^2 - K_a} \quad & \left(\frac{{K_c}^2}{K_b} \sin(\varphi+\psi)\cos(\varphi+\psi)\,\dot{\varphi}^2 + K_c \sin(\varphi+\psi)\,\dot{\psi}^2 \right. \\ & \left. + \frac{K_c K_e}{K_b} \sin(\psi)\cos(\varphi+\psi) + K_d \sin(\varphi) - M(t) \right) \\ \ddot{\psi} = \frac{1}{\frac{{K_c}^2}{K_a}\cos(\varphi+\psi)^2 - K_b} \quad & \left(\frac{{K_c}^2}{K_a} \sin(\varphi+\psi)\cos(\varphi+\psi)\,\dot{\psi}^2 + K_c \sin(\varphi+\psi)\,\dot{\varphi}^2 \right. \\ & + \frac{K_c K_d}{K_a} \sin(\varphi)\cos(\varphi+\psi) + K_e \sin(\psi) \\ & \left. - \frac{K_c}{K_a} \cos(\varphi+\psi)\, M(t) \right) \end{aligned}$$

Um die Gleichungen übersichtlicher darzustellen, werden die konstanten Terme herausgezogen. Dies führt zu folgender Form, wobei neue Abkürzungen eingeführt werden. Die unten beschriebenen Abkürzungen bestehen nur aus Konstanten, die sich

[1] In der Literatur sind die Winkel anders angesetzt.

vor der Simulation aus den technischen Daten des Systems berechnen lassen.

$$\ddot{\varphi} = \frac{1}{A\cos(\varphi+\psi)^2 - N} \quad \Big(A\,\sin(\varphi+\psi)\cos(\varphi+\psi)\dot{\varphi}^2 + C\,\sin(\varphi+\psi)\,\dot{\psi}^2 + B\,\sin(\psi)\cos(\varphi+\psi) + D\,\sin(\varphi) - M(t)\Big) \tag{3}$$

$$\ddot{\psi} = \frac{1}{E\cos(\varphi+\psi)^2 - L} \quad \Big(E\,\cos(\varphi+\psi)\sin(\varphi+\psi)\,\dot{\psi}^2 + C\,\sin(\varphi+\psi)\,\dot{\varphi}^2 + F\,\sin(\psi)\cos(\varphi+\psi) + H\,\sin(\psi) + G\,\cos(\varphi+\psi)\,M(t)\Big) \tag{4}$$

mit den neu eingeführten Abkürzungen:

$$A = \frac{{K_c}^2}{K_b} = \frac{{m_2}^2\,{l_1}^2\,{s_2}^2}{\Theta_P + m_2\cdot {s_2}^2} \qquad F = \frac{K_c\,K_d}{K_a} = \frac{-m_2\,l_1\,s_2\,g\,(m_1\,s_1 + m_2\,l_1)}{\Theta_A + \Theta_M + m_2\cdot {l_1}^2}$$

$$B = \frac{K_c\,K_e}{K_b} = \frac{-{m_2}^2\,l_1\,{s_2}^2\,g}{\Theta_P + m_2\cdot {s_2}^2} \qquad G = \frac{-K_c}{K_a} = \frac{-m_2\,l_1\,s_2}{\Theta_A + \Theta_M + m_2\cdot {l_1}^2}$$

$$C = K_c = m_2\cdot l_1\cdot s_2 \qquad H = K_e = -m_2\cdot g\cdot s_2$$

$$D = K_d = -m_1\cdot g\cdot s_1 - m_2\cdot g\cdot l_1 \qquad L = K_b = \Theta_P + m_2\cdot {s_2}^2$$

$$E = \frac{{K_c}^2}{K_a} = \frac{{m_2}^2\,{l_1}^2\,{s_2}^2}{\Theta_A + \Theta_M + m_2\cdot {l_1}^2} \qquad N = K_a = \Theta_A + \Theta_M + m_2\cdot {l_1}^2$$

Zum Vergleich sind hier die äquivalenten Formeln angegeben, wie sie von Mathematica errechnet werden. Hierbei kann man leicht erkennen, daß sich die beiden Gleichungen hauptsächlich um einen Term unterscheiden, der den Einfluß der Motorwirkung beschreibt.

$$\begin{aligned}\ddot{\varphi} &= \frac{K_e\,\sin(\psi) + K_c\,\dot{\varphi}^2\,\sin(\varphi+\psi)}{K_c\,\cos(\varphi+\psi)} \\ &- \frac{K_b}{K_c\,\cos(\varphi+\psi)\,\left(K_a K_b - {K_c}^2\cos(\varphi+\psi)^2\right)} \\ &\cdot\Big[K_a\,\Big(K_c\,\sin(\varphi+\psi)\,\dot{\varphi}^2 + K_e\,\sin(\psi)\Big) \\ &\quad + K_c\,\cos(\varphi+\psi)\,\Big(K_c\,\sin(\varphi+\psi)\,\dot{\psi}^2 + K_d\,\sin(\varphi) - M(t)\Big)\Big]\end{aligned} \tag{5}$$

$$\begin{aligned}\ddot{\psi} &= \frac{-1}{K_a\,K_b - {K_c}^2\cos(\varphi+\psi)^2} \\ &\cdot\Big[K_a\,\Big(K_c\,\sin(\varphi+\psi)\,\dot{\varphi}^2 + K_e\,\sin(\psi)\Big) \\ &\quad + K_c\,\cos(\varphi+\psi)\,\Big(K_c\,\sin(\varphi+\psi)\,\dot{\psi}^2 + K_d\,\sin(\varphi) - M(t)\Big)\Big]\end{aligned} \tag{6}$$

2.2. Numerische Lösung mit Hilfe des Runge-Kutta Verfahrens

Das gekoppelte Differentialsystem zweiter Ordnung des Simulationsmodells, beschrieben durch das Gleichungssystem (3, 4) oder äquivalent (5, 6), kann numerisch mit dem *Runge-Kutta Verfahren* gelöst werden. Das Verfahren von Runge-Kutta zum Lösen von Differentialgleichungen wird in der Praxis wegen seiner hohen Genauigkeit der Approximation geschätzt. Der Nachteil des Verfahrens liegt in der Berechnung von vier Funktionswerten, was den Rechenaufwand erhöht. Die Genauigkeit wird durch den Abgleich der Taylor-Glieder bis einschließlich vierter Ordnung erreicht.
Bei der oben beschriebenen Simulation tritt ein Differentialgleichungssystem zweiter Ordnung auf, wobei die Anfangsbedingungen für $t = t_0$ lauten:

$$\begin{aligned} \varphi_t &= \varphi_0 & \qquad \psi_t &= \psi_0 \\ \dot{\varphi}_t &= \dot{\varphi}_0 & \qquad \dot{\psi}_t &= \dot{\psi}_0 \end{aligned}$$

Die Lösung des Differentialgleichungssystems durch das Runge-Kutta Verfahren unter diesen Anfangsbedingungen kann wie folgt berechnet werden, wobei die Schrittweite des Verfahrens h beträgt, vgl. Zurmühl 1965:

Erster Schritt, $t = t_0$

$$\begin{aligned} \varphi_1 &= \varphi_0 & \qquad \psi_1 &= \psi_0 \\ u_1 &= \dot{\varphi}_1 \cdot h & \qquad v_1 &= \dot{\psi}_1 \cdot h \\ k_1 &= \ddot{\varphi}_1 \cdot \frac{h^2}{2} & \qquad l_1 &= \ddot{\psi}_1 \cdot \frac{h^2}{2} \end{aligned}$$

Zweiter Schritt, $t = t_0 + \frac{h}{2}$

$$\begin{aligned} \varphi_2 &= \varphi_0 + \frac{1}{2}u_0 + \frac{1}{4}k_1 & \qquad \psi_2 &= \psi_0 + \frac{1}{2}v_0 + \frac{1}{4}l_1 \\ u_2 &= u_0 + k_1 & \qquad v_2 &= v_0 + l_1 \\ k_2 &= \ddot{\varphi}_2 \cdot \frac{h^2}{2} & \qquad l_2 &= \ddot{\psi}_2 \cdot \frac{h^2}{2} \end{aligned}$$

Dritter Schritt, $t = t_0 + \frac{h}{2}$

$$\begin{aligned} \varphi_3 &= \varphi_0 + \frac{1}{2}u_0 + \frac{1}{4}k_1 & \qquad \psi_3 &= \psi_0 + \frac{1}{2}v_0 + \frac{1}{4}l_1 \\ u_3 &= u_0 + k_2 & \qquad v_3 &= v_0 + l_2 \\ k_3 &= \ddot{\varphi}_3 \cdot \frac{h^2}{2} & \qquad l_3 &= \ddot{\psi}_3 \cdot \frac{h^2}{2} \end{aligned}$$

Vierter Schritt, $t = t_0 + h$

$$\begin{aligned} \varphi_4 &= \varphi_0 + u_0 + k_3 \\ u_4 &= u_0 + 2 \cdot k_3 \\ k_4 &= \ddot{\varphi}_4 \cdot \frac{h^2}{2} \end{aligned} \qquad \begin{aligned} \psi_4 &= \psi_0 + v_0 + l_3 \\ v_4 &= v_0 + 2 \cdot l_3 \\ l_4 &= \ddot{\psi}_4 \cdot \frac{h^2}{2} \end{aligned}$$

Letzter Schritt – Mittelwertbildung

$$\begin{aligned} k &= \frac{1}{3}(k_1 + k_2 + k_3) \\ 2k' &= \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \qquad \begin{aligned} l &= \frac{1}{3}(l_1 + l_2 + l_3) \\ 2l' &= \frac{1}{3}(l_1 + 2l_2 + 2l_3 + l_4) \end{aligned}$$

$$\begin{aligned} \varphi_n &= \varphi_0 + u_0 + k \\ u_n &= u_0 + 2k' \end{aligned} \qquad \begin{aligned} \psi_n &= \psi_0 + v_0 + l \\ v_n &= v_0 + 2l' \end{aligned}$$

Nun erhält man die neuen Werte für $\varphi,\ \dot{\varphi},\ \psi,$ und $\dot{\psi}$ nach der Schrittweite h durch:

$$\begin{aligned} \varphi_{t+h} &= \varphi_n \\ \dot{\varphi}_{t+h} &= \frac{u_n}{h} \end{aligned} \qquad \begin{aligned} \psi_{t+h} &= \psi_n \\ \dot{\psi}_{t+h} &= \frac{v_n}{h} \end{aligned}$$

2.3. Bestimmung der Schrittweite h des Runge-Kutta Verfahrens

Für das Runge-Kutta Verfahren muß die Schrittweite h der einzelnen Rechenschritte bestimmt werden. Dazu gibt es verschiedene Möglichkeiten. Das einfachste Verfahren ist es, durch Ausprobieren verschiedener Werte von h zu bestimmen, welche Genauigkeit für das zu lösende Problem hinreichend ist. In der Literatur zur Steuerung des inversen Pendels, vergleiche Saerens und Soquet (1991), wird der Wert $h = 0.001$ angegeben. Dieser Wert ist nach den Probeläufen der Simulation geeignet. Das Problem ist dabei, daß ein zu großer Wert die Systemgenauigkeit der Simulation senkt, und ein zu kleiner Wert durch die unvermeidlichen Rundungsfehler bei der Rechnung die Genauigkeit ebenfalls vermindert. Ein weiteres Problem ist natürlich, daß sich der Rechenaufwand mit der Vergrößerung von h erhöht. Bei der Erprobung des Systems wurden mit verschiedenen Werten von h die unten tabellierten Ergebnisse erzielt.
Die Ausgabe gibt die Werte des Winkels, der Winkelgeschwindigkeit und der Winkelbeschleunigung jeweils für den Pendelarm („`pen:`“) und Antriebsarm („`mot:`“) an:

$h = 0.001$ `pen: 236.9 -2.69 -1.79 mot: 152.6 -6.06 +3.52 M:00.00`
nach 10.000 Iterationen.

$h = 0.0001$ `pen: 236.9 -2.69 -1.79 mot: 152.6 -6.06 +3.52 M:00.00`
nach 100.000 Iterationen.

$h = 0.001$ `pen: 003.3 +0.29 +0.01 mot: 321.3 +1.79 -5.41 M:00.00`
nach 26.000 Iterationen.

$h = 0.0001$ `pen: 003.3 +0.29 +0.01 mot: 321.3 +1.79 -5.41 M:00.00`
nach 260.000 Iterationen.

Diese Berechnungen erfolgten ohne Reibung, ohne Steuerung des Motors und bei einem Startwert des Pendelwinkels von 1 Grad. Die restlichen Parameter waren zu Beginn auf 0.0 gesetzt. Wie man erkennt, reicht die Berechnung mit $h = 0.001$ durchaus aus, da die Rechnung mit zehnfacher Genauigkeit keinerlei Veränderung der Ergebnisse bewirkt.

2.4. Arbeitsweise der Simulation

Die Simulation des Pendelsystems erfolgt nun nach folgendem Schema: Ausgehend von den Anfangswerten, die vom Anwender eingestellt werden können, siehe dazu Kap. A.2.2, werden bei jedem Zeitschritt die neuen Werte der Parameter des Simulationsmodells berechnet. Der Zeitschritt entspricht dem Wert h des Runge-Kutta Verfahrens. Er kann vom Anwender beim Erstellen des Programms geändert werden, siehe Kap. E.1. Bei jedem Schritt werden die Paramter Winkel, Winkelgeschwindigkeit und Winkelbeschleunigung für den Pendel- und den Antriebsarm neu berechnet. Diese sind durch das Differentialgleichungssystem aus den Ausgangsparametern des vorigen Zeitschritts eindeutig bestimmt. Die Ausgangsparameter sind dabei: Der Winkel, die Winkelgeschwindigkeit und die Winkelbeschleunigung des Pendel- und Antriebsarms, sowie die Kraftwirkung des Motors auf den Antriebsarm. Mit dem Runge-Kutta Verfahren ergeben sich dann, wie vorher beschrieben, jeweils die neuen Werte des nächsten Zeitschritts.

3. Regelung des Pendelsystems

Um eine Regelung des Pendelsystems zu ermöglichen, muß zuerst der augenblickliche Zustand des Systems bestimmt werden. Bei der Simulation des Pendels nach dem in Kap. 2 vorgestellten Algorithmus sind in jedem Zeitschritt die Parameter Winkel, Winkelgeschwindigkeit und Winkelbeschleunigung für das Pendel und den Antriebsarm verfügbar.
Von der Hardware können in jedem Zeitschritt nur die Winkel vom Pendel- und Antriebsarm eingelesen werden, siehe hierzu Kap. B und D. Durch den Zeitschritt kann mit Hilfe der Winkel aus den vorherigen Schritten die Winkelgeschwindigkeit berechnet werden. Dies ist nötig, weil der Controller zur Regelung auch die Winkelgeschwindigkeiten von Pendel und Antriebsarm benötigt.
Aufbauend auf diesen Parametern, die zu jedem Zeitschritt neu gemessen bzw. berechnet werden müssen, kann eine Regelung implementiert werden. Da das System nichtlinear ist, können nur dementsprechende Regler, wie z.B. ein Fuzzy-Controller oder ein neuronales Netz, verwendet werden. Bei der Steuerung des Pendelsystems, also der Hardwaresteuerung, stellt die Lösung der Echtzeitfähigkeit der Steuerung ein großes Problem dar. Falls die Echtzeitfähigkeit nicht immer sichergestellt werden kann, also zu große Zeitspannen ohne Steuerung auftreten, wird das Pendel umfallen, was den Fehlschlag der Steuerung bedeutet. Daher ist das wichtigste Problem die Sicherstellung der Echtzeitfähigkeit zu jedem Zeitpunkt im System. Die Lösung dieses Problems legt das Betriebssystem fest, wie im folgenden genauer ausgeführt wird, und erfordert eine aufwendigere Implementierung des Controllers.

3.1. Wahl eines geeigneten Systems für die Steuerung

Bevor die Hardwaresteuerung implementiert werden konnte, mußte zuerst ein geeignetes System gefunden werden, welches die nicht geringen Anforderungen der Steuerung an die Echtzeitfähigkeit des Systems erfüllt. Das genaue Einhalten des Zeittaktes, was für das Lernen und Steuern mit dem neuronalen Netz essentiell ist, führt in einem nicht echtzeitfähigen Betriebssystem zu großen Problemen. Da zur Steuerung kein spezielles Echtzeitsystem zur Verfügung stand, mußte die Steuerung mit einem Standardsystem durchgeführt werden. Windows ist für Echtzeitanwendungen prinzipiell nicht geeignet, da bei Windows 3.11 kein preemptatives Multitasking existiert, und bei Windows 95 oder NT der Prozeßwechseltakt nicht beeinflußt werden kann. Unter DOS ist es zwar möglich, wenn nur das Steuerprogramm läuft und die Hardware direkt angesteuert wird. Dazu muß das Steuerprogramm aber auch unter den eingeschränkten Möglichkeiten von DOS funktionieren, was u.a. bedeutet: kein Multitasking, nur 640 KB Speicher und 16-Bit Speichermodell.
Für unsere Steueraufgabe mit einem neuronalen Netz reicht diese Leistungsunfähigkeit nicht aus. Daher wurde die Steuerung unter dem UNIX^{TM} kompatiblen System

Linux implementiert. Dies hat auch noch den großen Vorteil, daß das komplette System im Quellcode verfügbar ist und an entsprechende Anforderungen angepaßt werden kann. Außerdem ergibt sich durch Verwendung eines UNIX^{TM} Systems der große Vorteil, daß die Anwendung leicht portierbar bleibt. Wie in Kap. A.1 dargelegt, läuft die Simulation auf den unterschiedlichsten Rechnern. Die Hardwaresteuerung funktioniert aber nur auf PCs, in denen die originale Schnittstellenkarte eingebaut ist. Da der Gerätetreiber für die Hardware aber vollkommen gekapselt ist und nur über eine wohldefinierte Schnittstelle angesprochen wird, sollte eine Portierung der Hardwaresteuerung auf andere Hardware möglich sein.
Um nun die oben geforderte Echtzeitfähigkeit auf einem Linux System zu erreichen, sind die im folgenden beschriebenen Anpassungen erforderlich. Dazu wird erst etwas über Prozeßwechselstrategien, *Scheduling* genannt, ausgeholt. Genauere Information über Scheduling finden sich z.B. in Hieronymous 1993 oder anderen Büchern über die UNIX-Architektur.

3.2. Echtzeitsteuerung unter Multitasking-Systemen

In Multitaskingsystemen werden durch den sogenannten *Scheduler* in einem bestimmten Takt Δs die lauffähigen Prozesse auf dem Prozessor gewechselt. Ein Prozeßwechsel kann dabei nur zu den Schedulezeitpunkten stattfinden. Innerhalb eines Zeitabschnitts Δs werden keine Prozeßwechsel durchgeführt.
Die einzige Ausnahme von dieser Regel ist, wenn ein Prozeß freiwillig Rechenzeit abgibt, sich also schlafen legt. Dies wird immer dann durchgeführt wenn der Prozeß nicht mehr weiterarbeiten kann, weil z.B. benötigte Daten noch nicht vorliegen. Aufgeweckt wird ein schlafender Prozeß dann durch das Eintreffen eines beim Schlafenlegen angegebenen Ereignisses. Ereignisse bedeuten normalerweise das Eintreffen von Daten, z.B. von der Platte, dem Netz, externen Geräten oder auch durch Benutzeraktivität, die der Prozeß zum Weiterarbeiten benötigt. Diese Ereignisse lösen einen Interrupt aus, der den Scheduler zum Wecken des Prozesses veranlaßt. Zu den lauffähigen Prozessen zählen alle Prozesse, die im Moment rechnen könnten, also nicht auf bestimmte Ereignisse warten.
Der Scheduler teilt die Rechenzeit möglichst gleichmäßig auf die einzelnen Prozesse auf, d.h. wenn fortwährend zwei Prozesse lauffähig sind, kommt jeder ungefähr jeden zweiten Scheduletakt zum Rechnen[2]. Auf Multiuser-/Multitaskingsystemen werden normalerweise zwischendurch auch andere Prozesse lauffähig, meist durch das Eintreffen oben beschriebener Ereignisse, wodurch sie dann aufgeweckt werden. Daher kann im allgemeinen nicht vorausgesagt werden, wann ein bestimmter Prozeß Rechenzeit bekommt und wie lange er rechnen kann. Dies führt bei Echtzeitanwendungen zu großen Problemen, da nicht sichergestellt werden kann, daß ein Echtzeitprozeß wirklich dann seine Rechenzeit bekommt, wenn er sie benötigt. Durch die später beschriebenen Modifikationen des normalen Schedulers muß diese

[2]Dies ist so nur richtig, wenn die beiden Prozesse die gleiche Priorität besitzen.

Echtzeitfähigkeit erreicht werden. Zuerst wird aber die prinzipielle Funktionsweise der Hardware Steuerung dargelegt.

3.3. Ansteuerung des Pendels

Für die Steuerung des Pendels müssen dem Motor fortwährend neue Befehle gegeben werden. Da die Berechnung der Befehle auch einige Zeit in Anspruch nimmt, verwendet man normalerweise eine getaktete Ansteuerung. Bei der Steuerung über das neuronale Netz und vor allem beim Lernen des Netzes ist eine sehr genaue Einhaltung dieses Steuertaktes nötig.
Dazu ist auf der Schnittstellenkarte ein Timer eingebaut, der in einem bestimmten Takt, einstellbar sind 100 Hz bis 10 kHz, Interrupts auslösen kann. Diese Interrupts bewirken im Gerätetreiber das Aufrufen der Interruptfunktion `pendel_interrupt()`, welche bei jedem Takt einen Steuerbefehl an den Motor abliefert und die aktuellen Werte des Pendelsystems von der Hardware einliest. Hiermit ist sichergestellt, daß die Steuerung genau getaktet abläuft.
Der Gerätetreiber muß außerdem die Parameter der Hardware umrechnen, da das Programm ein anderes Format verwendet. Die Hardware liefert Winkel im Bereich von -2000 bis 2000, wobei der Winkel des Pendels relativ zum Antriebsarm gemessen wird. Das Programm bekommt die Winkel über das Device im Bereich von $-\pi$ bis π bezüglich der Vertikalen. Die vom Steuerprozeß benötigte Winkelgeschwindigkeit wird auch von der Interruptfunktion berechnet:

$$v = \frac{(\text{Winkel} - \text{Winkel vor } \texttt{VCOUNTER} \text{ Takten}) * \text{Frequenz des Interrupts}}{\texttt{VCOUNTER}}$$

Die Variable `VCOUNTER` wird in der Headerdatei `pendulum.h` definiert.
Die Steuerimpulse, die vom Interrupt an die Hardware gegeben werden, müssen vorher vom Programm berechnet werden und rechtzeitig zum Interrupt bereitstehen. Falls nicht rechtzeitig ein neuer Steuerbefehl an den Gerätetreiber übergeben wurde, wird nichts gesteuert. Falls mehrere Interrupts hintereinander ohne Steuerung auftreten, bewegt sich das Pendel in dieser Zeit ohne Steuerung, *Totzeit* genannt, als freischwingendes Pendelsystem. Wenn die *Totzeit* einen von der Stellung des Antriebsarms und der Kraft des Motors abhängigen Grenzwert übersteigt, wird das Pendel instabil und fällt um. Aus diesem Grund synchronisiert der Gerätetreiber das Programm, was so funktioniert:
Bei einem Lesezugriff auf das Device wird das Programm mit der `sleep()` Funktion schlafen gelegt, bis neue Werte der Parameter durch die Interruptfunktion gelesen wurden. Diese weckt mit `wakeup()` das Programm wieder auf. Das Programm berechnet nun aus den gerade gelesen Werten den neuen Steuerimpuls und übergibt ihn an den Gerätetreiber. Danach versucht es wieder neue Werte zu lesen. Hier wird es nun wieder schlafen gelegt usw.. Die Interruptfunktion schickt den vom Programm übergebenen Steuerimpuls an die Hardware, bevor es das Programm

wieder weckt. Während das Programm schläft, können im System andere Prozesse laufen. Das Zusammenspiel der einzelnen Teile ist in Abbildung 5 dargestellt. Im oberen Bereich ist das Pendelsystem mit seinen Hardwareparametern dargestellt. Der von der Schnittstellenkarte erzeugte Takt bewirkt jeweils einen Interruptaufruf im Gerätetreiber, welcher die Parameter des Pendelsystems für das Programm umrechnet, die vom Programm berechneten Steuerparameter an den Motor übergibt und den Programmzugriff über das Device synchronisiert. Das ganze Verfahren kann nur dann in der beschriebenen Weise ablaufen, wenn das Programm bis zum nächsten Interrupt mit der Berechnung fertig ist. Andernfalls würde es nicht mehr mit den Steuerimpulsen nachkommen, und es würden regelmäßig Interrupts ohne Steuerung auftreten.

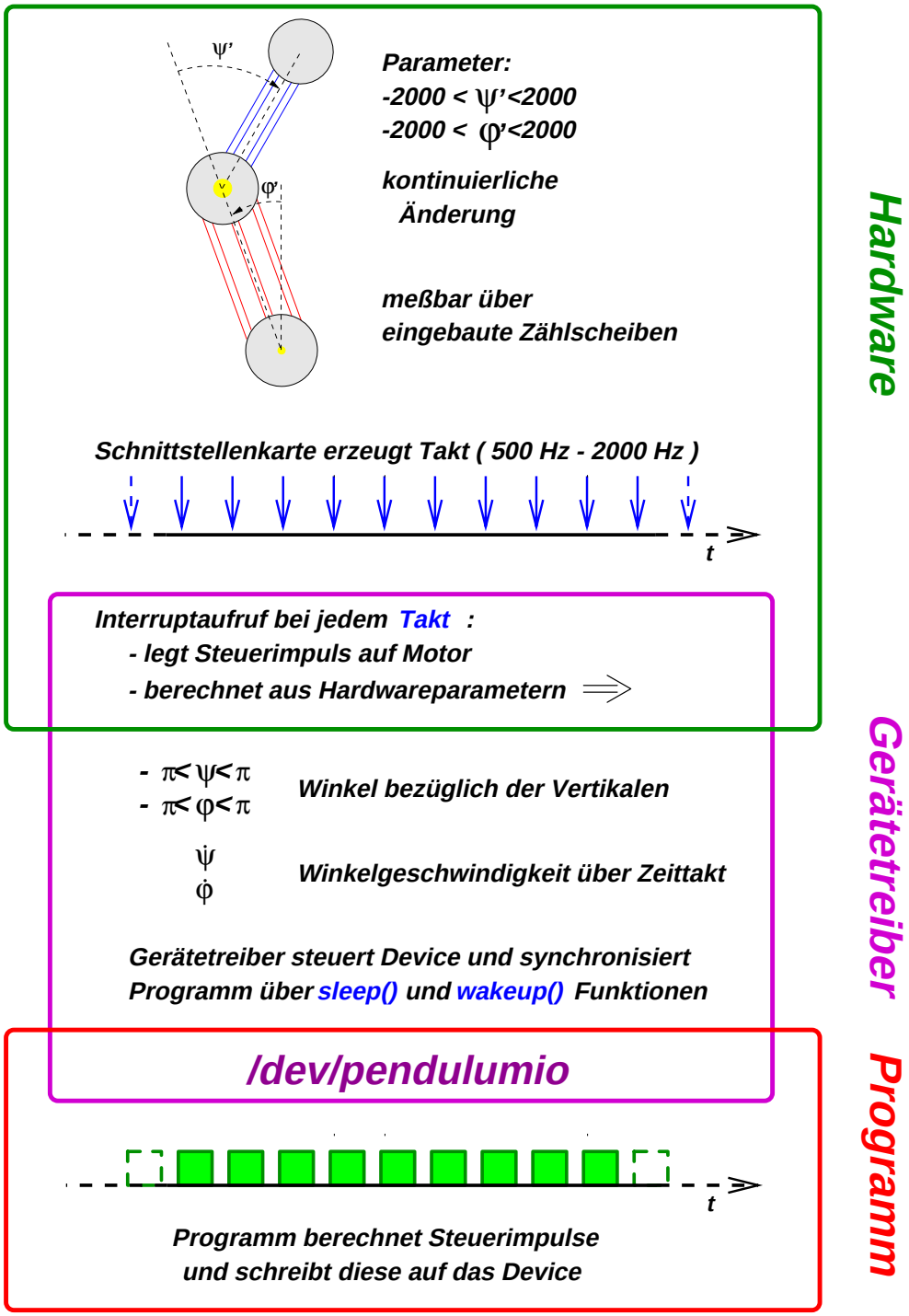


Abbildung 5: Zusammenspiel der einzelnen Teile bei der Steuerung

3.4. Steuerung durch einen normalen Prozeß

In den Abbildungen 6 bis 8 wird das Scheduling bei der Steuerung des Pendels dargestellt. Von der Schnittstellenkarte kommen die Interruptanforderungen, wie in Kap. 3.3 beschrieben, in einem bestimmten Takt. Das Steuerprogramm muß zwischen den Takten jeweils die neuen Steuerbefehle berechnen und an die Hardware übergeben. Dies muß rechtzeitig vor dem Eintreffen des nächsten Taktimpulses erfolgen. Nach der Übergabe des berechneten Steuerbefehls an den Gerätetreiber hat das Programm grundsätzlich zwei Möglichkeiten:
a) Es kann in einer Endlosschleife warten, bis wieder neue Daten von der Hardware über den Gerätetreiber gelesen werden können. Dies hat den Vorteil, daß der Steuerprozeß bis zum nächsten Prozeßwechsel sicher weiterrechnen und dabei bei jedem Takt einen Steuerimpuls berechnen kann. Der Nachteil davon ist, daß sinnlos Rechenzeit verschwendet wird, da in der Zeit, in der das Programm auf neue Daten wartet, kein anderer Prozeß zum Zuge kommen kann. Daraus wird ein gravierendes Problem, wenn noch andere Prozesse lauffähig sind, was bei einem Multitasking/Multiusersystem den Normalfall darstellt. Da der Scheduler die Rechenzeit auf die verschiedenen Prozesse möglichst gleichmäßig verteilt, kommt beim nächsten Taskwechsel ein anderer Prozeß an die Reihe. In der Rechenzeit von anderen Prozessen können natürlich vom Steuerprozeß keine Steuerimpulse berechnet und an die Hardware geschickt werden.
b) Das Programm kann sich schlafen legen und wird erst wieder aufgeweckt, wenn neue Daten von der Hardware über den Gerätetreiber ankommen. Dazu weckt die Interruptfunktion das Programm auf. Dies hat den Vorteil, daß das Programm nur Rechenzeit verwendet, wenn es wirklich etwas zu rechnen hat. Es wird also keine Rechenzeit in Warteschleifen verschwendet. Nun gibt es aber auch hier einen gravierenden Nachteil. Wenn ein Prozeß wieder aufgeweckt wird, bekommt er nicht sofort Rechenzeit zugeteilt, sondern frühestens zum nächsten Prozeßwechselzeitpunkt. Nur wenn gerade kein anderer Prozeß läuft bekommt er sofort Rechenzeit. Falls aber beim Schlafenlegen des Steuerprozesses ein anderer Prozeß lauffähig war, bekommt dieser sofort die Rechenzeit und der Steuerprozeß kann erst wieder beim nächsten Takt des Schedulers Rechenzeit bekommen.
Wie lange nun der Steuerprozeß nicht rechnen kann, das Pendel also ohne Steuerimpulse bleibt, hängt vor allem vom Scheduletakt Δs ab. Falls zum Prozeßwechselzeitpunkt mehrere Prozesse lauffähig sind, hängt es auch noch von den Auswahlkriterien des Schedulers ab, welchen Prozeß er als nächstes aktiviert.

3.4.1. Normaler Scheduler

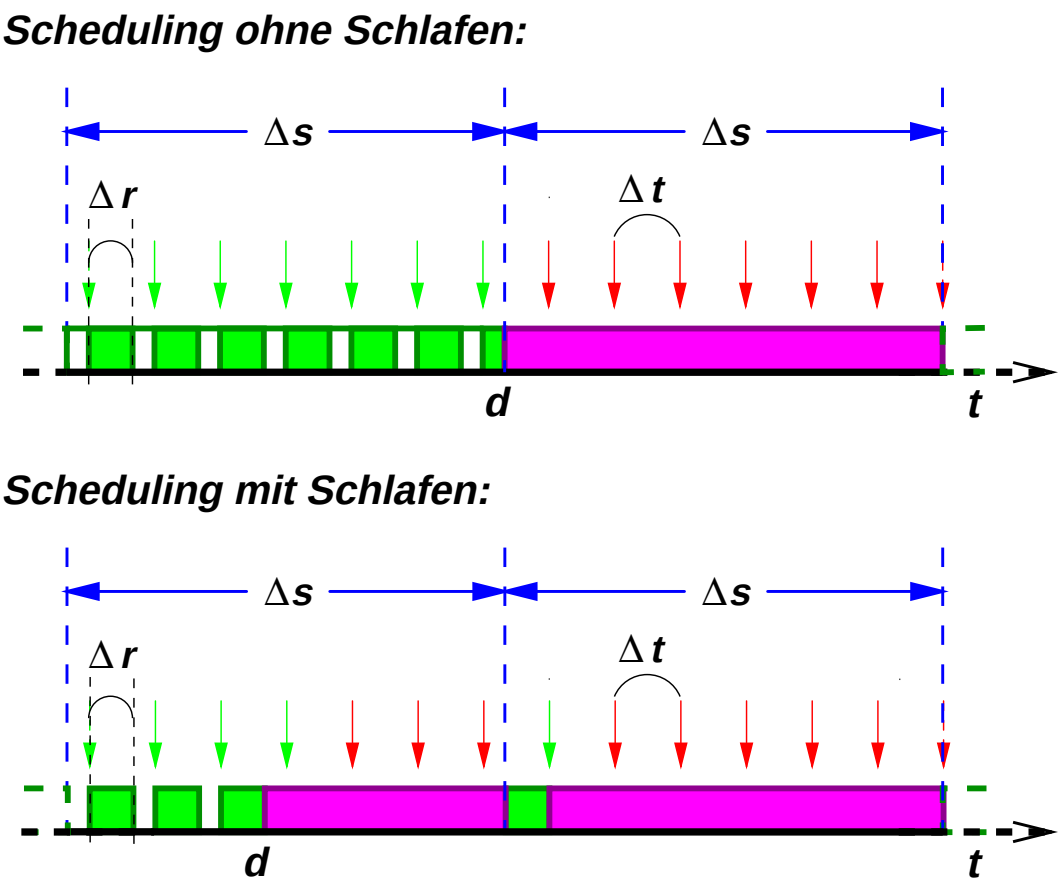


Abbildung 6: Normaler Scheduler, $\Delta s \gg \Delta t$

In Abbildung 6 wird das Verhalten bei einem normalen Scheduler mit einem Takt von meistens 100 Hz dargestellt, d.h. $\Delta s = 0.01s$. Der Takt der Steuerung Δt ist hier also wesentlich kleiner, etwa um den Faktor 10, als der Scheduletakt. Man sieht deutlich, daß der Steuerprozeß alle Steuerimpulse bedienen kann, also nur

Taktimpulse mit Steuerbefehl auftreten, solange kein anderer Prozeß lauffähig ist. Wenn aber ein anderer Prozeß an die Reihe kommt, ab Zeitpunkt d in der Zeitachse, ist mindestens für die Zeitspanne Δs keine Steuerung möglich. Wegen der Schedulerstrategie, allen Prozessen möglichst gleich viel Rechenzeit zu geben, kann es aber vorkommen, daß andere Prozesse $x \cdot \Delta t$ Takte zum Rechnen bekommen, wenn mehrere Prozesse lauffähig sind oder eine höhere Priorität haben. Dies ist besonders dann ein Problem, wenn der Steuerprozeß nicht schläft, da er so unnötig Rechenzeit verschwendet, die eigentlich andere Prozesse nutzen könnten. Falls sich der Steuerprozeß schlafenlegt, kann in der Zeit vom Ende der Befehlsberechnung bis zum nächsten Prozeßwechsel schon ein anderer Prozeß an die Reihe kommen. Dies bedeutet aber auch, daß dann nur alle Δs eine Steuerung erfolgen kann. Zusammengefaßt kann die Totzeit $x \cdot \Delta s$ lang sein, d.h. diese Zeitspanne kann zwischen zwei Steuerimpulsen vergehen. Das Pendel bewegt sich in dieser Zeit vollkommen ungeregelt. In der Praxis wurden Ausfälle der Steuerung bis zu $800ms$ erreicht. Ein normaler Scheduler ist also für die Echtzeitregelung nicht brauchbar.

3.4.2. Echtzeit-Scheduler

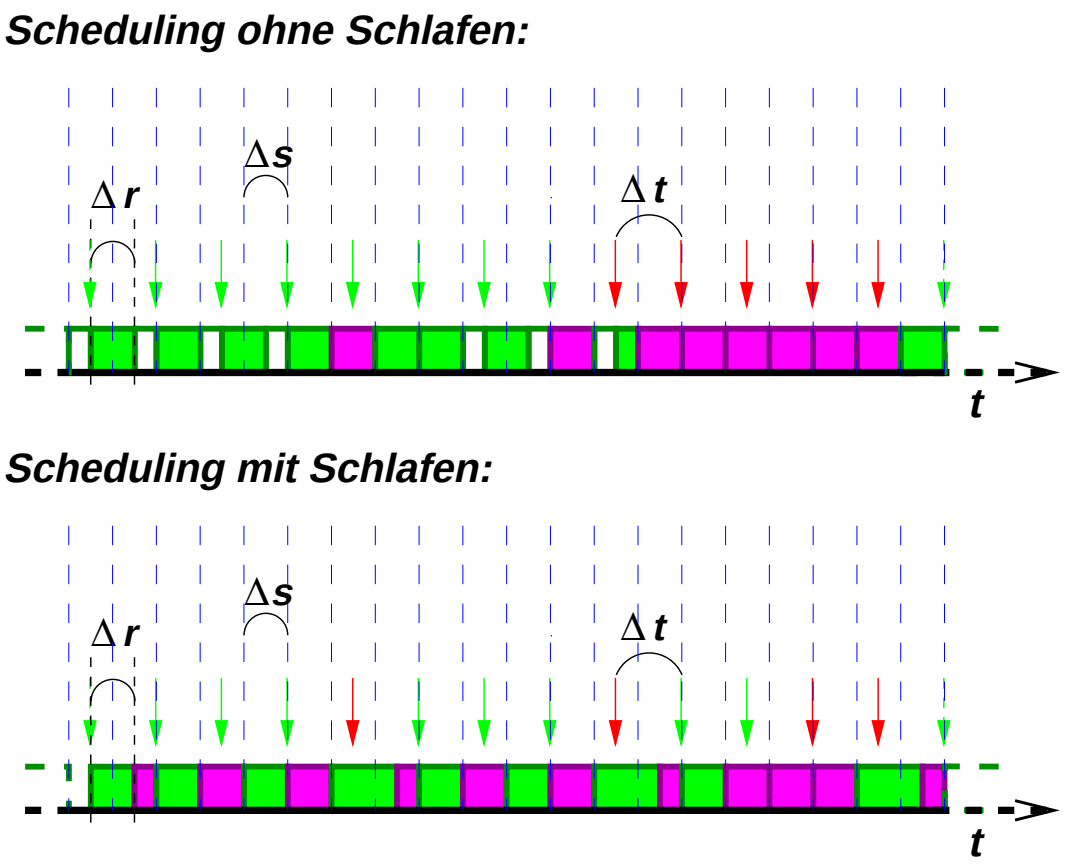


Abbildung 7: Echtzeit-Scheduler, $\Delta s \leq \Delta t$

Wenn man den Scheduletakt Δs im Bereich des Taktes der Steuerung oder kleiner wählt, bekommt man einen Echtzeit-Scheduler. Man wählt dann z.B. einen Scheduletakt von 1500 Hz oder 2000 Hz. Nach Möglichkeit sollte vermieden werden, daß der Hardwaretakt ein Vielfaches des Scheduletaktes ist, da sonst Interferenzerscheinungen die Steuerung erheblich stören können, z.B. wenn sich die Interrupts der Hardware dauernd mit dem Scheduler überschneiden.
Wenn beim Echtzeit-Scheduler ein anderer Prozeß zum Laufen kommt, ist die Zeitspanne bis zum nächsten Prozeßwechsel wesentlich kürzer, da Δs viel kleiner ist, z.B. $10ms$ beim normalen Scheduler und $0,5ms$ beim Echtzeitscheduler. Dies bewirkt, daß die Zeit ohne Steuerimpulse im Mittel geringer ist.
In Abbildung 7 wird der Vorteil des Schlafens deutlich. Obwohl die maximale Zeit ohne Steuerimpulse in der zweiten Zeitreihe erheblich kürzer ist, bekommen die anderen Prozesse dort sogar mehr Rechenzeit (9 Zeiteinheiten im Vergleich zu 8 Einheiten in der ersten Zeitreihe). Wenn der Steuerprozeß schläft verteilt sich in diesem Fall die Rechenzeit der anderen Prozesse besser, da im Falle des Nichtschlafens zur Rechenzeit des Steuerprozesses auch die Zeit gezählt wird, in der er eigentlich nichts rechnet, sondern nur Rechenzeit beim Warten auf neue Daten verbraucht.
Insgesamt erreicht man schon eine wesentliche Verbesserung der Steuerung. Die maximale Totzeit ist, wie oben beschrieben, immer noch $x \cdot \Delta s$. Da der Scheduletakt Δs wesentlich kleiner ist, wird die maximale Zeit ohne Steuerimpulse, die für die Pendelsteuerung äußerst relevant ist, reduziert. Falls aber mehrere Prozesse zur Laufzeit anstehen, besteht trotzdem noch ein Problem von zu langen Totzeiten ohne Steuerimpulsen, denn der Wert von x kann bei mehreren lauffähigen Prozessen immer noch zu hoch sein.

3.4.3. Echtzeit-Scheduler mit Prioritätsanpassung

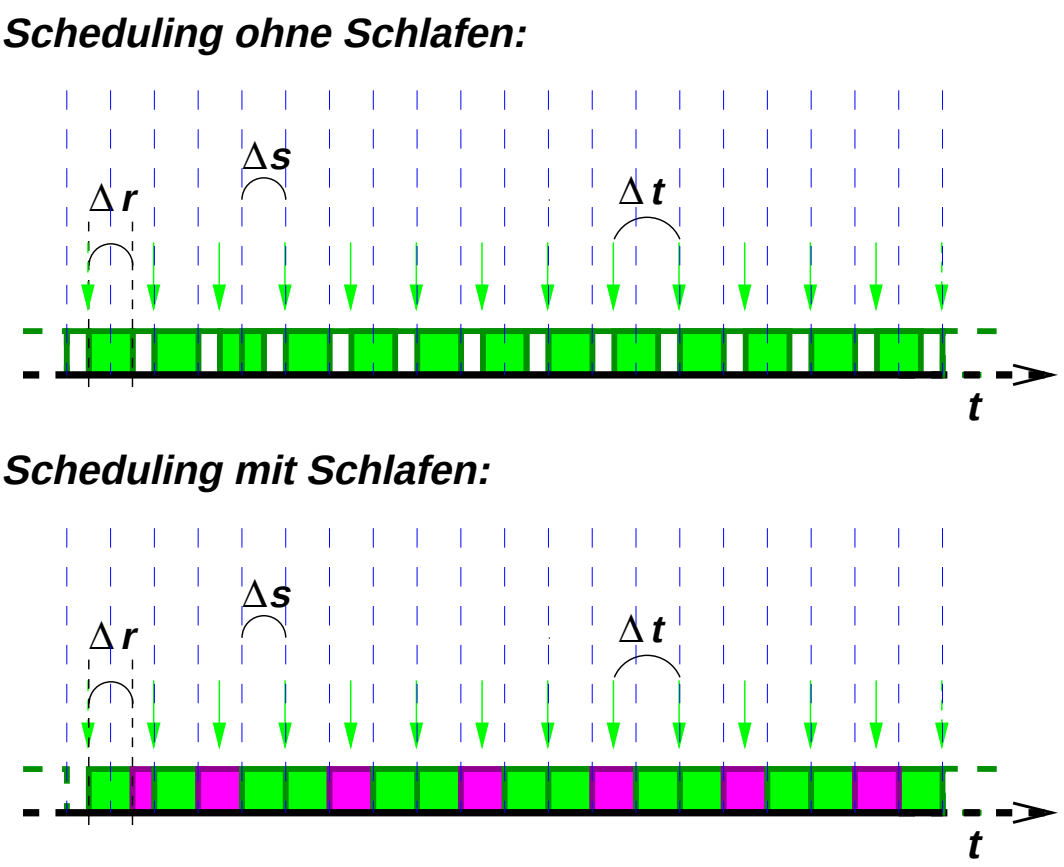


Abbildung 8: Echtzeit-Scheduler mit Prioritätsanpassung, $\Delta s \leq \Delta t$

Der Grund für zu lange Totzeiten bei mehreren lauffähigen Prozessen ist, daß andere Prozesse bevorzugt werden können, wenn sie gleichzeitig mit dem Steuerprozeß lauffähig sind. Um dies zu verhindern, wird in der Interruptfunktion die Priorität des Steuerprozesses geändert.
Der Prozeß wird zum einen als Echtzeitprozeß eingestuft. Zum anderen wird bei jedem Interrupt von der Hardware die aktuelle Laufpriorität auf den maximalen Wert gesetzt. Dies bewirkt, daß beim nächsten Prozeßwechsel im Scheduler auf jeden Fall der Steuerungsprozeß ausgeführt wird. Die Funktionsweise wird in Kap. B.5 in der Interruptfunktion erläutert. Auf einem System darf aber immer nur ein Echtzeitprozeß laufen. Andernfalls geht die Echtzeitfähigkeit durch die konkurrierenden Echtzeitprozesse verloren.
Der Vorteil dieser Prioritätsanpassung ist nun, daß der Steuerprozeß wirklich nach jedem Prozeßwechsel als nächstes rechnen kann. Damit sinkt die maximale Zeit ohne Steuerung von $x \cdot \Delta s$ auf Δs, da x nur den Wert 1 annehmen kann. Mit einem Echtzeitscheduler, wobei Δs kleiner als Δt ist, geht normalerweise kein Takt verloren.

Leider trifft dies nur solange zu, wie keine intensiven I/O-Zugriffe stattfinden. Denn bei Zugriffen auf bestimmten Geräten, z.B. IDE-Platten, wird für längere Zeit durch *busy-waiting* der Prozeßwechsel unterbrochen bzw. die Interrupts abgeschaltet, siehe dazu auch Kap. 3.6. Bei intensiven Zugriffen auf solche Geräte kann sich die Totzeit auf einige Millisekunden erhöhen.
Ein Problem besteht jetzt noch, wenn der Rechner zu langsam ist, d.h. die Zeit zum Berechnen des Steuerimpulses höher ist, als die Taktrate Δt. In diesem Fall wird der Steuerprozeß fortwährend ausgeführt. Dies bedeutet, daß dann kein anderer Prozeß laufen kann und das ganze System scheinbar steht. Um dies zu verhindern, wird durch die Interruptfunktion der Steuerprozeß abgebrochen, falls dieser Fall eintreten sollte.

3.5. Leistungsfähigkeit der Echtzeitsteuerung

Mit den in Abbildung 8 beschriebenem Echtzeitscheduler mit Prioritätsanpassung wird eine für ein Multiuser-/Multitaskingsystem sehr gute Echtzeitfähigkeit erreicht. Für eine gute Steuerung des Pendels sind Taktraten der Hardware von etwa 1000 Hz bis 2000 Hz nötig. Wenn man dazu noch den Scheduler mit einem Takt von 1500 Hz bis 2000 Hz verwendet, gehen im normalen Betrieb keine Steuerimpulse verloren, d.h. die Totzeit ist maximal ein Scheduletakt.
Durch IO-Zugriffe, z.B. auf Platten, kann die maximale Totzeit aber auf einige Millisekunden erhöht werden. Auf einem Rechner, auf dem außer dem Steuerprozeß keine besonders plattenintensiven Prozesse laufen, bleibt die Totzeit bei einem Scheduletakt. Dies entspricht nahezu der Antwortzeit von speziellen Echtzeitsystemen und ist für die Regelungsaufgabe des Pendels völlig ausreichend. Hierbei ist noch zu betonen, daß während der Pendelsteuerung trotzdem auf dem System ganz normal weitergearbeitet werden kann. Die Geschwindigkeit ist aber durch den erhöhten Scheduleraufwand und den Regelprozeß herabgesetzt.
Probleme mit der Steuerung treten auf, wenn wichtige Teile des Steuerprozesses wegen Speichermangels durch Paging oder Swapping auf die Platte ausgelagert werden müssen. Dann können Unterbrechungen im Sekundenbereich auftreten. Dieser Fall tritt aber nur bei totaler Speicherüberlastung des Rechners auf.

3.6. Anforderungen für die Echtzeitsteuerung

Die Hardwareansteuerung ist nur auf IBM-kompatiblen PCs unter Linux lauffähig und benötigt die originale Schnittstellenkarte mit dem Aufbau des Pendels, siehe dazu Kap. B und Bickele (1996). Damit die Hardware in Echtzeit gesteuert werden kann, müssen folgende drei Bedingungen erfüllt sein:

a) Die Reaktionszeit auf einen Hardwareinterrupt darf einige Mikrosekunden regelmäßig nicht überschreiten.

b) Die maximale Zeit ohne Steuerung der Hardware, die *Totzeit*, muß kleiner als ungefähr 10 Millisekunden sein. Die *Totzeit* ist das maximale Zeitintervall, in dem der Steuerprozeß vom Scheduler keine Rechenzeit bekommt. Entscheidend dafür ist vor allem die Zeit, die nach dem Aufwecken des Steuerprozesses durch einen Interrupt vergeht, bis der Steuerprozeß wieder Rechenzeit vom Scheduler bekommt.

c) Für eine stabile Steuerung müssen die Steuerbefehle in einem äußerst gleichmäßigen Zeittakt erfolgen. Daher darf die maximale Abweichung ausgeführter Steuerbefehle pro Sekunde einen Wert von etwa 1% nicht überschreiten.

Um diese Bedingungen zu erfüllen sind im einzelnen folgende Voraussetzungen erforderlich:

a) Für die Steuerung mit dem neuronalen Netz ist eine Rechenleistung von mindestens einem Pentium 75 oder höher nötig. Für die Steuerung mit dem Fuzzy-Controller reicht ein schneller 486 aus.

b) Es wird ein spezieller Echtzeitkernel benötigt, da der Standardkernel nur mit einem Takt von 100 Prozeßwechseln pro Sekunde arbeitet. Um diesen speziellen Kernel zu erzeugen, muß in der Datei `include/asm/param.h` im Linux-Kernel Quellbaum, z.B. `/usr/src/linux`, der Eintrag:

```
#ifndef HZ
#define HZ 100
#endif
```

durch

```
#define HZ 2000
```

ersetzt werden. Hierbei kann man statt 2000 Werte im Bereich von 1500 bis 2000 ausprobieren.

c) Der Gerätetreiber für die Hardware benötigt einen Linux-Kernel ab Version 1.3.70. Empfohlen wird die Verwendung des neuesten Linux-Kernels von Version

2.0. Es kann nur ein nach obiger Methode angepaßter Kernel verwendet werden! Um das Modul des Gerätetreibers zu laden, werden noch die passenden *Modutils* benötigt. Das Modutils-Paket ist nötig, um Linux Gerätetreiber in Modulform verwenden zu können, d.h. um den Gerätetreiber in den Kernel einzubinden. Das Modutils-Paket ist als modules-2.0.0.tar.gz auf den bekannten Linux-Servern zu finden.

d) Damit die Interrupts der Karte auch wirklich den Prozessor gleichmäßig erreichen, darf kein anderes Gerät die Interrupts für längere Zeit abschalten. Außerdem darf kein Gerätetreiber im Kernel längere Warteschleifen ohne Prozeßwechsel, sog. *busy-waiting* durchführen. Probleme bereitet daher vor allem ältere und schlecht entworfene Hardware, das sind z.B. IDE-Platten oder Floppy-Streamer.
Für die IDE-Platten gibt es aber meist eine Lösung, die verhindert, daß die Interrupts zu lange vom Datentransfer mit der Platte unterbrochen werden. Die Lösung besteht darin, daß man bei fast allen IDE-Platten mit dem Programm *hdparm* die Interrupts während des Datentransfers eingeschaltet lassen kann. Dazu wird folgender Befehl nach jedem Rechnerstart ausgeführt:

```
hdparm -u 1 /dev/hda /dev/hdb
```

WICHTIG: Vorher sind aber unbedingt die Warnhinweise in der Dokumentation von *hdparm* zu beachten!
Das Programm *hdparm* kann man als hdparm-2.9.tar.gz oder neuer auf bekannten Linux-Servern finden. Platten, die über Busmaster Schnittstellen angesprochen werden, wie fast alle SCSI-Systeme, oder moderne EIDE-Platten im DMA-Modus machen keine Probleme. In Linux 2.0 wird der DMA-Modus bei EIDE-Platten nur bei dem Intel Triton und seinen Nachfolgerchipsätzen unterstützt.

4. Regelung mit einem neuronalen Netz

Nachdem die Steuerung mit dem Fuzzy-Controller erfolgreich implementiert wurde, wie in den vorigen Kapiteln beschrieben, wird nun die Möglichkeiten einer neuronalen Regelung untersucht. Dazu bekommt das neuronale Netz als Eingabe die Parameter Winkel und Winkelgeschwindigkeit des Pendel- und Antriebsarms und bestimmt daraus eine Steuergröße für den Motor. Ziel ist es dann, wie beim Fuzzy-Controller, das Pendel zu balancieren. Die Eingabeparameter des Pendels werden hierbei an die vier Eingabeneuronen des neuronalen Netzes gelegt. Nach dem Propagieren des Netzes liefert das Ausgabeneuron den Wert für die Kraft, die der Motor auf den Antriebsarm ausüben soll. Die Kraft bewirkt eine Änderung in den Parametern des Pendelsystems. Bei der Simulation werden nun im nächsten Zeitschritt, unter Berücksichtigung der Kraft, die neuen Werte der Parameter berechnet. Im Hardwaremodell wird die Kraft an den Motor angelegt und danach im nächsten Zeitschritt die neuen Werte des Pendelsystems gemessen. Die Funktionsweise des neuronalen Netzes und die Lernregeln für die Steuerung werden ausführlichst in der parallelen Studienarbeit, Laufer (1996), beschrieben.

5. Programm `xpendulum`

Um die Simulation visuell darzustellen, wurde das Programm **xpendulum** mit grafischer Oberfläche implementiert. Die ganze Funktionalität der Simulation wie auch der Steuerung ist unter dieser Oberfläche bedienbar. Die Oberfläche wurde auch mit dem Ziel entworfen, daß auch Laien, die nicht in der Materie eingearbeitet sind, das Programm bedienen und damit die Regelung des Pendels testen können. Durch die grafische Oberfläche wird auch das komplizierte System mit zwei Prozessen, die, wie im nächsten Kapitel erläutert, zur Steuerung notwendig sind, vor dem Benutzer weitestgehend verborgen.

5.1. Grundlegender Aufbau

Das Programm **xpendulum** besteht aus zwei getrennten Modulen. Nach dem Programmstart wird die grafische Oberfläche unter dem X-Window System angezeigt. Hier können die Parameter für die Simulation bzw. Regelung eingestellt und die gesamte Simulation bzw. Regelung gesteuert werden. Die Anzeige der Parameter, die grafische Darstellung des Pendelsystems und das Benutzerinterface stellen einen Prozeß dar.
Wenn nun eine Simulation bzw. die Regelung der Hardware gestartet wird, wird mit der Systemfunktion *fork()* ein zweiter Prozeß erzeugt, der die ganze Simulation bzw. Regelung übernimmt. Im Prinzip ist der zweite Prozeß nur ein sogenannter *Thread (Task)* des Hauptprogramms. Da die Threadfunktionen unter Linux noch nicht geeignet implementiert waren, werden die beiden Threads in dem Programm als eigenständige Prozesse realisiert.

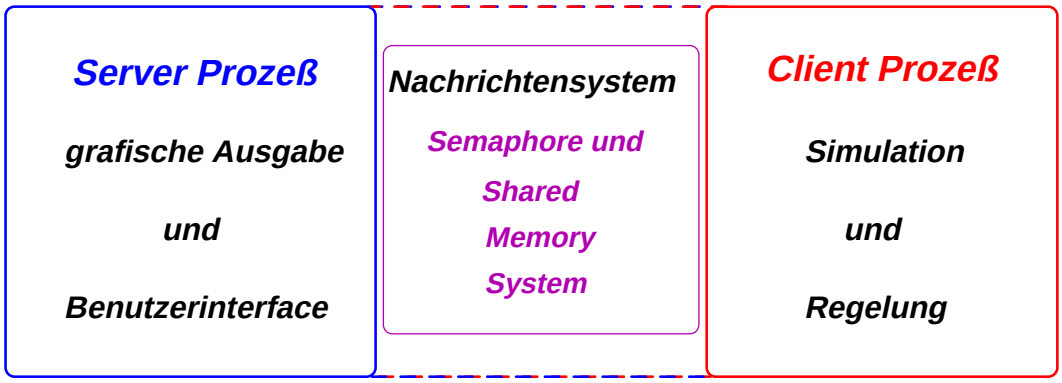


Abbildung 9: Server- und Clientprozeß

Das System mit den zwei Prozessen ist in Abbildung 9 dargestellt. Der Serverprozeß, ist dabei nur für das Benutzerinterface und die grafische Ausgabe verantwortlich. Seine Rechenzeit hängt direkt von der Angabe beim Parameter *Refresh Rate*, siehe Kap. A.2.2, und der Geschwindigkeit des X-Servers ab.

Der Clientprozeß (zweiter Prozeß) berechnet fortwährend die Steuerbefehle für die Regelung und die Simulationswerte bei der Simulation. Die Angabe bei *Calculation* gibt nur an, in welcher Zeitspanne Neuberechnungen zu erfolgen haben. Unabhängig davon ist die Rechenzeit des Clientprozesses immer maximal. Wohingegen der Server wirklich nur Rechenzeit bei der grafischen Ausgabe benötigt.
Die Verbindung der beiden Prozesse geschieht durch einen gemeinsam genutzten Speicherbereich. Dazu wird das im UNIX System-V definierte Shared Memory System eingesetzt. Auf der Grundlage diese Speicherbereichs ist ein Nachrichtensystem implementiert, welches den beiden Prozessen ermöglicht, Nachrichten, d.h. Befehle mit Parametern, auszutauschen. Durch getrennte Speicherbereiche für jede Richtung wird die Kommunikation in beide Richtungen gleichzeitig ermöglicht (vom Client zum Server und umgekehrt). Die Synchronisation des Zugriffs von Client und Server auf den gemeinsamen Speicherbereich erfolgt durch das Konzept der *Semaphore*. Dies bedeutet, daß jeweils nur ein Prozeß auf den Speicherbereich zugreifen kann. Dadurch wird die Datenkonsistenz sichergestellt, d.h. der empfangende Prozeß darf nur Daten lesen, die vollständig sind, und der sendende Prozeß nur Daten schreiben, wenn nicht gerade der lesende Prozeß Daten ausliest.
Der Client schickt zum Server hauptsächlich die neuen Parameter, die er berechnet bzw. gemessen hat, wohingegen der Server zum Client Befehle schickt, die neue Parameter setzen, die z.B. durch den Benutzer verändert wurden.

5.2. Schnittstelle zur Hardware

In Abbildung 10 wird grafisch die Funktionsweise des Hardwarezugriffs aufgezeigt. In einem UNIX^{TM} System können die Programme nicht direkt auf die Hardware zugreifen, sondern müssen sich spezieller Gerätetreibern bedienen, die in den Kernel eingebunden sind und die Hardware ansprechen können. Die Gerätetreiber stellen spezielle Dateien, *Devices* genannt (z.B. `/dev/pendulumio`), zur Verfügung, welche von den Programmen zum Zugriff auf die Hardware verwendet werden. Der Zusammenhang zwischen dem Programm, dem Gerätetreiber und der Hardware wird in der Grafik illustriert.

5.3. Anforderungen für das Programm

Das Programm **xpendulum** läuft nur unter *UNIX*^{TM} kompatiblen Betriebssystemen, z.B. unter *Linux*. Vorausgesetzt werden eine *X-Window* Implementation (Version X11R5 oder höher), der *GNU* C-Compiler (gcc, ab Version 2.7.0) und ein Shared Memory System nach *System-V-Standard*. Verwendet wurde die Simulation hauptsächlich auf SUN Sparc Workstation und dem Linux PC. Grundsätzlich sollte das Programm auf allen Rechnern laufen, die obige Voraussetzungen erfüllen. Die genauen Anforderungen für die Simulation sind in Kapitel A.1 dargelegt.

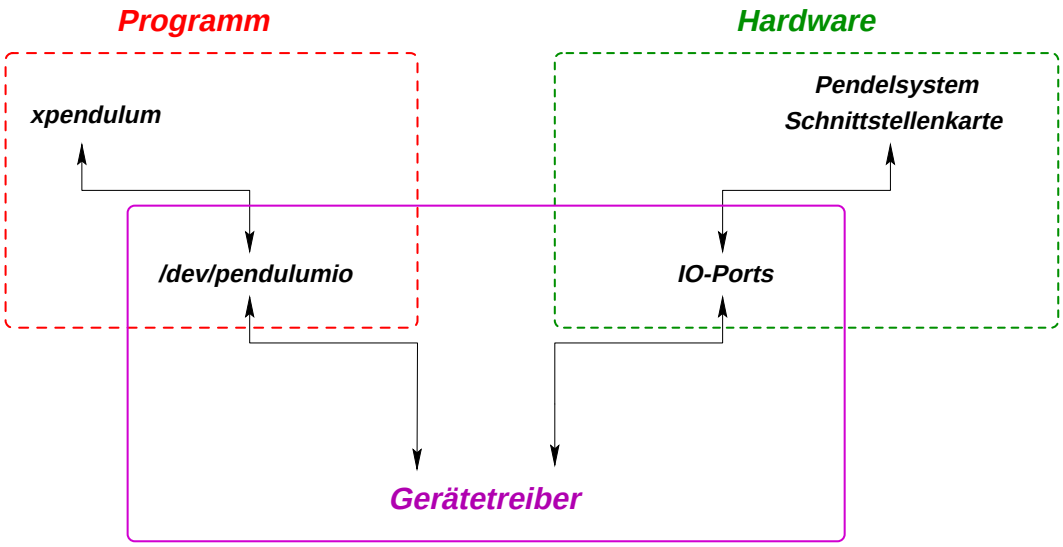


Abbildung 10: Hardwareschnittstelle

Anforderungen für die Echtzeitsteuerung

Die Hardwareansteuerung ist nur auf IBM-kompatiblen PCs unter Linux lauffähig und benötigt die originale Schnittstellenkarte mit dem Aufbau des Pendels, siehe dazu Kap. B und Bickele (1996). Damit die Hardware in Echtzeit gesteuert werden kann, müssen einige Bedingungen erfüllt sein. Die Reaktionszeit auf Hardwareinterrupts muß im Mikrosekundenbereich liegen. Wichtig ist aber vor allem, daß die maximale Zeit ohne Steuerung der Hardware unter etwa 10 Millisekunden bleibt. Für die neuronale Regelung ist auch ein möglichst gleichmäßiger Steuertakt erforderlich. Die genauen Anforderungen und die Voraussetzungen zum Erfüllen dieser Bedinungen sind im Kapitel 3.6 beschrieben.

5.4. Grafische Oberfläche

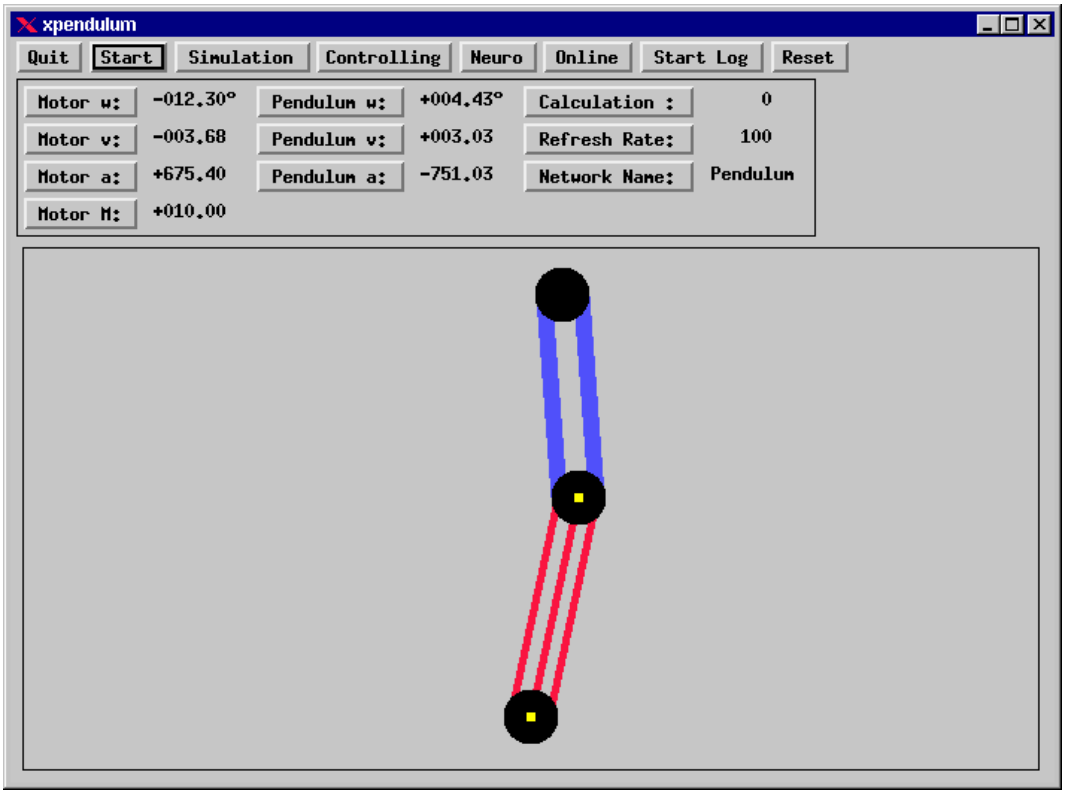


Abbildung 11: Hauptfenster von xpendulum

Das Programm wird mit **xpendulum** gestartet. Nach dem Aufruf erscheint eine grafische Oberfläche, auf der die augenblickliche Stellung des Pendels schematisch angezeigt wird. Mit Hilfe der Oberfläche können alle für die Simulation oder Steuerung relevanten Einstellungen vorgenommen werden.

6. Zusammenfassung

Während der Implementierung stellte sich heraus, daß bei eigentlich bekannten Verfahren, z.B. bei der neuronalen Regelung, unerwartete Probleme auftraten. Vor allem die geringe Ausführungsgeschwindigkeit der Neuro-Netzbibliothek, die sich aus der Universalität der Bibliothek zwangsläufig ergibt, führte zu großen Problemen. Der Rechenaufwand überforderte bei der Echtzeitsteuerung selbst schnelle PCs.
Andererseits ergaben sie in anderen Bereichen, bei denen anfangs weder auf bekannte Verfahren zurückgegriffen werden konnte, noch überhaupt eine Lösungsidee vorhanden war, schnell erfolgversprechende Konzepte. Der Gerätetreiber für die Schnittstellenkarte ergab aufgrund der guten Dokumentation, sowohl der Karte selber als auch des Gerätetreiberkonzepts von Linux, keine unerwarteten Probleme. Die Echtzeitfähigkeit von Linux erreichte so gute Werte, wie sie vorher in keinster Weise von einem UNIX^{TM} kompatiblen Multiuser-/Multitaskingsystem zu erwarten war. Diese Echtzeitfähigkeit spricht auch für das gelungene Design von Linux.

A. Programm

A.1. Anforderungen des Programms

Das Programm **xpendulum** läuft nur unter *UNIX*^{TM} kompatiblen Betriebssystemen, z.B. unter *Linux*. Vorausgesetzt werden eine *X-Window* Implementation (Version X11R5 oder höher), der *GNU* C-Compiler (gcc, ab Version 2.7.0) und ein Shared Memory System nach *System-V-Standard*. Zusätzlich werden die *XToolkit* und *Athena Widget* Bibliotheken benötigt, wobei von dieser die 3D Version *Xaw3d* empfohlen wird[3].

Wenn das Programm läuft, wird etwa 8 MB freier Hauptspeicher verwendet. Um eine fließende grafische Ausgabe zu erzielen, ist ein schneller X Server nötig (Grafikkarte mit Beschleunigerfunktionen!). Auf folgenden Systemen wurde das **xpendulum** Programm erfolgreich getestet:

- **Pentium 100 MHz; 32, 48 MB Speicher;**
  4 MB SVGA Beschleunigergrafikkarte; Linux 2.0;
  4 GB SCSI Platte, 2 GB EIDE Platte im DMA-Modus
  Auf diesem *Referenzsystem* wurde das Programms entwickelt und getestet. Die Regelung des Pendels mit dem Fuzzy-Controller läuft mit grafischer Ausgabe sogar bei hoher Last des Systems stabil. Alle Angaben über die Echtzeitfähigkeit in dieser Arbeit beziehen sich, wenn nichts anderes angegeben, auf dieses System.

  Verwendet wurde ein selbst zusammengestelltes modernes Linux System, basierend auf Linux 2.0 (ELF) und dem X-Window System X11R6.1 mit den Compilern gcc 2.7.2*i*3 und 2.7.2*pentium*, sowie den Bibliotheken libc-5.4.2, libm-5.0.6, libXaw3d-1.1 und den X-Window Bibliotheken von XFree86 3.1.2E.

- **Pentium 75 MHz; 16 MB Speicher;**
  2 MB SVGA Beschleunigergrafikkarte; Linux 2.0;
  1 GB EIDE Platte im PIO-Modus
  Dieses System wird als Steuerungsrechner für das Pendel verwendet. Die Rechenleistung und der Hauptspeicher reichen für die Steuerung mit dem neuronalen Netz aus. Obwohl die Platte an diesem Rechner nicht im DMA-Modus betrieben wird, läuft die Regelung selbst unter Last stabil. Das verwendete Linux System entspricht dem des Referenzsystems.

[3]Diese ist als Quellcode im Contrib Teil der X11-Window Quellen enthalten.

- **486 33 MHz; 8 MB Speicher;**
  256 KB VGA Grafikkarte; Linux ab 1.3.70
  Dieser Rechner hat sehr wenig Speicher für ein Linux System. Trotzdem läuft die Regelung des Pendels mit dem Fuzzy-Controller stabil, wenn der Echtzeitkernel verwendet wird. Die grafische Ausgabe auf diesem Rechner ist aber nicht möglich, da die Grafikkarte wesentlich zu langsam ist.

  Die Regelung mit einem neuronalen Netz ist nicht möglich, da die Rechenleistung etwa um den Faktor 10 zu gering ist.

- **Sun Ultrasparc 1/140; 143 MHz; 1 Prozessor;**
  64 MB Speicher; Solaris 2.5
  Dieser Rechner ist der schnellste, der am Lehrstuhl vorhanden ist. Die Geschwindigkeit erreicht 230% der Geschwindigkeit des Referenzsystems.

A.2. Bedienung

Nach dem Aufruf des Programms mit **xpendulum** erscheint das Hauptfenster, welches in drei Bereiche aufgeteilt ist, siehe Abbildung 11. Im obersten, direkt unter der Leiste des Fenstermanagers, sind die Buttons der Menüleiste angeordnet. Darunter befinden sich die verschiedenen Buttons, um die Parameter des Pendels und des Antriebsarms einzustellen. Außerdem können hier die Geschwindigkeit der Steuerung bzw. Regelung und der grafischen Ausgabe festgelegt werden. Desweiteren wird das Netz, welches die Steuerung übernimmt, hier angegeben. Den größten Teil des Fensters nimmt die grafische Darstellung der augenblicklichen Pendelposition ein.

A.2.1. Beschreibung der Menüleiste

Quit

Hiermit wird das Programm beendet. Falls gerade eine Simulation oder die Hardwaresteuerung läuft, wird sie abgebrochen. Das Programm sollte nur mit diesem Button beendet werden, damit die von ihm belegten Ressourcen wieder für das System freigegeben werden.

Start/Stop

Durch Betätigen des Start Buttons wird eine neue Simulation oder die Hardwaresteuerung gestartet. Von der Stellung des Buttons Simulation/Hardware hängt es ab, welche von beiden gestartet wird. Beim Start der Simulation werden die im Moment eingestellten Werte der Parameter, siehe auch Kap. A.2.2, als Startwert genommen. Für die Steuerung bzw. Simulation wird ein eigener Prozeß gestartet. Es laufen dann zwei Prozesse gleichzeitig, wobei einer die Darstellung und das Benutzerinterface übernimmt und der andere die Simulation bzw. Regelung durchführt. Falls der Button auf Stop steht, wird die augenblickliche Simulation oder die Hardwaresteuerung abgeschaltet.

Simulation/Hardware

Dieser Button hat nur Auswirkung, wenn die Simulation bzw. Regelung angehalten ist, d.h. der Button Start sichtbar ist. Hiermit wird festgelegt, ob beim nächsten Aufruf von Start die Simulation oder die Hardwaresteuerung gestartet wird.

Genaueres zu der Simulation ist in Kap. 2 und zur Regelung in Kap. 3 zu finden.

No Control/Controlling Bei der Button Stellung Controlling ist die Steuerung des Pendels eingeschaltet, bei der Stellung No Control abgeschaltet.

Fuzzy/Neuro Wenn die Steuerung aktiviert ist, wird hiermit festgelegt, ob der eingebaute Fuzzy-Controller das Pendel steuert oder das neuronale Netz. Welches neuronale Netz zum Steuern geladen wird, kann man mit dem Button Network Name festlegen.

Online/Offline Mit diesem Button wird die grafische Darstellung an- bzw. ausgeschaltet. Falls die Steuerung bzw. Regelung gerade läuft, ist die Wirkung des Befehls sofort sichtbar. Andernfalls wirkt sie sich beim nächsten Start mit Start aus.

Start/Stop Log Hiermit wird das Protokollieren der Parameter des Pendelsystems an- bzw. ausgeschaltet. Die Wirkung des Befehls tritt sofort ein – auch bei laufender Simulation. In dem Logfile `logfile_param` werden fortwährend die aktuellen Werte des Pendels und Antriebsarms mitgeschrieben. Das Logfile wird dabei im aktuellen Verzeichnis beim Programmstart abgelegt.
Das Format des Logfiles ist:

pen:	Winkel, Winkelgeschwindigkeit, Winkelbeschleunigung des Pendels
mot:	Winkel, Winkelgeschwindigkeit, Winkelbeschleunigung des Motors
M:	Krafteinwirkung auf den Motor

Reset Setzt bei der Simulation alle Parameter des Pendels und des Antriebsarms wieder auf die Ausgangswerte zurück, d.h. alle Werte werden mit 0.0 initialisiert.

A.2.2. Beschreibung der Buttons der Parameter

Neben der Beschreibung der Parameter wird jeweils der augenblickliche Wert angezeigt. Um einen Parameter zu ändern, muß der Button mit dem Namen des zu ändernden Parameters angeklickt werden. In der sich daraufhin öffnenden Dialogbox kann der neue Wert des Parameters eingeben werden. Folgende Parameter können eingestellt werden:

Motor w — Winkel des Motors in Grad

Motor v — Winkelgeschwindigkeit des Motors

Motor a — Winkelbeschleunigung des Motors. Dieser Wert kann nur vor einer Simulation verändert werden. Bei der Regelung wird er nicht benutzt und während eines Simulationslaufs werden Veränderungen nicht übernommen.

Motor M — Kraft auf dem Antriebsarm

Pendulum w — Winkel des Pendels

Pendulum v — Winkelgeschwindigkeit des Pendels

Pendulum a — Winkelbeschleuning des Pendels. Dieser Wert kann nur vor einer Simulation verändert werden. Bei der Regelung wird er nicht benutzt und während eines Simulationslaufs werden Veränderungen nicht übernommen.

Calculation — Mit diesem Wert wird das Intervall festgelegt, in dem die Berechnung neuer Werte des Pendelsystems durchgeführt wird. Der Wert wird in der Einheit [1 Mikrosekunde $(1.000.000^{-1}s)$] angegeben. Falls 0 angegeben ist, wird mit maximaler Geschwindigkeit gerechnet, d.h. die Anzahl der Schritte pro Sekunde hängt nur noch von der Rechenleistung des Computers ab. Andernfalls kann die Anzahl der Rechenschritte pro Sekunde durch die Formel $\frac{1.000.000}{\text{(Wert von Calculation)}}$ berechnet werden.

Refresh Rate Dieser Wert gibt das Zeitintervall für das Ausgeben der Parameter und das Neuzeichnen des Pendelsystems an. Die Angabe erfolgt in der Einheit [1 Millisekunde ($1.000^{-1}s$)].

Network Name Hier wird der Name des gerade geladenen neuronalen Netzwerks angezeigt. Durch Anwählen des Buttons kann ein neues Netzwerk festgelegt werden, welches dann beim nächsten Start mit Start geladen wird.

A.3. Übersicht über die Hauptprogramm- und Includedateien

In Abbildung 12 werden grafisch die Abhängigkeiten der verschiedenen Module aufgezeigt. Bei der Darstellung wird die Unterteilung in die beiden Prozesse *Client* und *Server* deutlich, die für das Verständnis des Programms wichtig ist, da einzelne Programmteile nur von einem Prozeß verwendet werden.

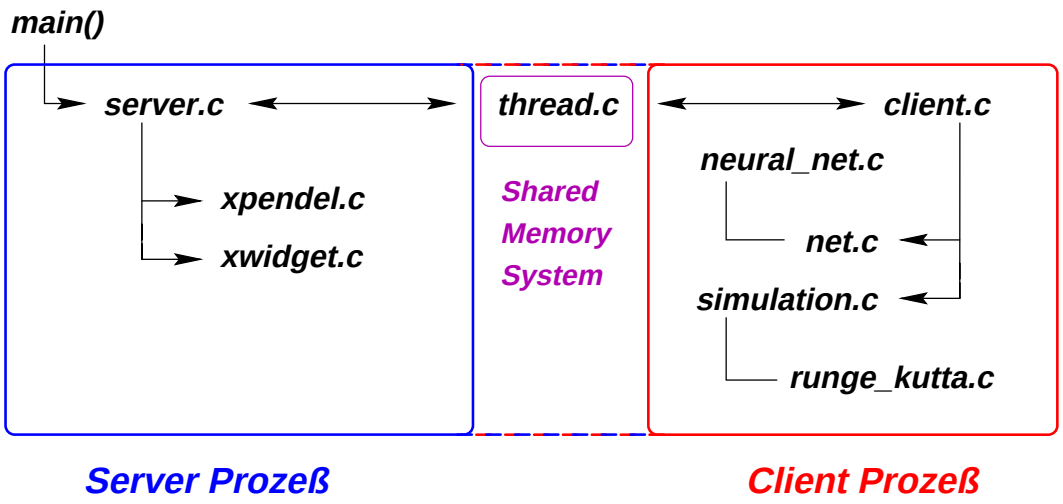


Abbildung 12: Aufbau und Abhängigkeiten der einzelnen Module

`Makefile`

Diese Makedatei ist zur Erstellung des Programms nötig. Sie stellt außerdem sicher, daß die Neuro-Netzbibliothek richtig erstellt wird.

In dem Makefile sind schon Einstellungen für verschiedene Computersysteme enthalten, die vor dem Compilieren ausgewählt werden müssen.

`XPendulum`

Dies ist die X-Window-Ressourcedatei des Programms. Sie muß in das Verzeichnis kopiert werden, in dem die Ressource Dateien-für X-Window- Anwendungen stehen (in Linux z.B. `/usr/lib/X11/app-defaults`). Alternativ kann sie auch in einem Verzeichnis stehen, welches in der Umgebungsvariable `XAPPLRESDIR` definiert ist.

A.3.1. Includedateien

`pendel.h`	Das ist die wichtigste Include-Datei. Hierin werden alle für die einzelnen Teile des Programms relevanten Daten abgelegt. Auch die von mehreren Modulen verwendeten Variablen und Strukturen werden hier definiert. Außerdem sind die Funktionsprototypen der Hauptfunktionen enthalten. In dieser Datei enthalten sind auch die Definitionen, die vom Anwender an seine Bedürfnisse angepaßt werden können.
`hardware.h`	In dieser Includedatei werden die technischen Daten der Hardware (z.B. Massen, Längen) für die Simulation definiert.
`net.h`	In dieser Includedatei werden alle Angaben, die für alle neuronalen Netz gleich sind, definiert.
`feedforward.h`	In dieser Includedatei werden alle Angaben, die für das Feedforward-Netz nötig sind, definiert.
`recurrent.h`	In dieser Includedatei werden alle Angaben, die für das Rekurrente-Netz nötig sind, definiert.
`net.h`	In dieser Includedatei werden alle Angaben, die für das Elman-Jordan-Netz nötig sind, definiert.
`xicon.h`	Hierin ist das Icon des Programms als X11-Bitmap enthalten.
`runge_kutta.h`	Diese Includedatei gehört zum `runge_kutta.c` Modul und enthält lokale Definitionen dafür.

A.3.2. Programmdateien

`thread.c` — Das Modul enthält die komplette Prozeßsteuerung und das Nachrichtensystem der beiden Prozesse (Server und Client). Es ist für den Nachrichtenaustausch zwischen `client.c` und `server.c` zuständig, weswegen es auch das Shared Memory System verwaltet.

`client.c` — In diesem Modul sind alle Funktionen des Clients enthalten, die nur von diesem benutzt werden, u.a. auch die `main()` Funktion des Clients. Das Modul dient als Bindeglied zwischen der Prozeßsteuerung `thread.c`, von der es die Nachrichten des Servers empfängt, der Simulation `simulation.c` und der Steuerung `net.c`, welches die Regelung und das neuronale Netz enthält.

Eine sehr wichtige Aufgabe dieses Moduls ist es noch, die Schnittstelle zum Gerätetreiber bereitzustellen. Über die Schnittstelle werden dem Hardwaretreiber die Befehle zur Regelung übermittelt und die Parameter des Hardwaresystems eingelesen.

`server.c` — In diesem Modul sind alle Funktionen des Servers enthalten, die nur von diesem benutzt werden, u.a. auch die `main()` Funktion des Servers. Es stellt das Bindeglied zwischen der grafischen Oberfläche, `xpendel.c` und `xwidget.c`, und der Prozeßsteuerung `thread.c` dar. Außerdem ist es dafür verantwortlich den Clientprozeß zu starten bzw. zu stoppen, wenn der Benutzer die entsprechenden Menüs auswählt.

`simulation.c` — Dies ist das Hauptmodul der Simulation. Es wird nur vom Client verwendet und berechnet die Simulation mit Hilfe des Moduls `runge_kutta.c`.

`runge_kutta.c` — Hierin sind alle Funktionen zur Berechnung des Runge–Kutta Verfahrens enthalten. Mit Hilfe dieses Verfahrens wird die Simulation durchgeführt, d.h. das in Kap. 2 beschriebene Simulationsmodell wird damit berechnet. Diese Funktionen werden von `simulation.c` verwendet.

`xpendel.c`	Dies ist das Hauptmodul der grafischen Oberfläche. Hierin ist auch die Nachrichtenschleife des X-Window Systems für das Programm enthalten. Es führt die gesamte grafische Ausgabe der Parameter aus und wird vom Anwender interaktiv bedient. Zur Darstellung des Pendelmodells im Fenster wird `xwidget.c` benutzt, welches alle Funktionen zur Darstellung enthält.
`xwidget.c`	Dieses Modul enthält die Fensterklasse (Widget) der grafischen Darstellung des Pendels. Es wird von `xpendel.c` benutzt, um das Pendel zu zeichnen.
`net.c`	In diesem Modul sind die Hauptfunktionen der Regelung des Pendelsystems enthalten, wobei aber nur der Fuzzy-Controller in `net.c` implementiert. Die neuronalen Regler dagegen sind in der dem Netztyp entsprechenden Datei enthalten.
`feedforward.c`	Hierin wird ein Feedforward-Netz implementiert, welches von `net.c` zur Regelung verwendet wird. Das Modul bietet die Schnittstelle zur Neuro-Netz-Bibliothek.
`recurrent.c`	Hierin wird ein Rekurrentes-Netz implementiert, welches von `net.c` zur Regelung verwendet wird. Das Modul bietet die Schnittstelle zur Neuro-Netz-Bibliothek.
`elman.c`	Hierin wird ein Elman-Jordan-Netz implementiert, welches von `net.c` zur Regelung verwendet wird. Das Modul bietet die Schnittstelle zur Neuro-Netz-Bibliothek.

A.3.3. Unterverzeichnisse

`learn\*` — In diesem Verzeichnis befinden sich die Quellen für das eigenständige Programm zum Lernen der verschiedenen neuronalen Netze. Dazu wird entweder eine Logdatei mit den Parametern des Pendels eingelesen, oder die Simulation verwendet, um das Netz zu lernen.

Ein gelerntes Netz kann dann mit dem **xpendulum** Programm zur Steuerung des Pendelsystems verwendet werden.

`network\*` — In diesem Verzeichnis ist die komplette Neuro-Netzbibliothek enthalten, die vom Programm zur neuronalen Steuerung verwendet wird. Informationen zur Neuro-Netzbibliothek finden sich in Tutschku (1992), Aivalis und Möhres (1996) und Scherg und Jodl (1996). Die Funktionen für rekurrente Netze werden in Heister (1996) beschrieben.

`module\*` — In diesem Verzeichnis befindet sich der Quellcode des Gerätetreibers. Diese Dateien werden im nächsten Kapitel beschrieben.

`doku\*` — In diesem Verzeichnis befindet sich die komplette Dokumentation der beiden Studienarbeiten im Postscript Format. In den Unterverzeichnissen `html_netz\*` und `html_program\*` befindet sich die Dokumentation im HTML Format, welche mit den bekannten Browsern angezeigt werden kann.

A.4. Übersicht über die Dateien des Gerätetreibers

In Abbildung 13 wird der Zusammenhang zwischen dem Programm, dem Gerätetreiber und der Hardware illustriert.

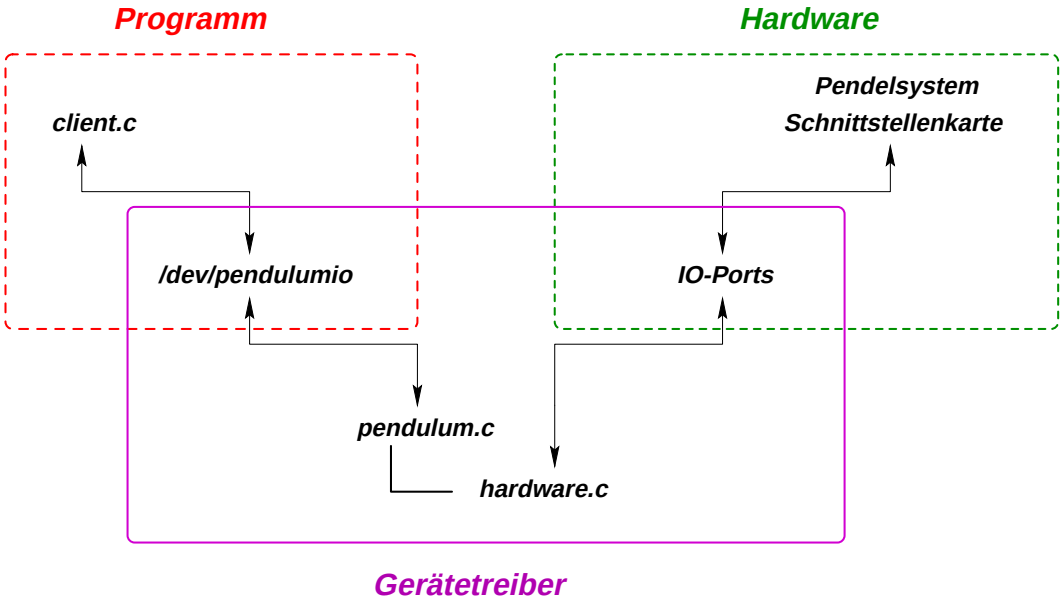


Abbildung 13: Funktionsweise des Hardwarezugriffs

`Makefile` — Das Makefile ist für die Erzeugung des Gerätetreibers als ein ladbares Modul wichtig. Alle vom Benutzer veränderbaren Einstellungen werden im Makefile festgelegt, wie z.B. der Takt des Interrupts und die Devicenummern. Es ist sehr wichtig, daß diese Nummer noch nicht vom Kernel oder anderen Modulen verwendet werden!

`pendulum.h` — Das ist die wichtigste Includedatei des Gerätetreibers. Hierin sind die Funktionsprototypen und die zum Zugriff auf die Schnittstellen nötigen Daten enthalten.

`hardware.h` — In dieser Includedatei werden alle Definitionen festgelegt, die zum direkten Zugriff auf die Hardware nötig sind. Dies ist vor allem die Datenstruktur der im Speicher eingeblendeten Schnittstelle zur Interfacekarte.

`ioctl.h` — In dieser Includedatei sind die Strukturen für den Datenaustausch zwischen dem Device und dem Treiber

enthalten. Die Datei wird sowohl vom Devicetreiber, `pendulum.c` und `hardware.c`, als auch vom Client `client.c` eingebunden, um den Datenaustausch zwischen dem Programm und dem Gerätetreiber im Kernel zu ermöglichen.

`pendulum.c` Dieses Modul enthält alle Funktionen, die zur Verwaltung eines Zeichengerätes nötig sind. Außerdem ist die Funktionalität für eine ladbares Kernel-Modul enthalten. Zur Ansteuerung der Hardware bedient es sich des Moduls `hardware.c` und stellt die Zugriffsmöglichkeiten für den Steuerprozeß bereit.

`hardware.c` Hierin sind die Funktionen der Hardwaresteuerung enthalten, die von `pendulum.c` verwendet werden. Wichtig ist dabei vor allem die Bedienfunktion für den Interrupt der Schnittstellenkarte, der für die gesamte Regelung essentiell ist.

A.5. Interface zur Hardware

Das Interface zur Hardware besteht aus einer PC ISA-Bus Karte, mit der das Hardwaresystem am Rechner angeschlossen wird, und einem Gerätetreiber unter Linux, der die Kommunikation zwischen Steuerprozeß und der Interface Karte sicherstellt. Dazu stellt der Treiber unter Linux zwei Zeichengeräte (Charakterdevices) zur Verfügung:

`/dev/pendulum` Dieses Device liefert bei jedem Lesezugriff eine Ausgabe im ASCII–Format der augenblicklichen Parameter der Hardware. Außerdem können Steuerbefehle im ASCII-Format an den Gerätetreiber geschickt werden. Die Syntax der Steuerbefehle ist bei der Beschreibung des Gerätetreibers in Kap. D.1 ersichtlich. Beispielsweise kann mit:

```
echo calibrate > /dev/pendulum
```

das Pendel kalibriert werden.

Als Default hat das Device die Nummer: 61, 0.

`/dev/pendulumio` Dieses Device stellt die Schnittstelle zwischen dem Clientprozeß, der die Hardware steuert, und dem Gerätetreiber dar. Auf das Device sind Lesezugriffe im Binärformat definiert, welche die Parameter der Hardware zurückliefern. Außerdem können mit den *ioctl()* Funktionen Befehle an die Schnittstelle gesendet werden.
Als Default hat das Device die Nummer: 61, 1.

B. Hardware

Im folgenden Kapitel wird die Hardware des Pendelsystems beschrieben und die Möglichkeiten untersucht, wie man das Pendelsystem in Echtzeit steuern kann. Als Hardware wird ein vom Ingenieurbüro *Bickele & Bühler*, Stuttgart, speziell angefertigtes System verwendet. Eine ausführliche Beschreibung der Hardware befindet sich in Anhang B und der Dokumentation zur Hardware von Bickele (1996).

B.1. Grundlegender Aufbau

Die Hardware des Pendelsystems beinhaltet drei Teile. Der wichtigste Teil ist das Pendelsystem selbst, welches aus dem freischwingenden Pendel und dem Antriebsarm besteht. Wobei am Antriebsarm noch der Motor zur Steuerung angeflanscht ist. Der Aufbau ist in Abbildung 14 dargestellt.
Der zweite sichtbare Teil stellt das Steuergerät dar, welches das Netzteil, die Leistungselektronik für den Motor und die Elektronik zur Winkelmessung enthält. Das Steuergerät ist in Abbildung 15 unter dem Pendelsystem sichtbar.
Im Steuerungsrechner eingebaut ist die Schnittstellenkarte. Die Karte stellt über eine im Adreßbereich eingeblendete Speicherseite die Schnittstelle zwischen dem Steuergerät und dem Rechner bereit.

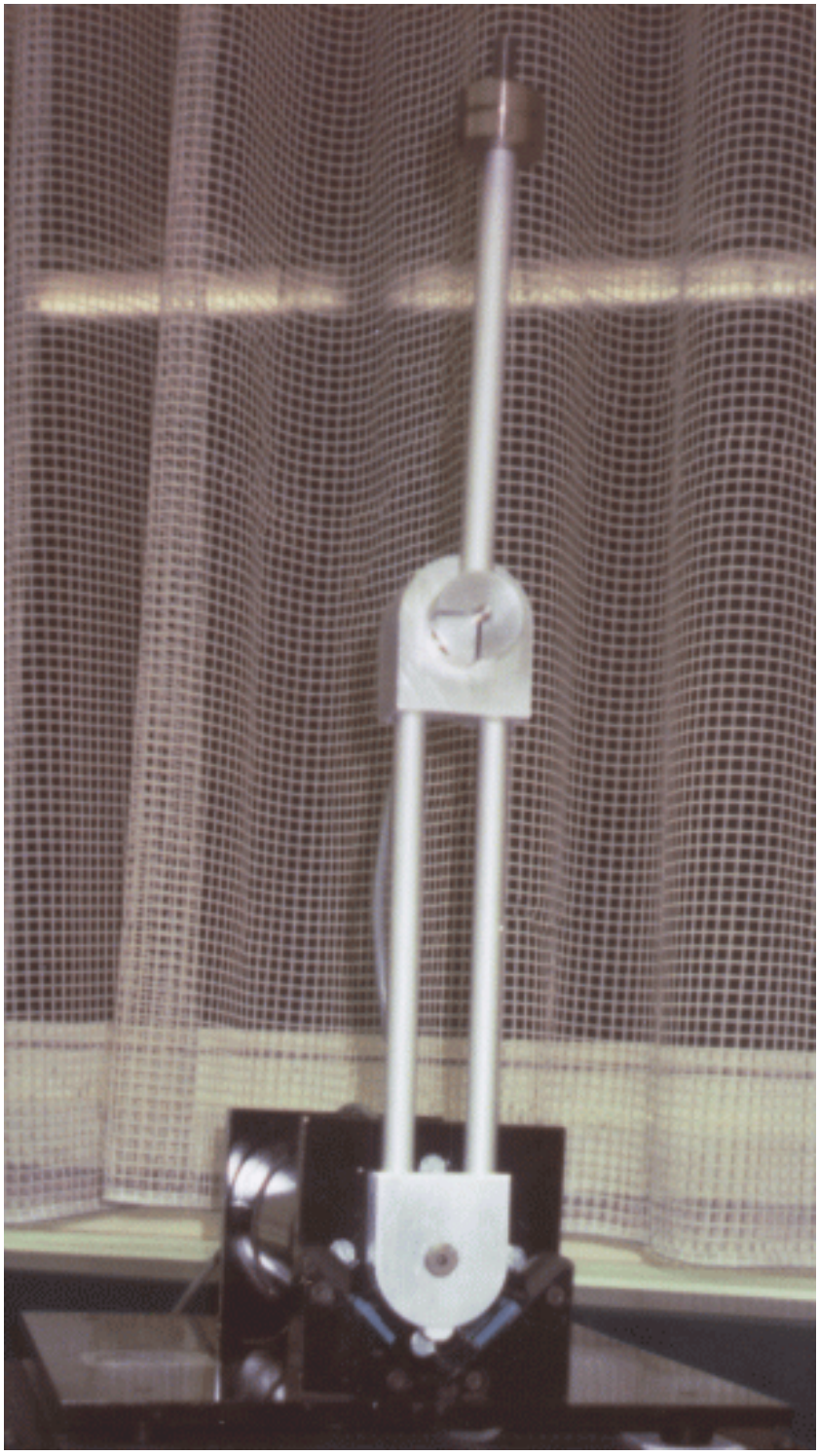
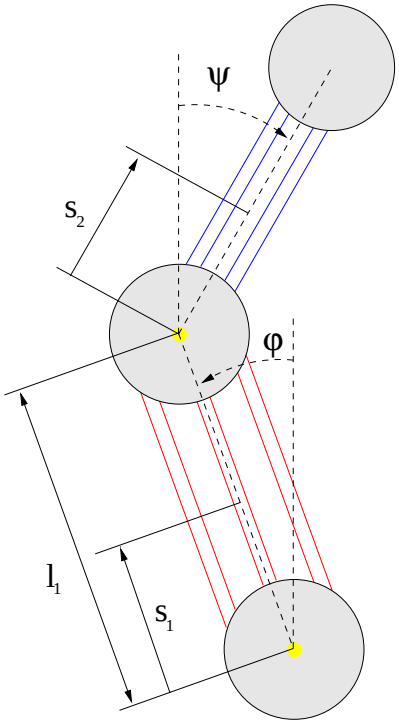


Abbildung 14: Pendelsystem

Abbildung 15: Pendelsystem mit Steuergerät

B.2. Technische Daten des Pendelsystems

Im folgenden werden die technischen Daten des Pendelsystems, wie sie vom Ingenieurbüro *Bickele & Bühler* angegeben wurden, tabelliert.



Pendelwinkel	ψ
Schwerpunktabstand	s_2
Massenträgheitsmoment des Pendels	Θ_P

Antriebsarmwinkel	φ
Armlänge	l_1
Schwerpunktabstand	s_1
Massenträgheitsmoment des Arms	Θ_A

Antriebsarm:

Masse	m_1	=	$354g$
Armlänge	l_1	=	$0.22m$
Schwerpunktabstand	s_1	=	$0.151m$
Massenträgheitsmoment	Θ_A	=	$11.0 \cdot 10^{-3} kg \cdot m^2$

Pendelarm:

	1 Gewicht	2 Gewichte	3 Gewichte	3 G. mit Schraube
Masse m_2	$95g$	$146g$	$197g$	$202g$
Schwerpunktabstand s_2	$107.5mm$	$126.6mm$	$139.5mm$	$140.2mm$

Massenträgheitsmoment $\Theta_P = 1.76 \cdot 10^{-3} kg \cdot m^2$

B.3. Technische Daten der Schnittstellenkarte

Der auf der Schnittstellenkarte eingestellte Adreßbereich und der Regelinterrupt müssen identisch mit den Werten sein, die im `Makefile` des Gerätetreibers angegeben sind! Dies ist die Vorraussetzung dafür, daß der Gerätetreiber die Schnittstellenkarte finden und ansteuern kann.

Adreßbereich:

Die Schnittstellenkarte wird nicht im IO-Bereich, sondern im Speicherbereich eingebunden und belegt dort 1 KB Speicher. Die Basisadresse der Karte kann mit dem auf der Karte plazierten DIP-Schalter in Schritten von 00400H im Bereich ab C0000H (bis EFC00H) eingestellt werden. Ein geschlossener Schalter entspricht dabei dem Wert 0. Schalter 1 stellt das höchstwertige Bit da, und Schalter 8 das Bit mit der Wertigkeit 00400H. Die Defaulteinstellung für die Adresse ist D0000H.

Regelinterrupt:

Zur Steuerung stellt die Schnittstellenkarte einen eigenen Zeitgeber zur Verfügung. Dieser löst einen Regelinterrupt aus, welcher mit Jumper 1 festgelegt werden kann. Der Jumper 1 hat folgenden Belegung:

1				IRQ 10	IRQ 11	IRQ 12	IRQ 15

Dargestellt ist hier die Jumperleiste, wie sie auf der Karte vorhanden ist. Die nicht beschrifteten Steckplätze auf der Leiste haben keine Funktion. Der Defaultwert für den IRQ ist 12.

B.4. Motorsteuerung

Der für die Motorsteuerung zuständige IC Texas Instruments *HCTL 1100* kann in verschiedenen Modi betrieben werden. Im Programm werden nur die zwei wichtigsten verwendet. Den Steuermodus legt man mit der Funktion `set_hctl_mode(mode)` fest. Für `mode` können die Parameter `MODE_INIT` oder `MODE_CONTROL` übergeben werden. Die Ansteuerung funktioniert dabei folgendermaßen:

- Im Modus `MODE_INIT` kann man den Motor direkt ansteuern, d.h. mit der Funktion `set_mcp(kraft)` wird die Spannung, die am Motor anliegt, gesetzt, was einen Kraftstoß entsprechender Größe bewirkt. Für den Parameter `kraft` können Werte im Bereich von -127 bis $+127$ übergeben werden, die linear der Kraft des Motors entsprechen. Nach dem Kraftstoß bleibt der Motor ohne Kontrolle und kann sich frei bewegen.

- Im Modus `MODE_CONTROL` wird dem Motor eine Winkelposition im Bereich von -2000 bis $+2000$ mit der Funktion `set_arm_pos(wert)` angeben, die der Motor dann ansteuert. Der Motor kann aber nur Werte von ungefähr -440 bis $+440$ aussteuern, da andernfalls der Antriebsarm auf den Anschlag trifft.

  Beim Erreichen der Zielwinkelposition bleibt der Motor auf dieser Position stehen und bleibt dort arretiert. Die Kraft die der Motor verwendet, also auch die Geschwindigkeit mit der sich der Motor bewegt, hängt nur von der Entfernung der aktuellen Position von der Zielposition ab. Es ist möglich, dem Motor schon vor Erreichen der Endposition eine neue Zielposition zu übergeben, die er dann sofort anfährt. Dies wird bei der Steuerung des Pendels mit dem Fuzzy-Controller genutzt.

B.5. Interruptservicefunktion

Die wichtigste Funktion im Gerätetreiber ist die Bedienfunktion für den Hardwareinterrupt. In ihr wird der Hauptteil der Hardware- und Prozeßsteuerung ausgeführt. Aus diesem Grund wird hier die ganze Interruptbearbeitungsfunktion erläutert:

```
void pendel_interrupt()
{
```

Mit dieser Funktion wird die Priorität des Steuerprozesses auf das Maximum gesetzt, was bewirkt, daß er beim nächsten Prozeßwechsel sofort ausgeführt wird. Da hier auf interne Kernelstrukturen zugegriffen wird, hängt die Funktion stark von der Kernelversion ab!
Das Feld `process->policy` gibt hierbei die Art des Prozesses an, wobei `SCHED_RR` die Kennung für einen Echtzeitprozeß ist. In `process->current` wird die augenblickliche Laufpriorität – nicht zu verwechseln mit der Priorität `process->priority` – gespeichert. Der Wert von 2000 sollte der maximalen Priorität entsprechen, was einem *nice* Wert von < -20 ergibt.

```
        /* this changes the process scheduling to real-time */
        /* and manipulates the task queue */
        /* the calculation process will be started by the */
        /* next schedule event */

        /* WARNING: this is very dependant on kernel version!*/
        void high_priority()
        {
                if (process) {
                        if (process_pid != process->pid) {
                                printk(KERN_ALERT "Fatal: Control
                                 process %d was lost\n", process_pid);
                                kill_proc(process->pid, SIGSEGV, 1);
                                kill_proc(process_pid, SIGSEGV, 1);
                                process = NULL;
                                return;
                        }
                        #ifdef SCHED_RR
                        process->policy  = SCHED_RR;
                        #endif
                        process->counter = process->priority = 2000;
                        need_resched = 1;
                }
        }
```

Hier wird überprüft, ob der Rechenaufwand des Steuerungsprozesses zu hoch ist. Dies ist z.B. dann der Fall, wenn die Rechenzeit für einen Steuerimpuls höher ist als

der Takt des Interrupts Δt. Dann läuft wegen der Prioritätsanpassung im Interrupt nur noch der Steuerprozeß. Alle anderen Prozesse des Rechners stehen in diesem Fall und damit scheinbar auch das System selbst. Um dies zu verhindern, wird der Steuerprozeß mit `kill(SIGSEGV)` beendet, wenn das System länger als eine Sekunde nur den Steuerprozeß ausführt.

```
/* check for overload */
if (process && process->pid == current->pid) overload++;
if (irq_counter % freq == 0) {
        if ( process && freq == overload) {
                printk(KERN_ALERT "Fatal: CPU overload! Processor to
                  slow! Process %d killed!\n", current->pid);
                kill_proc(process->pid, SIGSEGV,1);
                process = NULL;
        }
        overload = 0;
}
```

Hier wird die aktuelle Position des Pendel- und Antriebsarms ausgelesen.

```
/* save pendulum position */
 _motw = get_arm_pos();
 _penw = get_pen_pos();

 _status = get_input();
```

Falls die Steuerung eingeschaltet ist und ein Steuerimpuls vom Prozeß übergeben wurde, wird dieser an die Hardware weitergeleitet und damit der Motor geregelt.

```
/* control motor of pendulum */
 if (force_flag) {
        if (new_force) {
                skipped+= force_flag - 1;
                if (force_flag > max_skipped)
                        max_skipped = force_flag - 1;
                force_flag= 1;
                new_force = 0;
                /* control statement */
                set_arm_pos(force);
        } else {
                high_priority();
                force_flag++;
        }
 }
```

Da die Geschwindigkeit des Pendel- und Antriebsarms für die Regelung benötigt wird, aber nicht direkt von der Hardware gemessen werden kann, wird sie hier berechnet. Dazu werden die Positionen des Pendels und Antriebsarms über `VCOUNTER` Interrupttakte gespeichert und über die Positionsänderungen durch die Zeit die Geschwindigkeit berechnet.

```
        /* calculate pendulum speed */
        memcpy(&pen_speed[0], &pen_speed[1], sizeof(int)
                  * (VCOUNTER - 1 ));
        memcpy(&mot_speed[0], &mot_speed[1], sizeof(int)
                  * (VCOUNTER - 1 ));
        pen_speed[VCOUNTER-1] = _penw;
        mot_speed[VCOUNTER-1] = _motw;

        _penv = (_penw - pen_speed[0]) * freq / VCOUNTER;
        _motv = (_motw - mot_speed[0]) * freq / VCOUNTER;

        irq_counter++;
```

Falls der Steuerprozeß gerade schläft und auf neue Parameter von der Hardware wartet, wird er hiermit aufgeweckt, damit er einen Steuerimpuls für den nächsten Interrupt berechnen kann.

```
       /* wakeup process if sleeping */
        if (wakeup) {
                sleeping--;
                if (sleeping < 0) {
                        high_priority();
                        wake_up_interruptible(&pendel_waitq);
                }
        }
}
```

C. Linux Gerätetreiber

Für einen Linux Gerätetreiber sind prinzipiell zwei Formen denkbar:

Im Kernel integrierte Treiber

Diese werden mit dem Kernel geladen und sind somit jederzeit benutzbar. Alle Treiber, die zur Initialisierung des Linux-System benötigt werden, müssen im Kernel integriert sein.

Ladbare Gerätetreiber

Diese Treiber werden erst während der Laufzeit des Linux-Systems geladen. Dazu dienen die Programme aus dem *Modutils*-Paket, welches man als modules-2.0.0.tar.gz auf den bekannten Linux-Servern finden kann. Mit deren Hilfe kann man einzelne Treiber laden und auch wieder entfernen. Die geladenen Treiber werden in die Kernelstrukturen eingebunden und können dann alle Kernelfunktionen benutzen. Nach dem Registrieren der ladbaren Treiber durch den Kernel besteht faktisch kein Unterschied mehr zu den im Kernel integrierten. Die ladbaren Treiber können auch wieder zur Laufzeit entfernt werden, wenn sie nicht mehr in Gebrauch sind. Dies spart zum einen Speicher. Viel wichtiger ist aber, daß man dadurch Treiber zur Laufzeit des Systems ersetzen kann, was bei der Treiberentwicklung ein sehr großer Vorteil ist.
Aus diesen Gründen stehen in neueren Kerneln fast alle integrierten Treiber von Linux auch als ladbares Modul zur Verfügung. Beim Übersetzen des Kernels kann man wählen, in welcher Form die Treiber bereitgestellt werden. Außerdem gibt es noch das Programm `kerneld`, welches in Verbindung mit dem Kernel das automatische Laden von Kernel-Modulen ermöglicht. Im Kernel selber sind dann nur noch die Treiber, die zum Systemstart benötigt werden. Alle anderen werden bei Bedarf geladen, z.B. beim Zugriff auf das Diskettenlaufwerk oder ein bestimmtes Dateisystem. Wenn der Treiber dann einige Zeit nicht mehr benutzt wird, wird er automatisch wieder entfernt. Dies ermöglicht sehr kleine Kernel mit voller Funktionalität.
Die Treiber für spezielle externe Hardware, z.B. Scanner oder andere Controller, werden immer als Modul ausgeführt. Ein Beispiel hierfür ist der in Kap. C.4 beschriebene Treiber für einen Joystick oder auch der in Kap. D beschriebene Treiber für die Schnittstellenkarte des Pendelsystems.

C.1. Erstellung von Gerätetreibern als Modul

Die ladbaren Treiber liegen als Objektdatei vor, die mit dem Programm `insmod` geladen werden. Dabei kann sowohl das *a.out*, als auch das *ELF*-Format genutzt werden, falls im Kernel die Unterstützung für das jeweilige Format aktiviert ist. Mit `rmmod` kann ein Treiber wieder entfernt werden, falls er gerade nicht benutzt wird. Diese Programme sind in dem Modutils-Paket enthalten (aktuelle Version ist modules-2.0.0.tar.gz). Das Paket bietet noch weitere Befehle zur angenehmen Modulverwaltung. Näheres dazu ist in der Dokumentation des Modutils-Pakets enthalten.
Um einen Gerätetreiber als Modul zu schreiben, sind verschiedene Dinge zu beachten. Zum einen braucht man ein spezielles Makefile, um ein Modul richtig zu Compilieren, zum anderen muß auch im Quellcode selbst eine bestimmte Form eingehalten werden.

C.2. Hardwarezugriff

In einem Gerätetreiber muß man natürlich auch die Hardware des Geräts ansprechen können. Dazu stellt der Kernel verschiedene Funktionen bereit, mit deren Hilfe man die Hardware-Ressourcen, z.B. IO-Ports, verwalten und ansprechen kann.
Normalerweise belegt man beim Start des Gerätetreibers, also in `init_module()`, alle Ressourcen, die das Gerät braucht. Dazu geht man folgendermaßen vor:

1. Mit `check_region(int from, int num)` fragt man ab, ob der entsprechende Portbereich von `from` bis `from + num` frei ist, d.h. nicht von anderen Treibern verwendet wird.

2. Falls er frei ist, wird mit Hilfe von `request_region(int from, int num, const char *name)` der Bereich für den Treiber reserviert, wobei `name` einen aussagekräftigen Namen für den Portbereich angibt. Die belegten Portbereiche werden unter `/proc/ioports` angezeigt.

3. Wenn der Treiber entfernt werden soll, also beim Aufruf von `cleanup_module()`, muß der Portbereich wieder freigegeben werden. Dies geschieht mittels `release_region(int from, int num)`, welches den entsprechenden Portbereich wieder freigibt.

Auf die IO-Ports kann dann mittels `inb(port), inw(port), outb(data, port)` oder `outw(data, port)` zugegriffen werden, mit denen jeweils ein Byte (`inb, outb`) oder ein Wort (`inw, outw`) vom Port `port` gelesen, oder der Wert `data` auf den Port `port` geschrieben wird.
Falls man Interrupts belegen will, versucht man während der Initialisierung mit der Funktion

```
request_irq(unsigned int irq,
     void (*handler)(int, void *, struct pt_regs *), long flags,
     char *device, void *dev_id)
```

einer Interruptfunktion für den angegeben Interrupt zu belegen. Wenn der Aufruf nicht fehlschlägt, wird die Funktion `handler` bei jedem Interrupt aufgerufen. Mit `free_irq(int irq, void *dev_id))` gibt man den Interrupt wieder frei, wenn der Treiber entfernt wird.
Wenn man Memory-Mapped Hardware ansprechen will, kann man direkt Variablen auf die Speicherbereiche der Hardware legen, wenn der Speicherbereich im ersten Megabyte, auch Adapterbereich genannt, liegt. Mit folgender Codesequenz bekommt man z.B. eine Variable, mit der man auf dem Adapterbereich an Adresse D0000H wordweise zugreifen kann:

```
short *wert;
int address = 0xD0000;

werte = (short *)address;
```

Weiterhin stehen im Kernel Funktionen bereit, um Speicherbereiche (`kmalloc()`) oder DMA Kanäle (`request_dma()`) zu belegen. Die Funktionsweise ist im Quellcode des Kernels ersichtlich, z.B. im Soundtreiber oder verschiedenen Blocktreibern.

C.3. Makefile eines Gerätetreibers

```
# Makefile for Linux kernel module driver

# major device number
# this must be unused in the linux kernel!
MAJOR   := 63

CC      := gcc -Wall -fomit-frame-pointer -fno-strength-reduce -pipe

# if you have enabled CONFIG_MODVERSION when compiling the kernel
# you must add -DMODVERSIONS after -DMODULE!
MFLAGS  := -D__KERNEL__\
           -DMODULE -DMODVERSIONS\
           -D__NO_VERSION__\
           -DBSP_MAJOR=$(MAJOR)\

CFLAGS  := -O2

joystick.o:     joystick.c
        $(CC) -o joystick.o $(CFLAGS) $(MFLAGS) -c joystick.c

clean:
        rm -f *.o

device:
        rm -r /dev/joystick; true
        mknod -m 777 /dev/joystick c $(MAJOR) 0

load:
        /sbin/insmod joystick

unload:
        /sbin/rmmod joystick
```

Mit diesem Makefile läßt sich ein allgemeiner Gerätetreiber, der ein bestimmtes Device steuert, erstellen. Zum Compilieren des Gerätetreibers reicht ein Aufruf von `make` aus. Zusätzlich werden Regeln definiert, um den Gerätetreiber zu laden (`make load`), wieder zu entfernen (`make unload`) und das Device anzulegen (`make device`).

Erläuterung der einzelnen Compileroptionen:

`-O2 -fomit-frame-pointer` schaltet den Optimierer des Compilers an.

`-Wall` schaltet alle Warnmeldungen des Compilers an. Dies ist nützlich, um Programmfehler leichter zu finden.

`-pipe` dient zur Beschleunigung des Compilerlaufs.

`-fno-strength-reduce` verhindert etwaigen fehlerhaften Code bei eingeschalteter Optimierung wegen eines Bugs im `gcc` Compiler (mindestens bis Version 2.7.2).

`-D__KERNEL__` ist nötig, um auf interne Kerneldefinitionen in den Headerdateien zugreifen zu können.

`-DMODULE -DMODVERSIONS` definiert die Flags für Module. Dadurch wird dem Modul der Zugriff auf Funktionen des laufenden Kernels ermöglicht. Die Option `MODVERSIONS` muß immer dann angeben werden, wenn der Kernel mit eingeschalteter Option `CONFIG_MODVERSIONS` übersetzt wurde.

`-D__NO_VERSION__` ist aus Kompatibilitätsgründen nötig. In neueren Kernelversionen wird die Variable `kernel_version` in den Headerdateien definiert, falls diese Option nicht angegeben wird. Die Variable wird beim Übersetzen mit der augenblicklichen Kernelversion initialisiert. Sie wird von dem Modutils-Paket benötigt, um sicherzustellen, daß der Kernel, in dem das Modul eingebunden werden soll, dem entspricht, mit dem das Modul übersetzt wurde. Die internen Kernelstrukturen unterscheiden sich nämlich bei verschiedenen Kernelversionen.

Da der Treiber aber auch mit älteren Kerneln laufen soll, wird `kernel_version` in der Quelldatei definiert und mit `__NO_VERSION__` sichergestellt, daß die Variable bei neueren Kerneln nicht doppelt definiert wird.

C.4. Quellcode eines Gerätetreibers

Der Quellcode eines Gerätetreiber muß auch bestimmte Bedingungen erfüllen. Wichtig ist hierbei zunächst, daß `linux/module.h` als erste Headerdatei eingebunden wird! Die Definition `char kernel_version[] = UTS_RELEASE;` ist, wie im vorigen Kapitel beschrieben, nur aus Kompatibilitätsgründen nötig. In den neuesten Kerneln ist dies schon in den Headerdateien enthalten. Hierzu ein Beispiel für einen Gerätetreiber, der das Einlesen des Joystick Ports ermöglicht:

```
/************************************************************************/
/*      This is a sample device driver for Linux                        */
/*      It implements a joystick device                                 */
/*      written by Marius Heuler                                        */
/************************************************************************/

#define BSP_NAME "joystick"

#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif
#include <linux/module.h>
#include <linux/version.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/ioport.h>
#include <linux/kernel.h>
#include <linux/sched.h>

/************************************************************************/
/*      Local Prototypes                                                */
/************************************************************************/

static void ReadJoystick(int *x0, int *y0, int *x1, int *y1, int *bt);

static int bsp_open(struct inode*, struct file*);
static void bsp_close(struct inode*, struct file*);
static int bsp_read(struct inode*, struct file*, char*, int);
static int bsp_write(struct inode*, struct file*, const char*, int);
static int bsp_ioctl(struct inode*, struct file*, uint, ulong);
```

```
int init_module(void);
void cleanup_module(void);

char kernel_version[] = UTS_RELEASE;    /* kernel version */

int major = BSP_MAJOR;  /* major device id */

static struct file_operations bsp_fops = {
        NULL,           /* lseek */
        bsp_read,       /* read */
        bsp_write,      /* write */
        NULL,           /* readdir */
        NULL,           /* select */
        bsp_ioctl,      /* ioctl */
        NULL,           /* mmap */
        bsp_open,       /* open */
        bsp_close,      /* close */
        NULL,           /* fsync */
        NULL,           /* fasync */
        NULL,           /* check media_change */
        NULL            /* revalidate */
};

/*************************************************************************/
/*      This is called when loading the module                           */
/*************************************************************************/

int init_module()
{
        int error;

        /* register the device to the kernel */
        if ((error = register_chrdev(major, BSP_NAME, &bsp_fops))) {
                printk(KERN_ALERT
                  "Joystick device %d is already registered!\n", major);
                return error;
        }
        printk(KERN_DEBUG "Joystick driver loaded on device %d\n", major);
        return(error);
}

/*************************************************************************/
/*      This is called when unloading the module                         */
```

```
/*************************************************************************/

void cleanup_module()
{
        printk(KERN_DEBUG "Joystick driver unloaded\n");
        unregister_chrdev(major, BSP_NAME);
}

/*************************************************************************/
/*      This is the handler for read requests                            */
/*************************************************************************/

static int bsp_read(struct inode *ip, struct file *fp, char* to, int len)
{
        int x0, y0, x1, y1, bt;
        int error;
        int minor = MINOR(ip->i_rdev);
        static char sbuf[100];

        /* Check for sanity */
        if ((error = verify_area(VERIFY_WRITE, to, len))) return error;
        if (minor != 0)
                return(-ENODEV);

        ReadJoystick(&x0, &y0, &x1, &y1, &bt);
        sprintf(sbuf, "x0:%+5d  y0:%+5d     x1:%+5d  y1:%+5d"
                      "     bt0:%d   bt1:%d   bt2:%d   bt3:%d\n",
                        x0, y0, x1, y1,
                        (bt & 16) ? 0:1,
                        (bt & 32) ? 0:1,
                        (bt & 64) ? 0:1,
                        (bt & 128) ? 0:1);
        memcpy_tofs(to, sbuf, strlen(sbuf));
        return(strlen(sbuf));
}

/*************************************************************************/
/*      This is the handler for write requests                           */
/*************************************************************************/

static int bsp_write(struct inode *ip, struct file *fp, const char* from, int len)
{
        return -EINVAL;
```

```
}

/*************************************************************************/
/*      IOctl handler of device                                          */
/*************************************************************************/

static int bsp_ioctl(struct inode *ip, struct file *fp, uint cmd, ulong arg)
{
        return -EINVAL;
}

/*************************************************************************/
/*      Open handler of module                                           */
/*************************************************************************/

static int bsp_open(struct inode *ip, struct file *fp)
{
        int minor = MINOR(ip->i_rdev);

        if (minor != 0)
                return(-ENODEV);
        MOD_INC_USE_COUNT;
        return 0;
}

/*************************************************************************/
/*      Close handler of module                                          */
/*************************************************************************/

static void bsp_close(struct inode *ip, struct file *fp)
{
        int minor = MINOR(ip->i_rdev);

        if (minor != 0)
                return;
        MOD_DEC_USE_COUNT;
}

/*************************************************************************/
/*      Read data from joystick port                                     */
/*************************************************************************/

static void ReadJoystick(int *x0, int *y0, int *x1, int *y1, int *bt)
```

```
{
        int t, w;

        t = jiffies;
        *x0 = 0;
        *y0 = 0;
        *x1 = 0;
        *y1 = 0;
        outb(0, 0x201);
        w = inb(0x202);
        *bt = (w & (16+32+64+128));

        while (t == jiffies) {
                w = inb(0x201);
                if (w & 1) (*x0)++;
                if (w & 2) (*y0)++;
                if (w & 4) (*x1)++;
                if (w & 8) (*y1)++;
        }
}
```

Beim Laden des Moduls wird die Funktion `init_module()` vom Kernel aufgerufen, die die Initialisierung und Einbindung in den Kernel vornimmt. Im Erfolgsfall muß sie 0 zurückliefern. Die Funktion `cleanup_module` wird beim Entfernen des Treiber aufgerufen. Sie muß alle vom Treiber benötigten Ressourcen wieder freigeben.
Gerätetreiber benötigen eine Gerätekennung (Major Device ID), die eindeutig sein muß. In der Datei `devices.txt` im Linux-Kernel werden die verschiedenen belegten IDs aufgelistet. Für einen neuen Treiber muß eine freie ID verwendet werden!
Um nun ein Zeichengerät zu registrieren, wird die Funktion `register_chrdev(major, name, &file_ops)` aufgerufen. Normalerweise wird dies beim Laden des Moduls in `init_module()` durchgeführt. Bei den Parametern gibt `major` die obige ID an. Mit `name` wird der Name des Geräts festgelegt, u.a. für das Modutils-Paket. Unter `/proc/devices` werden alle geladenen Gerätetreiber mit Namen und Gerätekennung aufgelistet.

Die Struktur `file_ops` gibt die Fähigkeiten des Geräts an. Sie ist so aufgebaut:

```
struct file_operations {
        int (*lseek) (struct inode *, struct file *, off_t, int);
        int (*read) (struct inode *, struct file *, char *, int);
        int (*write) (struct inode *, struct file *, const char *int);
        int (*readdir) (struct inode *, struct file *, void *,
                        filldir_t);
        int (*select) (struct inode *, struct file *, int,
                       select_table *);
        int (*ioctl) (struct inode *, struct file *, unsigned int,
                      unsigned long);
        int (*mmap) (struct inode *, struct file*, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
        int (*fasync) (struct inode *, struct file *, int);
        int (*check_media_change) (dev_t dev);
        int (*revalidate) (dev_t dev);
};
```

Die Felder enthalten jeweils Zeiger auf C-Funktionen mit den entsprechenden Parametern. Falls eine Funktion nicht unterstützt wird, gibt man `NULL` an. Die wichtigsten Funktionen werden nun kurz beschrieben:

`lseek` Wählt die Position innerhalb einer auf dem Gerät geöffneten Datei an (siehe `lseek()` C-Funktion).

`read` Liest aus der geöffneten Datei Daten (siehe `read()` C-Funktion).

`write` Schreibt in die geöffnete Datei Daten (siehe `write()` C-Funktion).

`ioctl` Verändert Parameter der geöffneten Datei (siehe `ioctl()` C-Funktion).

`open` Öffnet eine Datei auf dem Gerät (siehe `open()` C-Funktion).

`close` Schließt eine Datei auf dem Gerät (siehe `close()` C-Funktion).

`sync` Schreibt alle Daten der geöffneten Datei auf das Gerät. (siehe `sync()` und `fsync()` C-Funktionen).

Nachdem die Funktionalität der einzelnen Funktionen implementiert wurde, ist die Schnittstelle zu Programmen fertig.
Bei der Erstellung eines Gerätetreibers müssen aber unbedingt folgende Dinge beachtet werden:

- Es muß sichergestellt werden, daß der Treiber nur entfernt werden kann, wenn keine Ressource des Treibers, z.B. durch eine auf dem Device geöffnete Datei, von Programmen mehr verwendet wird. Dies wird durch das Einfügen von `MOD_INC_USE_COUNT` im C-Quellcode an den Stellen, an denen der Treiber eine Ressource bereitstellt, z.B. in der `open()` Funktion. Bei dem Freigeben von Ressourcen wird entsprechend `MOD_DEC_USE_COUNT` eingefügt, z.B. in der `close()` Funktion. Ein Treiber kann nur entfernt werden, wenn der interne Zähler, der von `MOD_*_USE_COUNT` verändert wird, auf 0 steht, also der Treiber im Moment nicht benutzt wird.

  Die Anweisungen `MOD_INC_USE_COUNT` und `MOD_DEC_USE_COUNT` müssen immer als Paar auftreten! Und zwar jeweils dann, wenn die Ressource belegt wird und wenn sie wieder freigegeben wird.

- Da der Treiber im Kernel-Modus läuft, kann nicht einfach auf die Daten von Prozessen, die ja im User-Modus laufen, zugegriffen werden, z.B. bei `read()` oder `write()` Aufrufen. Bei Zeichen/Blocktreibern kann man die Funktionen
  `memcpy_fromfs(void *kernel_data, void *user_data, int size)`
  und
  `memcpy_tofs(void *kernel_data, void *user_data, int size)`
  verwenden. Sie kopieren den Speicherbereich der Länge `size` von der Stelle `user_data` im User-Modus an `kernel_data` im Kernel-Modus oder zurück. Diese und weitere Funktionen für den Datenaustausch sind im Headerfile `segment.h` des Kernels definiert.

- Wenn auf Speicherbereiche im User-Modus von Programmen zugegriffen werden soll, z.B. mit `memcpy_*()`, muß sichergestellt sein, daß die Bereiche existieren. Sonst würde beim Zugriff ein Seitenfehler (*Page Fault*) im Kernel-Modus auftreten, was das System mit einen Kernel-Panik anhalten würde. Um nun einen Speicherbereich zu prüfen, gibt es die Funktion `verify_area(int type, void *user_data, long size)`, welche einen Bereich der Länge `size` an der Adresse `user_data` des User-Modus überprüft. Bei `type` kann `VERIFY_WRITE` zur Prüfung auf Schreibzugriff oder `VERIFY_READ` zur Prüfung auf Lesezugriff übergeben werden.

Um den Gerätetreiber zu benutzen muß nur noch mittels `insmod joystick` der Treiber geladen werden. Nun kann ein Programm mittels eines Device, z.B. /dev/joystick, welches die Gerätekennung (Major Device ID) des Treibers hat, auf den Gerätetreiber zugreifen. Ein Device kann mittels des `mknod` Befehls oder der `mknod()` C-Funktion erzeugt werden.

D. Gerätetreiber - Datenstrukturen und Funktionen

D.1. Beschreibung der globalen Datenstrukturen und Definitionen

Kennungen für das Versionskontroll-System:

```
$Id: pendulum.h,v 1.114 1996/04/23 16:34:23 heuler$
$Id: hardware.h,v 1.2 1996/03/16 21:46:42 heuler$
$Id: ioctl.h,v 1.122 1996/03/16 21:47:23 heuler$
```

Alle Dateien für den Gerätetreiber befinden sich in dem Verzeichnis `module`. Die Definitionen, die sowohl vom Gerätetreiber, als auch vom Programm benötigt werden, sind in der Datei `ioctl.h` enthalten. Für den direkten Hardwarezugriff benötigte Definitionen sind in der Datei `hardware.h` enthalten. Wohingegen `pendulum.h` allgemeine Definitionen für Gerätetreiber enthält. Über die ASCII-Schnittstelle des Gerätetreibers (`/dev/pendulum`) können direkt Befehle an das Pendelsystem geschickt werden. Die möglichen Befehle sind im folgenden mit aufgeführt.

Beispiele:

```
echo calibrate > /dev/pendulum
echo interrupt=2000 > /dev/pendulum
```

`#define PENDEL_NAME "pendulum"`	Name des Pendel Gerätetreibers
`#define PENDULUM "pendulum interface"`	Name der Pendel Hardware
`#define IOVERSION 0x105`	Version der Schnittstelle zwischen Treiber und Programm; wird sowohl vom Programm, als auch vom Treiber verwendet, um sicherzustellen, daß die binäre Schnittstelle kompatibel ist
`#define TIMEOUT 30`	legt die maximale Zeitspanne in Sekunden für Befehle fest
`#define VCOUNTER 10`	gibt die Anzahl Interrupts für die Berechnung der Geschwindigkeit an

Über die `Ioctl()` Schnittstelle werden folgende Befehle definiert, die an den Gerätetreiber geschickt werden können.

`#define CMD_OFF`	fährt das Pendel herunter; entspricht dem Befehl: `down`
`#define CMD_CALIB`	calibriert das Pendel in der Nullposition; entspricht dem Befehl: `calibrate`
`#define CMD_UP`	schwingt das Pendel hoch; entspricht dem Befehl: `up=parameter`
`#define CMD_MODEINIT`	setzt den Motormodus auf `MODEINIT`; entspricht dem Befehl: `modeinit`
`#define CMD_MODERESET`	setzt den Motormodus auf `MODERESET`; entspricht dem Befehl: `modereset`
`#define CMD_MODECONTROL`	setzt den Motormodus auf `MODECONTROL`; entspricht dem Befehl: `modecontrol`
`#define CMD_ENDON`	schaltet die Endabschaltung an; entspricht dem Befehl: `endon`
`#define CMD_ENDOFF`	schaltet die Endabschaltung aus; entspricht dem Befehl: `endoff`
`#define CMD_PENCONTROL`	schaltet die Steuerung an; entspricht dem Befehl: `control`
`#define CMD_MOVE`	bewegt den Pendelarm auf die angebene Position; entspricht dem Befehl: `move=parameter`
`#define CMD_DIRECT`	gibt einen direkten Impuls auf den Motor; entspricht dem Befehl: `direct=parameter`
`#define CMD_INTERRUPT`	setzt den Takt für den Interrupt; entspricht dem Befehl: `interrupt=parameter`
`#define CMD_DELAY`	wartet angegebene Zeit lang; entspricht dem Befehl: `delay=parameter`
`#define CMD_CONTROL`	legt Steuerbefehl für nächsten Interrupt ab
`#define CMD_STARTCTRL`	startet die Steuerung durch den Interrupt
`#define CMD_STOPCTRL`	stopt die Steuerung durch den Interrupt

Für die Motorsteuerung sind folgende Modi definiert, die über die Funktion `set_hctl_mode` gesetzt werden können.

`#define MODE_RESET`	setzt den Motor komplett zurück und schaltet die Steuerung aus
`#define MODE_INIT`	schaltet den Motor in den direkten Modus
`#define MODE_CONTROL`	schaltet den Motor in den Steuermodus

Über die Funktion `get_input()` können die verschiedenen Zustände des Pendelsystems abgefragt werden. Die Zustände sind dabei bitweise definiert und können mit folgenden Definitionen überprüft werden.

`#define IN_ENDAN`	gibt an, ob das Steuergerät eingeschaltet ist
`#define IN_NULLA`	gibt an, ob der Antriebsarm in der Nullposition liegt; ist zur Kalibration nötig
`#define IN_NULLP`	gibt an, ob das Pendel in der Nullposition liegt; ist zur Kalibration nötig
`#define IN_LIML`	gibt an, ob der Antriebsarm am linken Rand, also kurz vor dem linken Anschlag ist
`#define IN_LIMR`	gibt an, ob der Antriebsarm am rechten Rand, also kurz vor dem rechten Anschlag ist
`#define IN_FREI`	gibt an, ob die Endstufe der Motorsteuerung freigeschaltet ist

D.1.1. Wichtige Datenstrukturen

Die `io_outstruct` wird benutzt um die aktuellen Daten des Pendelsystem vom Gerätetreiber zu lesen. Das binäre Device `/dev/pendulumio` verwendet diese Struktur, die in `ioctl.h` definiert ist, als Datenformat.

`typedef struct {`	
`int version;`	Version der Schnittstelle
`int count;`	Zähler für Zugriff
`int lost;`	Anzahl der Steuerimpulse, die verlorengingen
`int delay;`	Maximale Anzahl der Impulse zwischen zwei Steuerungen
`int motw;`	Winkel des Antriebsarms
`int penw;`	Winkel des Pendels
`int motv;`	Winkelgeschwindigkeit des Antriebsarms
`int penv;`	Winkelgeschwindigkeit des Pendels
`} io_outstruct;`	

D.1.2. Wichtige Variablen

`int major;`	Gerätenummer
`int irq;`	Interruptnummer der Schnittstellenkarte Default: 12
`int base;`	Basisadresse des Speicherbereichs der Karte Default: 0xD0000
`int freq;`	Takt des Interrupts der Karte
`volatile int irq_counter;`	Zähler für Interrupts
`volatile int skipped;`	Anzahl der Interrupts, die zwischen zwei Steuerbefehlen verlorengingen
`volatile int max_skipped;`	maximale Anzahl Interrupts, die zwischen zwei Befehlen verlorengingen
`struct task_struct *process;`	Prozeßstruktur aus dem Kernel
`int process_pid;`	Prozeßkennung des Steuerprozesses
`volatile int _penw;`	Pendelwinkel in Hardwareeinheiten
`volatile int _penv;`	Pendelgeschwindigkeit in Hardwareeinheiten
`volatile int _motw;`	Antriebsarmwinkel in Hardwareeinheiten
`volatile int _motv;`	Antriebsarmgeschwindigkeit in Hardwareeinheiten
`volatile int _status;`	Status der Hardware - bitweise

`volatile int force;`	aktueller Steuerbefehl
`volatile int force_flag;`	Flag, ob Steuerung eingeschaltet

D.2. Funktionen der einzelnen Module

D.2.1. Datei: pendulum.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: pendulum.c,v 1.141 1996/04/23 16:34:01 heuler$";
```

Globale Variablen:

```
char kernel_version[] = UTS_RELEASE;    /* kernel version */

int major = PENDEL_MAJOR;       /* major device id */
int irq   = PENDEL_IRQ;         /* pendulum card irq */
int base  = PENDEL_BASE;        /* pendulum card base */
int freq  = PENDEL_FREQ;        /* frequency of update */

volatile int force;             /* force for pendulum in next IRQ */
volatile int force_flag;        /* force controlling */
volatile int old_irqcounter;    /* old counter of irq */
volatile int skipped;           /* number of irqs skipped */
volatile int max_skipped;       /* max number of irqs skipped */

struct task_struct *process;    /* controlling process */
int process_pid;                /* controlling process pid */
```

Lokale Variablen:

```
static int lock_ctrl;           /* this flag is set when the
                                   ctrl device is openend */

static io_outstruct outdata;    /* struct for output of iodata */

static struct file_operations pendel_fops = {
        NULL,           /* lseek */
        pendel_read,    /* read */
        pendel_write,   /* write */
        NULL,           /* readdir */
        NULL,           /* select */
        pendel_ioctl,   /* ioctl */
        NULL,           /* mmap */
        pendel_open,    /* open */
        pendel_close,   /* close */
        NULL,           /* fsync */
        NULL,           /* fasync */
        NULL,           /* check media_change */
        NULL            /* revalidate */
};
```

Funktionen:

Callback – Wird aufgerufen, wenn das Modul geladen wird:

```
int init_module();
```

Callback – Wird aufgerufen, wenn das Modul entfernt wird:

```
void cleanup_module();
```

Callback – Wird bei Lesezugriffen auf das Device aufgerufen:

```
int pendel_read(struct inode *ip, struct file *fp, char* to, int
len);
```

Callback – Wird bei Schreibzugriffen auf das Device aufgerufen:

```
int pendel_write(struct inode *ip, struct file *fp, const char*
from, int len);
```

Callback – Wird bei Kontrollzugriffen auf das Device aufgerufen:

```
int pendel_ioctl(struct inode *ip, struct file *fp, uint cmd,
ulong arg);
```

Callback – Wird beim Öffnen des Device aufgerufen:

```
int pendel_open(struct inode *ip, struct file *fp);
```

Callback – Wird beim Schließen des Device aufgerufen:

```
void pendel_close(struct inode *ip, struct file *fp);
```

Bearbeitet ASCII-Befehle mit Argumenten:

```
void command_sarg(char *str, int cmd, int fl);
```

Bearbeitet ASCII-Befehle ohne Argumente:

```
int parse_command(char *str);
```

D.2.2. Datei: hardware.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: hardware.c,v 1.143 1996/05/08 11:02:46 heuler$";
```

Globale Variablen:

```
volatile int _penw;             /* angle of pendulum - set by irq */
volatile int _motw;             /* angle of motor - set by irq */
volatile int _penv;             /* velocity of pendulum - set by irq */
volatile int _motv;             /* velocity of motor - set by irq */
volatile int _status;           /* status of system - set by irq */

volatile int new_force    = 0;  /* flag for new force */

volatile int irq_counter  = 0;  /* counter of io irqs */
volatile int sleeping     = 0;  /* sleeptime in freq units */
volatile int wakeup       = 0;  /* process should be waked up */
volatile int overload     = 0;  /* processor overload */
```

Lokale Variablen:

```
volatile struct hctl *hctl_1, *hctl_2;
volatile ubyte *port_A, *port_B, *port_C, *p_cont;
volatile ubyte *count_0, *count_1, *count_2, *t_cont;

static int sample_timer = 0x40;
static struct wait_queue *pendel_waitq = NULL;

static int pen_speed[VCOUNTER]; /* for speed of pendulum */
static int mot_speed[VCOUNTER]; /* for speed of motor */

static int calibrated = 0;      /* calibrated */
```

Funktionen:

Setzt die Schnittstellenkarte zurück:

```
void reset_card();
```

Programmiert die Interruptrate auf der Karte:

```
void set_intr_timer(int f);
```

Schaltet den Interrupt von der Karte an:

```
void start_intr();
```

Schaltet den Interrupt von der Karte aus:

```
void stop_intr();
```

Initialisiert die Speicher-Adressen der Schnittstellenkarte:

```
void init_address();
```

Prüft, ob die Karte vorhanden ist:

```
static int hardware_test();
```

Initialisiert die Hardware:

```
int init_hardware();
```

Schaltet die Hardware ab:

```
void shutdown_hardware();
```

Interrupt-Callback - Interruptbedienfunktion:

```
void pendel_interrupt();
```

Führt direkten Motorbefehl aus:

```
void set_mcp(int wert);
```

Schaltet den Endstufen-Endschalter aus:

```
void endstufe_on();
```

Schaltet den Endstufe-Endschalter ein:

```
void endstufe_reset();
```

Stellt den Motormodus ein:

```
void set_hctl_mode(int mode);
```

Setzt die Regelparameter:

```
void set_param(ubyte werta, ubyte wertb, ubyte wertk);
```

Setzt die Sollposition für den Arm:

```
void set_arm_pos(long pos);
```

Setzt den Positionszähler für den Arm auf 0:

```
void null_arm_pos();
```

Setzt den Positionszähler des Pendels auf 0:

```
void null_pendel_pos();
```

Liefert Position des Arms zur Senkrechten:

```
long get_arm_pos();
```

Liefert Position des Pendels zum Arm:

```
long get_pen_pos();
```

Liest Digitalsignale von Endstufe:

```
ubyte get_input();
```

Wartet `time` Millisekunden durch Schlafen:

```
void delay(int time);
```

Fährt Pendel herunter:

```
void pendel_down();
```

Fährt Pendel hoch und kalibriert es:

```
int pendel_calib();
```

Bewegt den Arm langsam:

```
void move_arm(long ziel);
```

Schwingt den Arm hoch:

```
int pendel_up(int faktor);
```

Steuert das Pendel mit dem Fuzzy-Controller:

```
void control();
```

Führt Befehle mit Argumenten aus:

```
void command_arg(int cmd, int value, int fl);
```

Führt Befehle ohne Argumente aus:

```
void command(int cmd, int fl);
```

E. Beschreibung der globalen Datenstrukturen und Definitionen

E.1. Vom Benutzer an seine Bedürfnisse anpaßbar

Kennung für das Versionskontroll-System:

```
$Id: pendel.h,v 1.8 1996/07/04 13:43:55 heuler$
```

`#define DEBUG 1`	schaltet die Debugging Ausgabe ein, wenn auf 1 gesetzt
`#define LOGFILE "log_param"`	Name der Logdatei
`#define LOGFORMAT`	Ausgabeformat für die Logdatei
`#define NETNAME "Pendulum"`	Name des Netzes
`#define WWIDTH 700`	Breite des Programmfensters beim Start
`#define WHEIGHT 600`	Höhe des Programmfenster beim Start
`#define SERVERSLEEP 100`	legt fest, in welchem Abstand die Ausgabe der Parameter und die Darstellung des Pendels erfolgt; die Angabe erfolgt in $\frac{1}{1.000}$ Sekunden und kann über das Menü während des Programmlaufs geändert werden
`#define CLIENTSLEEP 1000`	legt fest, in welchem Abstand die Berechnung der Parameter und die Steuerung des Pendels erfolgt; die Angabe erfolgt in $\frac{1}{1.000.000}$ Sekunden und kann über das Menü während des Programmlaufs geändert werden; ein Wert von 0 schaltet die kontinuierliche Berechnung ohne Zeitverzögerung ein.
`#define RUNGESTEP 0.001`	Schrittweite des Runge-Kutta Verfahren

E.2. Shared Memory System

`#define TIMERES 1000000`	Auflösung der Uhr bei `gettimeofday()`
`#define MSG_PENDATA`	Nachricht: Neue Pendelparameter Wird zum Server geschickt!
`#define MSG_START`	Nachricht: Client starten
`#define MSG_STOP`	Nachricht: Client stoppen
`#define MSG_RESET`	Nachricht: Client zurücksetzen
`#define MSG_SETALL`	Nachricht: Alle Daten des Pendels definieren
`#define MSG_STARTLOG`	Nachricht: Logen starten
`#define MSG_STOPLOG`	Nachricht: Logen stoppen
`#define MSG_SIMULATION`	Nachricht: auf Simulation umschalten
`#define MSG_HARDWARE`	Nachricht: auf Steuerung der Hardware umschalten
`#define MSG_NETNAME`	Nachricht: neues Netz einladen
`#define MSG_NEURO`	Nachricht: auf neuronales Netz umschalten
`#define MSG_FUZZY`	Nachricht: auf Fuzzy-Controller umschalten
`#define MSG_NOCONTROL`	Nachricht: Steuerung abschalten
`#define MSG_CONTROL`	Nachricht: Steuerung einschalten
`#define MSG_NEWPARAM`	Nachrichtengruppe: Parameter neu setzen; die Nachricht wird durch `MSG_NEWPARAM` + (ID des Parameters) festgelegt

`#define IDMOTW`	Winkel des Motors
`#define IDMOTV`	Winkelgeschwindigkeit des Motors
`#define IDMOTA`	Winkelbeschleunigung des Motors
`#define IDPENW`	Winkel des Pendels
`#define IDPENV`	Winkelgeschwindigkeit des Pendels
`#define IDPENA`	Winkelbeschleunigung des Pendels

`#define IDFORCE`	Kraft auf dem Antriebsarm
`#define IDNET`	Netznamen ändern, siehe `MSG_NETNAME`
`#define IDDISP`	Anzeigeintervall ändern, siehe `SERVERSLEEP`
`#define IDSLEEP`	Rechenintervall ändern, siehe `CLIENTSLEEP`

`#define COLORPEN "Blue"`	Farbe des Pendeles
`#define COLORMOT "Red"`	Farbe des Motors
`#define COLORMECH "Black"`	Farbe der Mechanikteile
`#define COLORNAIL "Yellow"`	Farbe der Drehachsen

E.3. Technische Daten der Hardware aus `hardware.h`

Kennung für das Versionskontroll-System:

```
$Id: hardware.h,v 1.202 1996/04/23 16:55:39 heuler$
```

`#define GRAV 9.80665`	Gravitationskonstante
`#define MASS1 0.354`	Masse des Antriebsarms
`#define MASS2 0.146`	Masse des Pendels bei 2 Gewichten
`#define LEN1 0.22`	Länge des Antriebsarms
`#define LEN2 0.15`	Länge des Pendels
`#define THETA 0.011`	Massenträgheitsmoment des Antriebsarms
`#define THETP 0.00176`	Massenträgheitsmoment des Pendels bei 2 Gewichten
`#define THETM 0.0`	Massenträgheitsmoment des Motors
`#define S1 0.151`	Entfernung vom Schwerpunkt des Antriebsarms zum Drehpunkt
`#define S2 0.1266`	Entfernung vom Schwerpunkt des Pendels zum Drehpunkt

E.4. Wichtige Datenstrukturen

Mit der `pendatastruct` werden jeweils die augenblicklichen Daten des Pendelsystem vom Client an den Server geschickt. Die Struktur wird mit der Nachricht `MSG_PENDATA` mitgeschickt.

```
typedef struct {
        double motw;        Winkel des Motors
        double motv;        Winkelgeschwindigkeit des Motors
        double mota;        Winkelbeschleunigung des Motors
        double penw;        Winkel des Pendels
        double penv;        Winkelgeschwindigkeit des Pendels
        double pena;        Winkelbeschleunigung des Pendels
        double force;       Kraft auf den Antriebsarm
} pendatastruct;
```

Die `extradatastruct` enthält die zusätzlichen Daten, die mit einer Nachricht geschickt werden. Da diese Daten von verschiedenem Format sein können, wird eine `union` verwendet.

```
typedef union {
       pendatastruct d;        Parameter des Pendelsystems
       char name[80];          Name, des zu ladenden Netzwerks
} extradatastruct;
```

Die `shmstruct` wird in den beiden Shared Memory Bereichen angelegt. Sie ist jeweils einmal in der Richtung *Client* → *Server* und einmal in Richtung *Server* → *Client* vorhanden. Dies ist nötig, um die Kommunikation zwischen den beiden Prozessen zu entkoppeln.

```
typedef struct {
        int semaphor;          Semaphor für den ausschließenden Zugriff
                               auf die gesamte Struktur
        int ack;               Quittierung vom Empfänger nach Empfang
                               der Nachricht
        int count;             Zähler über die geschickten Nachrichten
        int command;           übertragene Nachricht
        int datalen;           Länge der Extradaten der Nachricht
        pendatastruct d;       Extradaten mit variabler Länge
} shmstruct;
```

Die `setparmstruct` wird in `xpendel.c` für die Parametereingabe unter X-Window benötigt. Mit ihr wird die Dialogbox definiert, die beim Anklicken einer der Parameter aufgerufen wird.

```
typedef struct {
        Widget parent;          Vater Widget
        String label;           Ausgabe beim Darstellen des Dialogs
        int id;                 Parameter der geändert werden soll
} setparmstruct;

typedef setparmstruct *psetparm;
```

Die folgenden Strukturen werden in `xwidget.c` verwendet und dienen dazu eine eigene Fensterklasse, Widget genannt, zu definieren. Diese Fensterklasse ist dann für die Anzeige des Pendels zuständig.

```
typedef struct {
        int dummy_field;
} PendulumClassPart;

typedef struct {
    CoreClassPart       core_class;
    PendulumClassPart   pendulum_class;
} PendulumClassRec;

typedef struct {
        int dummy_field;
} PendulumPart;

typedef struct {
    CorePart            core;
    PendulumPart        pendulum;
} PendulumRec;

typedef PendulumClassRec *PendulumWidgetClass;
typedef PendulumRec *PendulumWidget;
```

E.5. Wichtige Variablen

Shared Memory System

`FILE *logfile;`	Datei Deskriptor für die Logdatei
`int smid;`	Kennung für den Shared Memory Bereich
`int lastmsg;`	Nummer der letzten Nachricht; wird benötigt um festzustellen, ob schon eine neue Nachricht eingetroffen ist
`shmstruct *shmbuf;`	Zeiger auf den Shared Memory Bereich
`shmstruct *out_shmbuf;`	Shared Memory Bereich für ausgehende Nachrichten
`shmstruct *in_shmbuf;`	Shared Memory Bereich für ankommende Nachrichten

Grafische Ausgabe - `xpendel.c` Modul

`double xmotorw;`	angezeigter Winkel des Motors
`double xmotorv;`	angezeigte Winkelgeschwindigkeit des Motors
`double xmotora;`	angezeigte Winkelbeschleunigung des Motors
`double xpendelw;`	angezeigter Winkel des Pendels
`double xpendelv;`	angezeigte Winkelgeschwindigkeit des Pendels
`double xpendela;`	angezeigte Winkelbeschleunigung des Pendels
`double xforce;`	angezeigte Kraft auf den Antriebsarm
`XtAppContext app_context;`	Kontext der Anwendung

Grafische Ausgabe - `xwidget.c` Modul

`int pendulumwidth;`	Breite des Pendelfensters
`int pendulumheight;`	Höhe des Pendelfensters
`WidgetClass pendulumWidgetClass;`	Fensterklasse des Pendelfensters

Allgemeine globale Variablen

`int loging;`	Logfile an oder aus
`int running;`	Client Prozeß läuft gerade oder nicht
`int online;`	Anzeige der Parameter an oder aus
`int sleeprate;`	legt fest, in welchem Abstand die Ausgabe der Parameter und die Darstellung des Pendels erfolgt; wird mit `SERVERSLEEP` bei Programmstart vorbelegt
`int clientsleep;`	Zeitspanne, die der Client zwischen jedem Schritt bei der Simulation bzw. Regelung schläft; Variable im Client Prozeß
`int clients;`	Zeitspanne, die der Client zwischen jedem Schritt bei der Simulation bzw. Regelung schläft; wird mit `CLIENTSLEEP` bei Programmstart vorbelegt; Variable im Server Prozeß
`int simulation;`	Simulation oder Hardware; Variable im Client Prozeß
`int csimu;`	Simulation oder Hardware; Variable im Server Prozeß
`int control;`	Steuerung aktiviert; Variable im Client Prozeß
`int ccontrol;`	Steuerung aktiviert; Variable im Server Prozeß

`int neuro;`	Neuronales Netz oder Fuzzy-Steuerung; Variable im Client Prozeß
`int cneuro;`	Neuronales Netz oder Fuzzy-Steuerung; Variable im Server Prozeß
`char *name;`	Name des augenblicklichen Prozesses für Client und Server verschieden
`char *netname`	Name des geladenen Netzes

Die folgenden Variablen werden im Client verändert. Bei der Simulation werden sie durch das Simulationsmodell angepaßt. Bei der Hardwaresteuerung werden sie von der Hardware eingelesen. Die Parameter werden dann mit einer `MSG_PENDATA` Nachricht an den Server geschickt, der nach Empfang seine Variablen im Modul `xpendel.c` anpaßt und gegebenfalls neu anzeigt.

`double motw;`	augenblicklicher Winkel des Motors
`double motv;`	augenblickliche Winkelgeschwindigkeit des Motors
`double mota;`	augenblickliche Winkelbeschleunigung des Motors
`double penw;`	augenblicklicher Winkel des Pendels
`double penv;`	augenblickliche Winkelgeschwindigkeit des Pendels
`double pena;`	augenblickliche Winkelbeschleunigung des Pendels
`double force;`	augenblickliche Kraft auf den Antriebsarm

F. Funktionen der einzelnen Module

F.1. Datei: thread.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: thread.c,v 1.114 1996/04/23 16:54:26 heuler$";
```

Globale Variablen:

```
int smid;               /* shared memory id */
int lastmsg;            /* last message count */
int workprocid;         /* server work proc */

shmstruct *shmbuf;      /* shared memory */
shmstruct *in_shmbuf;   /* shared memory input for current thread */
shmstruct *out_shmbuf;  /* shared memory output for current thread */
```

Funktionen:

Gibt den Shared Memory Bereich frei:

```
void DeleteShm();
```

Setzt die Variablen bei jedem Client-Start zurück:

```
void ResetShm();
```

Initialisiert den Shared Memory Bereich:

```
void InitShm();
```

Verschickt eine Nachricht. Die Funktion kann sowohl vom Client, wie vom Server verwendet werden und schickt die Nachricht an den jeweils anderen Prozeß:

```
void SendMsg(int command, void *data, int len);
```

Wartet, bis die letzte verschickte Nachricht vom anderen Prozeß gelesen wurde. Normalerweise werden die Nachrichten asynchron verschickt!:

```
void WaitAcknowledge();
```

Schaut nach, ob eine neue Nachricht bereitliegt und liefert sie gegebenenfalls mit ihren Zusatzdaten zurück:

```
int GetMsg(int *command, void *data, int maxlen);
```

F.2. Datei: server.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: server.c,v 1.107 1996/04/23 16:53:41 heuler$";
```

Funktionen:

Schickt alle Parameter des Pendelsystems zum Client; wird nur beim Start des Clients aufgerufen:

```
void SendtoClient();
```

Schickt dem Client die Nachricht, den Parameter `id` zu ändern:

```
void UpdateParmChild(int id, double f);
```

Schickt dem Client die Nachricht, sich zu beenden:

```
void StopChild();
```

Schickt dem Client einen *Reset* Befehl:

```
void ResetChild();
```

Startet den Client mittels eines `fork()`, initialisiert seinen Programmkontext und übergibt im Client die Kontrolle an `ClientMain`:

```
void StartChild();
```

Setzt die Parameter des Pendels für die grafische Ausgabe und zeigt sie an:

```
void SetParameters(pendatastruct *d);
```

Initialisiert den Server beim Programmstart:

```
void InitServer();
```

`main()` des Servers, wird vom X11-System in bestimmten Zeitabständen aufgerufen, siehe `sleeprate`, Kap. E.5:

```
void ServerMain();
```

`main()` des Hauptprogramms:

```
int main(int argc, char **argv);
```

F.3. Datei: client.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: client.c,v 1.306 1996/06/19 17:30:00 heuler$";
```

Globale Variablen:

```
FILE *logfile;          /* descriptor for logging */
int clientsleep;        /* client sleep time */
int devfd;              /* descriptor of open device */
int control;            /* control enabled */
int neuro;              /* neuro or fuzzy control */
int cmd_lost;           /* lost commands by hardware */
int max_cmd_lost;       /* max number of lost commands */
```

Funktionen:

Sendet die aktuellen Parameter des Pendelsystems an den Server:

```
void SendtoServer();
```

Setzt die Parameter des Pendels im Client zurück:

```
void ResetChildParm();
```

Setzt die Parameter des Pendelsystems im Client auf die vom Server gesendeten Werte:

```
void SetChildParm(pendatastruct *d);
```

Startet das Mitschreiben der Pendelparameter in die Logdatei:

```
void StartLog();
```

Stopt das Mitschreiben der Pendelparameter in die Logdatei:

```
void StopLog();
```

Öffnet das Device zum Hardware-Gerätetreiber:

```
void InitHardware();
```

Schließt das Device zum Hardware-Gerätetreiber:

```
void CloseHardware();
```

Liest mittels des Devices von der Hardware die aktuellen Werte des Pendelsystems ein:

```
void ReadHardware();
```

Schickt einen Positionierbefehl an die Hardware:

```
void WriteHardware();
```

Schickt einen Befehl an den Gerätetreiber:

```
void CommandIO(int cmd, int arg);
```

Schreibt die aktuellen Parameter des Pendelsystems in das Logfile:

```
void LogParam();
```

Beendet den Client:

```
void KillChild();
```

Wertet die vom Server empfangene Nachricht aus und ruft die entsprechenden Funktionen auf:

```
void ParseCommands(int command, extradatastruct *data);
```

`main()` des Clients, wird nach dem Prozeßstart vom Server aufgerufen:

```
void ClientMain();
```

Spezielle Funktionen für den hochauflösenden Timer

Lokale Variablen:

```
static int start;               /* start of last calculation */
static struct timeval tv;       /* timer structure */
static int oldsec = 0;          /* last second */
static int calc   = 0;          /* count of calculations per second */
```

Funktionen:

Setzt den Zeitgeber zurück. Wird vor jedem Rechenschritt aufgerufen:

```
void TimerStart();
```

Wartet, bis die in `clientsleep`, siehe Kap. E.5, angebene Zeit abgelaufen ist. Diese Funktion wird nach jedem Rechenschritt aufgerufen und mißt die Zeit von `TimerStart` bis zum augenblicklichen Zeitpunkt:

```
void TimerSleepNext();
```

F.4. Datei: xpendel.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: xpendel.c,v 1.306 1996/07/04 10:19:43 heuler$";
```

Globale Variablen:

```
static psetparm pdata;

int clients;            /* client sleep time = clientsleep in client */
int csimu;              /* simulation or hardware = simulation in client */
int ccontrol;           /* control enabled = control in client */
int cneuro;             /* neuro of fuzzy control */

char netname[80];       /* name of net */

int pendulumwidth;                      /* width of pendulum */
int pendulumheight;                     /* height of pendulum */

double xmotorw, xmotorv, xmotora;       /* displayed paramters of motor */
double xpendelw, xpendelv, xpendela;    /* displayed parameters of pendulum */
double xforce;                          /* displayed force of motor */

int online;                             /* display information */
int loging;                             /* file loging */

XtAppContext app_context;
```

Lokale Variablen:

Variablen für die verschiedenen Fenster (Widgets) auf der X-Oberfläche:

```
static Widget topLevel;
static Widget wquit, wstart, wreset, wswitch, wdisp, wapopup, wlog,
              wcontr, wneuro;
static Widget wsetparmlabel, wsetparmdata, wsetparmDone;
static Widget wmotorw, wmotorv, wmotora;
static Widget wpendelw, wpendelv, wpendela;
static Widget wcmotorw, wcmotorv, wcmotora;
static Widget wcpendelw, wcpendelv, wcpendela;
static Widget wforce, wcforce;
static Widget wcdis, wdis;
static Widget wcclient, wclient;
static Widget wcnetn, wnetn;
static Widget wpendulum;
```

Funktionen:

MenuCallback – Beendet den Client und dann das Hauptprogramm:

```
void Quit(Widget w, XtPointer client_data, XtPointer call_data);
```

MenuCallback – Startet den Client oder beendet ihn:

```
void Start(Widget w, XtPointer client_data, XtPointer call_data);
```

MenuCallback – Schaltet zwischen Simulation und Steuerung der Hardware um:

```
void Switch(Widget w, XtPointer client_data, XtPointer
call_data);
```

MenuCallback – Schaltet die Steuerung ein oder aus:

```
void NetControl(Widget w, XtPointer client_data, XtPointer
call_data);
```

MenuCallback – Schaltet zwischen dem neuronalen Netz und dem Fuzzy-Controller um:

```
void NetNeuro(Widget w, XtPointer client_data, XtPointer
call_data);
```

MenuCallback – Startet bzw. stoppt das Mitschreiben der Pendelparameter in das Logfile:

```
void FileLog(Widget w, XtPointer client_data, XtPointer
call_data);
```

MenuCallback – Schaltet die Anzeige aus bzw. an:

```
void Disp(Widget w, XtPointer client_data, XtPointer call_data);
```

MenuCallback – Setzt die Parameter und den Client zurück:

```
void Reset(Widget w, XtPointer client_data, XtPointer call_data);
```

Zeigt die Parameter des Pendels im Fenster an:

```
void DisplayParameters();
```

Schaltet die Buttons ab, wenn ein Dialogfenster geöffnet wird:

```
void DisableButtons();
```

Schaltet die Buttons wieder an, nachdem das Dialogfenster geschlossen wurde:

```
void EnableButtons();
```

Callback – Öffnet das Dialogfenster zur Parametereingabe:

```
void SetParmPopup(Widget w, XtPointer client_data, XtPointer
call_data);
```

Callback – Wird beim Schließen des Dialogfensters aufgerufen und schickt die geänderten Parameter an den Client:

```
void SetParmDone(Widget w, XtPointer client_data, XtPointer
call_data);
```

Callback – Wird aufgerufen, wenn der Anwender Cancel im Dialogfenster auswählt:

```
void SetParmCancel(Widget w, XtPointer client_data, XtPointer
call_data);
```

Callback – Wird aufgerufen, wenn der Anwender die Eingabe abschließt. Ruft nur `SetParmDone()` auf:

```
void SetParmReturn(Widget w, XEvent *d1, String *d2, Cardinal
*d3);
```

Erzeugt alle Fenster der X-Oberfläche, legt deren Aussehen fest, richtet die Unterfenster für die einzelnen Parameter ein und registriert die *Callback*-Funktionen für die Buttons:

```
void create_boxes(Widget parent, XtAppContext app_context);
```

Richtet einen Programmkontext ein, öffnet das Hauptfenster und springt in die Hauptnachrichtenschleife der X-Oberfläche:

```
void InitX(int argc, char **argv);
```

F.5. Datei: xwidget.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: xwidget.c,v 1.113 1996/07/10 11:26:25 heuler$";
```

Lokale Definitionen:

Struktur für die Pendel Fensterklasse:

```
PendulumClassRec pendulumClassRec = {
        {
        /* core_class fields */
        /* superclass            */ (WidgetClass) &widgetClassRec,
        /* class_name            */ "Pendulum",
        /* widget_size           */ sizeof(PendulumRec),
        /* class_initialize      */ NULL,
        /* class_part_initialize */ NULL,
        /* class_inited          */ FALSE,
        /* initialize            */ Initialize,
        /* initialize_hook       */ NULL,
        /* realize               */ XtInheritRealize,
        /* actions               */ NULL,
        /* num_actions           */ 0,
        /* resources             */ NULL,
        /* num_resources         */ 0,
        /* xrm_class             */ NULLQUARK,
        /* compress_motion       */ TRUE,
        /* compress_exposure     */ TRUE,
        /* compress_enterleave   */ TRUE,
        /* visible_interest      */ FALSE,
        /* destroy               */ Destroy,
        /* resize                */ Resize,
        /* expose                */ Redisplay,
        /* set_values            */ NULL,
        /* set_values_hook       */ NULL,
        /* set_values_almost     */ XtInheritSetValuesAlmost,
        /* get_values_hook       */ NULL,
        /* accept_focus          */ NULL,
        /* version               */ XtVersion,
        /* callback_private      */ NULL,
        /* tm_table              */ NULL,
        /* query_geometry        */ NULL,
};
```

Lokale Variablen:

```
WidgetClass pendulumWidgetClass = (WidgetClass) & pendulumClassRec;

static GC gc, xgc;              /* display gcs */
static Pixmap penpixmap;        /* pendulum pixmap */
static Display *display;        /* display variable */
static int screen;              /* screen number of display */
static int cdepth;              /* color depth of display */
static Colormap cmap;           /* default colormap */

static int pixel_pen;           /* pixel color of pendulum */
static int pixel_mot;           /* pixel color of motor */
static int pixel_mech;          /* pixel color of mechanic */
static int pixel_back;          /* pixel color of background */
static int pixel_nail;          /* pixel color of nail in circle */

static double oxmotorw;         /* old angle of motor */
static double oxpendelw;        /* old angle of pendulum */

static int cwidth;              /* width of circle */
static int motlen;              /* length of motor */
static int penlen;              /* length of pendulum */
static int rwidth;              /* width of rectangle */
static int xpos;                /* xpos of pendulum */
static int ypos;                /* ypos of pendulum */
```

Funktionen:

Zeichnet ein gefülltes, gedrehtes Rechteck ähnlich wie `XFillRectangle()`:

```
XPoint XFillRectangleAngle(Display *display, Window window, GC
gc, int xp, int yp, int width, int height, double sina, double
cosa);
```

Berechnet die Größe der einzelnen Parameter des Pendels abhängig von der Fenstergröße:

```
void CalculatePenParam(PendulumWidget w);
```

Zeichnet den Motorteil des Pendelsystems:

```
XPoint DrawMotor(Window window, Boolean draw);
```

Zeichnet den Pendelteil des Pendelsystems:

```
void DrawPendulum(Window window, int xpos, int ypos, Boolean
draw);
```

Löscht das Pendelsystem an der alten Position und zeichnet es an der neuen:

```
void DisplayPendulum(PendulumWidget w, Window window);
```

Callback – Wird bei der Initialisierung der Fensterklasse aufgerufen:

```
void Initialize(Widget treq, Widget xw, ArgList args, Cardinal
*num_args);
```

Callback – Wird bei einer Größenänderung des Pendelfensters aufgerufen:

```
void Resize(Widget xw);
```

Callback – Wird beim Schließen des Pendelfensters aufgerufen:

```
void Destroy(Widget w);
```

Callback – Wird aufgerufen, wenn das Pendelfenster neu gezeichnet werden muß:

```
void Redisplay(Widget xw, XExposeEvent *event);
```

F.6. Datei: simulation.c

Kennung für das Versionskontroll-System:

```
 char *rcsid = "$Id: simulation.c,v 1.139 1996/04/23 16:54:12 heuler
Exp$";
```

Lokale Definitionen:

```
const double
        g = GRAV,       /* gravitational constant */
        m1 = MASS1,     /* mass of motorarm */
        m2 = MASS2,     /* mass of pendulum */
        l1 = LEN1,      /* length of motorarm */
        l2 = LEN2,      /* length of pendulum */
        thetaA = THETA, /* inertia of motorarm */
        thetaP = THETP, /* inertia of pendulum */
        thetaM = THETM, /* inertia of motor */
        s1 = S1,        /* distance from centre of rotation of motorarm
                           to centre of gravity */
        s2 = S2,        /* distance from centre of rotation of pendulum
                           to centre of gravity */
```

Globale Variablen:

```
int simulation;          /* hardware oder simulation */
int running;             /* simulation running ? */
int display;             /* displaying information */
int sleeprate;           /* time to sleep for display */

char *name;              /* name of thread */

double force;            /* current parameter of force */
double zeit = 0.0;       /* time */
double h = RUNGESTEP;    /* step size for Runge-Kutta method */

double motw, motv, mota; /* current parameters of motor */
double penw, penv, pena; /* current parameters of pendulum */
                         /* w : angle
                          * v : angle speed
                          * a : acceleration
                         */
```

Lokale Variablen:

```
static double A, B, C, D, E, F, G, H, L, N;
        /* variables to speed up calculation */
```

Funktionen:

Liefert die Kraft für den Antriebsarm zurück (für das Runge-Kutta Verfahren):

```
double M(double t);
```

Berechnet die Beschleunigung des Antriebsarms (für das Runge-Kutta Verfahren):

```
double phipp(double t, double phi, double phip, double psi,
double psip);
```

Berechnet die Beschleunigung des Pendels (für das Runge-Kutta Verfahren):

```
double psipp(double t, double phi, double phip, double psi,
double psip);
```

Initialisiert die Parameter, die für die Simulation benötigt werden:

```
void InitSimulation();
```

Berechnet die neuen Parameter des Pendelsystems für jeden Zeitschritt:

```
void Simulate();
```

F.7. Datei: runge_kutta.c

Kennung für das Versionskontroll-System:

```
 char *rcs = "$Id: runge_kutta.c,v 1.0 1996/01/15 17:46:55 heuler
Exp $";
```

Funktionen:

Berechnet die Runge Kutta Funktion mit den übergebenen Parametern:

```
void Runge_Kutta (func xpp, func ypp, double *t, double *x,
double *xp, double *y, double *yp, double *h);
```

F.8. Datei: net.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: net.c,v 1.123 1996/05/08 11:13:24 heuler$";
```

Lokale Variablen:

```
static double  integral = 0.0; /* for controlling */
```

Funktionen:

Fuzzy-Controller des Pendels. Steuert das Pendel, wenn **Fuzzy** in der Menüleiste gewählt ist, siehe Kap. A.2.1:

```
double fuzzy_control();
```

Berechnet die neue Kraft auf den Antriebsarm. Ruft dazu den Fuzzy-Controller oder die entsprechenden Funktionen der verschiedenen neuronalen Netze auf:

```
void CalculateNewForce();
```

F.9. Datei: elman.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: elman.c,v 1.2 1996/05/08 11:22:47 heuler$";
```

Globale Variablen:

```
NET             *net;
SAMPLE          *sample;
SAMPLE_LIST     *sample_list;
NODE            *output_node;
```

Funktionen:

Lädt ein Elman-Jordan-Netz:

```
int Elman_LoadNet(char *netname);
```

Löscht das Netz:

```
void Elman_KillNet();
```

Propagiert das Netz und liefert die neue Kraft zurück:

```
double Elman_Next();
```

F.10. Datei: recurrent.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: recurrent.c,v 1.101 1996/05/08 11:24:04 heuler$";
```

Globale Variablen:

```
static NET     *net;           /* the neural net */

static SAMPLE  *probe;         /* the probe */
static NODE    *output_node;   /* the output node of the net */
static double  eta     = NET_ETA;
static double  alpha   = NET_ALPHA;
static double  theta   = NET_THETA;
static double  MaxWert = NET_MAXWERT;
static double  seed    = NET_SEED;
```

Funktionen:

Erzeugt ein neues Rekurrentes-Netz:

```
void Recurrent_CreateNet();
```

Initialisiert das Rekurrente-Netz:

```
void Recurrent_InitNet();
```

Löscht das Netz:

```
void Recurrent_KillNet();
```

Lädt ein Rekurrentes-Netz:

```
int Recurrent_LoadNet(char *netname);
```

Propagiert das Netz und liefert die neue Kraft zurück:

```
double Recurrent_Next();
```

F.11. Datei: feedforwared.c

Kennung für das Versionskontroll-System:

```
char *rcs = "$Id: feedforward.c,v 1.101 1996/05/08 11:23:24 heuler$";
```

Globale Variablen:

```
static NET     *net;           /* the neural net */
static SAMPLE  *probe;         /* the probe */
static NODE    *output_node;   /* the output node of the net */
static double  eta     = NET_ETA;
static double  alpha   = NET_ALPHA;
static double  theta   = NET_THETA;
static double  MaxWert = NET_MAXWERT;
static double  seed    = NET_SEED;
```

Funktionen:

Erzeugt ein neues Feedforward-Netz:

```
void Feedforward_CreateNet();
```

Initialisiert das Feedforward-Netz:

```
void Feedforward_InitNet();
```

Löscht das Netz:

```
void Feedforward_KillNet();
```

Lädt ein Feedforward-Netz:

```
int Feedforward_LoadNet(char *netname);
```

Propagiert das Netz und liefert die neue Kraft zurück:

```
double Feedforward_Next();
```

G. Literaturverzeichnis

Aivalis, G. und H. Möhres (1996). Elman–Jordan Netze. Projektpraktikum Neuronale Identifikation und Regelung am Lehrstuhl für Informatik III, Universität Würzburg, Institut für Informatik.

Bickele, A. (1996). Hardwaredukomentation zum Pendelsystem. Technical report, Ingenieurbüro Bickele & Bühler, 70469 Stuttgart, Triebweg 25, 0711 / 8560441.

Heister, F. (1996). Recurrent Multilayer Perceptrons for Identification and Control with Extended Kalman Filter Training. Diplomarbeit, Universität Würzburg, Institut für Informatik.

Hieronymous, A. (1993). *UNIX-Systemarchitektur und Programmierung*. Braunschweig/Wiesbaden: Vieweg Verlag.

Laufer, R. (1996). Neuronale Echtzeitregelung eines doppeltinversen Pendels – Netze und Lernalgorithmen. Studienarbeit Neuronale Identifikation und Regelung am Lehrstuhl für Informatik III, Universität Würzburg, Institut für Informatik. Teil B von Neuronale Echtzeitregelung eines doppeltinversen Pendels.

Müller und Magnus (1987). *Übungen zur technischen Mechanik*. Stuttgart: Teubner Verlag.

Saerens, M. und A. Soquet (1991). Neural controller based on back-propagation algorithm. In *IEE Proceedings-F.*, Volume 138 (1), pp. 55–62. Institution of Electrical Engenieers.

Scherg, J. und W. Jodl (1996). Echtzeit–Rekurrentes Lernen. Projektpraktikum Neuronale Identifikation und Regelung am Lehrstuhl für Informatik III, Universität Würzburg, Institut für Informatik.

Szabo, J. (1964). *Höhere Technische Mechanik*. Berlin: Springer Verlag.

Tutschku, K. (1992). Das Problem der Dimensionierung von Multi-Layer-Perceptrons. Studienarbeit, Universität Würzburg, Institut für Informatik.

Zurmühl, R. (1965). *Praktische Mathematik für Ingenieure und Physiker*. Berlin: Springer Verlag.