INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

DANKSAGUNG

Diese Diplomarbeit entstand am Verein für alternative Energieforschung in Karlsruhe in Zusammenarbeit mit der Fachhochschule Kiel.

Ich bedanke mich an dieser Stelle bei Herrn Prof. Dr. –Ing. Habil Albrecht Zur, der diese Diplomarbeit seitens der Fachhochschule betreuet hat und so viel Geduld hatte. Ferner möchte ich bei Herrn Ing. Murad, Samir für die Ermöglichung dieser Diplomarbeit und zur Verfügungsstellung des Laboratoriums mit seinem gesamten Hardwareinhalt bedanken.

Mein besonderer Dank an dem Deutschen Staat, der mir die Möglichkeit gegeben hat, meine Bildung fortzusetzen und neue Perspektiven zu erwerben.

Zuletzt bedanke ich mich bei meinen Eltern, die mir zur Seite gestanden sind, trotz der Entfernung und bei meinen Freunden, die mich moralisch und tatsächlich unterstützt haben.

1. Einleitung zum Luftschiffprojekt

1.1. Das Lotte-Projekt und die „alternative Lotte“

Am Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktion an der Universität Stuttgart wurde ein Solarluftschiff gebaut. Im Februar 1992 wurde das Projekt „Solarluftboot“ eingeleitet. Der Zweck dieses Projektes war, neue Materialien, neue Konstruktionsmethoden und ein neues Konzept für das Steuern eines Luftschiffs zu entwickeln, das durch eine Solarenergiemaschine betrieben wird.
Die „alternative Lotte“, ein gemeinsames Projekt des VaEf e.V., der Universitäten Karlsruhe und Stuttgart sowie der Fachhochschule Karlsruhe, wurde als ein experimentelles Luftschiff geplant. Es soll direkt vom Boden gesteuert werden (erster Schritt der Implementierung) oder automatisch zu spezifizierten Koordinaten (zweiter Schritt der Implementierung) fliegen. Die Energieversorgung wird im ersten Schritt durch herkömmliche Batterien gewährleistet. Aber es ist geplant, später auf Solarenergie umzuschalten. Mit der „alternativen Lotte“ werden umgebungsspezifische Daten während des Fluges über unterschiedliche Sensoren gesammelt (Sensorik), die am Luftschiff angebracht werden können. Im ersten Schritt werden die Geschwindigkeit des Winds, die Temperatur und die Solarstrahlung gemessen. Zusätzlich zu solchen Flugdaten wie Beschleunigung werden Winkelgeschwindigkeit und Azimutwinkel zu Navigationszwecken gemessen.

1.2. Entwicklungsmethode

Das komplette System des Luftschiffs wurde in einem Labor mit einem Board Computer und einer Sensorik-, einer Aktorik- und einer Transceiverkarte als Embedded System bereitgestellt.
Die Software wird mit der C-Programmiersprache unter Linux entwickelt.
Der vollständige Entwicklungsprozess folgt im Allgemeinen dem bekannten V-Modell.

1.3. Aktivität und Realisierungsannäherung

Die Aufgabe besteht darin, eine Softwareschnittstelle zur Kommunikation zu entwickeln, die auf dem Board Computer läuft, der seinerseits mit den drei Steuerungskarten (Sensorik, Aktorik und Transceiver) am Luftschiff angeschlossen wird. Zweck dieses Programms ist es, die Kommunikation zwischen den Sensoren und den Motoren und dem Transceiver auf der „alternativen Lotte“ Sensor-Plattform herzustellen. Die Sensorik wird benutzt, um die Sonnenstrahlung, die Geschwindigkeit und die Richtung des Luftschiffs zu messen und die Daten zum Board Computer zu schicken. Die Aktorik wird zum Steuern der Richtung und der Beschleunigung (Geschwindigkeit, Position) des Luftschiffes (Motoren) verwendet, indem sie die Daten vom Board Computer abhängig von den empfangenen Daten von der Sensorik erhält. Die Transceiverkarte wird verwendet, um die Daten in der einen Richtung vom Board Computer zum lokalen Computer zu senden und in der anderen Richtung vom lokalen Computer zu empfangen.
So kann die Diplomarbeit in folgende Aufgaben unterteilt werden:

- ✓ Vorbereitung, Konfiguration und Übersetzung des Kerns.
- ✓ Installation von Linux und Echtzeiterweiterung.
- ✓ Programmierung des Kommunikationsprotokolls zwischen den Karten und dem Board Computer mit der C-Programmiersprache.

1.4. Überblick

Nach einer kurzen Einleitung wird in knapper Form Grundlagenwissen über die verwendeten Entwicklungsmethoden gegeben.
Das V-Modell und einige allgemeine Informationen zu (SA/SD) werden in Kapitel 2 vermittelt. Kapitel 3 gibt einen Eindruck dessen, was für Werkzeuge verwendet worden sind. Im 4. Kapitel wird eine ausführliche Beschreibung des vollständigen Systems gegeben. Kapitel 5 beschäftigt sich mit dem Betriebssystem und der Echtzeiterweiterung. Kapitel 6 vermittelt die erforderlichen Informationen über die serielle Schnittstelle sowie die Vorgehensweise bei der Programmierung und die Tests, die durchgeführt worden sind, um die Funktionalität des Systems zu überprüfen.

2. **Grundlagen der Arbeit**

2.1. Das V-Modell

Das V-Modell ist eine umfangreiche Wissensansammlung über das beste Verfahren der Software-Entwicklung. Dieses Wissen kann als Prozess beschrieben werden. Zusätzlich zu den geplanten Produkten und zu den Aktivitäten enthält das V-Modell auch Informationen über den Prozess und den Verlauf der Entwicklung des Projekts. Zu diesem Zweck umfasst der Prozessstandard auch die Ausgabeprodukte, die durch eine Aktivität erzeugt werden sollen, und welche Nachfolgeraktivitäten dieses Produkt als Eingang benötigen. Dieser interne Produktfluss erlaubt es, einen chronologischen Auftrag für die Aktivitäten abzuleiten.
Das V-Modell ist ein logisches Produktmodell. Es beschreibt den Inhalt der Produkte, die während eines Software-Projektes hergestellt werden müssen und ihre Verhältnisse auf einem sehr hohen Niveau. In einem realen Projekt jedoch kann die physikalische Struktur dieser Produkte (und wird in den meisten Fällen) ziemlich unterschiedlich sein.

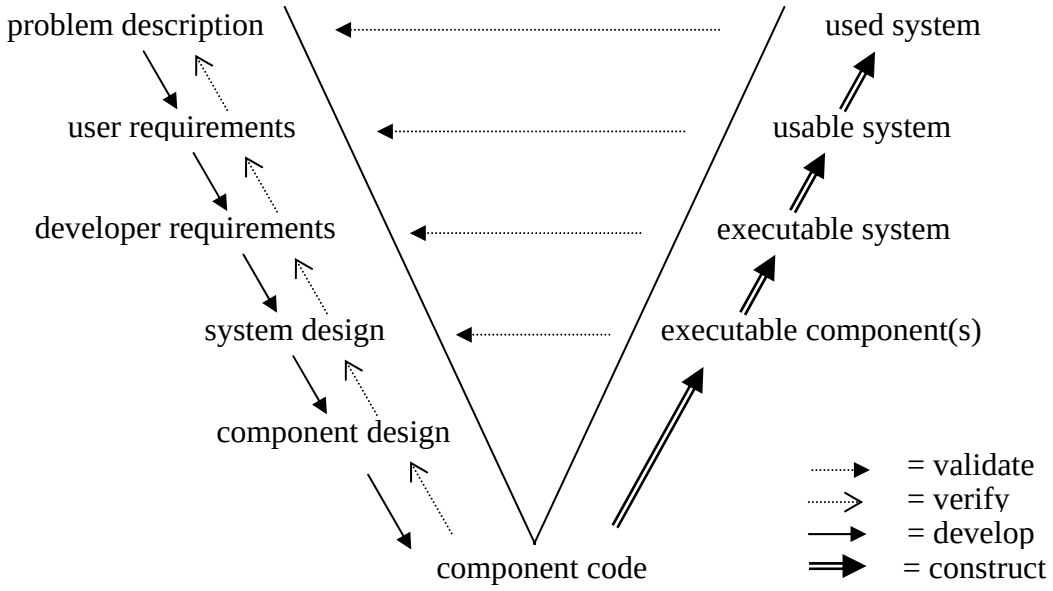


Abbildung 2- 1: V-Modell [1].

Die linke Seite des V zeigt die Produkte, die aus Entwicklungsaktivitäten resultieren: Problembeschreibung, Benutzeranforderungen, Entwickleranforderungen, Systemdesign, Komponentendesign und Komponentencode. Die Produkte sind hier nicht weiter beschrieben.

Die rechte Seite des V dokumentiert den Aufbau des abschließenden Systems beginnend mit einzelnen Bestandteilen. Die Produkte auf dieser Seite des V werden ausführbare Bestandteile, ausführbares System, verwendbares System und benutztes System genannt.

Zusätzlich zu Entwicklung und Aufbauaktivitäten enthält das V-Modell auch analytische Aktivitäten und zwar Überprüfung und Gültigkeitserklärung:

- ✓ Die Überprüfung findet nach jeder Entwicklungsaktivität statt. Der Ausgang der Aktivität wird gegen den Eingang überprüft, um zu testen, ob beide wirklich zusammenpassen.
- ✓ Die Gültigkeitserklärung geschieht nach jedem Aufbauschritt, um sicherzustellen, dass sich das bis dahin konstruierte System so verhält, wie es soll. Er wird folglich gegen das gegenüberliegende Produkt auf der anderen Seite des V-Modells geprüft.

Es sollte hervorgehoben werden, dass das V-Modell weder ein physikalisches Produktmodell noch irgendeine Art Prozess oder Lebenszyklusmodell ist. Jedoch wegen seiner sehr allgemeinen Beschreibung können die in einem Lebenszyklusmodell benutzten Produkte immer mit den Produkten zusammengepasst werden, die im V-Modell enthalten sind. Dem zu Folge kann es als generisches Produktmodell gesehen werden.

Das V-Modell ist ein Lebensdauerprozessmodell, ursprünglich entwickelt, um den Software-Entwicklungsprozess innerhalb der Bundesregierung der Bundesrepublik Deutschland zu regulieren.

Das V-Modell besteht aus vier Submodellen: Softwareentwicklung, Qualitäts-sicherung, Konfigurationsmanagement und Projektmanagement.

Die Submodelle werden nah zusammengeschaltet und gegenseitig durch Austausch der Produkte und der Ergebnisse beeinflusst:

- ✓ Das Software-Entwicklungssubmodell entwickelt das System oder die Software.

- ✓ Das Qualitätssicherungssubmodell reicht Anforderungen bei den anderen Submodellen und bei den Testfällen und -kriterien ein, um die Produkte und die Befolgung von Standards zu sichern.
- ✓ Das Konfigurationsmanagementsubmodell verwaltet die erzeugten Produkte.
- ✓ Das Projektmanagement-Model plant, überwacht und informiert die anderen Submodelle.

Jedes dieser Submodelle kann weiter zerlegt werden. Zum Beispiel kann das Software-Entwicklungssubmodell wie folgt unterteilt werden [2]:

- ✓ Systemanforderungsstudie und Design.
- ✓ Datenverarbeitende Anforderungsstudie und Design.
- ✓ Software-Anforderungsstudie.
- ✓ Einleitendes Design.
- ✓ Ausführliches Design.
- ✓ Implementierung.
- ✓ Software-Integration.
- ✓ Datenverarbeitende Integration.
- ✓ Anpassung.

2.2. Strukturierte Analyse (SA)/Strukturiertes Design (SD)

In diesem Kapitel werden die Strukturierte Analyse (SA) und das Strukturierte Design (DS) der siebziger und achtziger Jahre des 20. Jahrhunderts kurz beschrieben.

SA war eine Technik, in der die Anforderungen des Kunden in eine Hierarchie von Funktionen unterteilt waren. Datensätze wurden in den abstrakten Bezeichnungen spezifiziert, und ihre Handhabungen wurden durch funktionell zerlegte Datenflusspläne graphisch dargestellt. In dieser Form beschrieb **SA,** was das System tun würde, obgleich diese Bezeichnungen sehr technisch ausgedrückt sind. Folglich war ein **SD**, mehr oder weniger, eine Beschreibung dessen, wie ein System strukturiert werden würde, um den Anforderungen zu entsprechen. **SD** beschrieb das Verteilen von der Software in Modulen und den Fluss von Daten unter diesen Modulen.

SA/SD enthielt ein Verfahren, welches als Transformationsanalyse bekannt ist. Dies wurde verwendet, um die Diagramme, die eine strukturierte Analyse **(SA)** darstellten, umzuwandeln, um Diagramme eines Strukturierten Designs **(SD)** wiederzugeben.

Das heißt, dass die Analyse wirklich eine einleitende Beschreibung des Designs war, die nur ein Diagramm erforderte, um vervollständigt zu werden. Die Analyse hat einfach die Datenmengen und deren Umwandlungen beschrieben, ohne darüber hinaus etwas zur Software-Struktur zu erwähnen.

SA/SD deutete auch eine zeitliche Begrenzung zwischen Analyse und Design an, die nicht in der vorherigen Systemanalyse enthalten war [3].

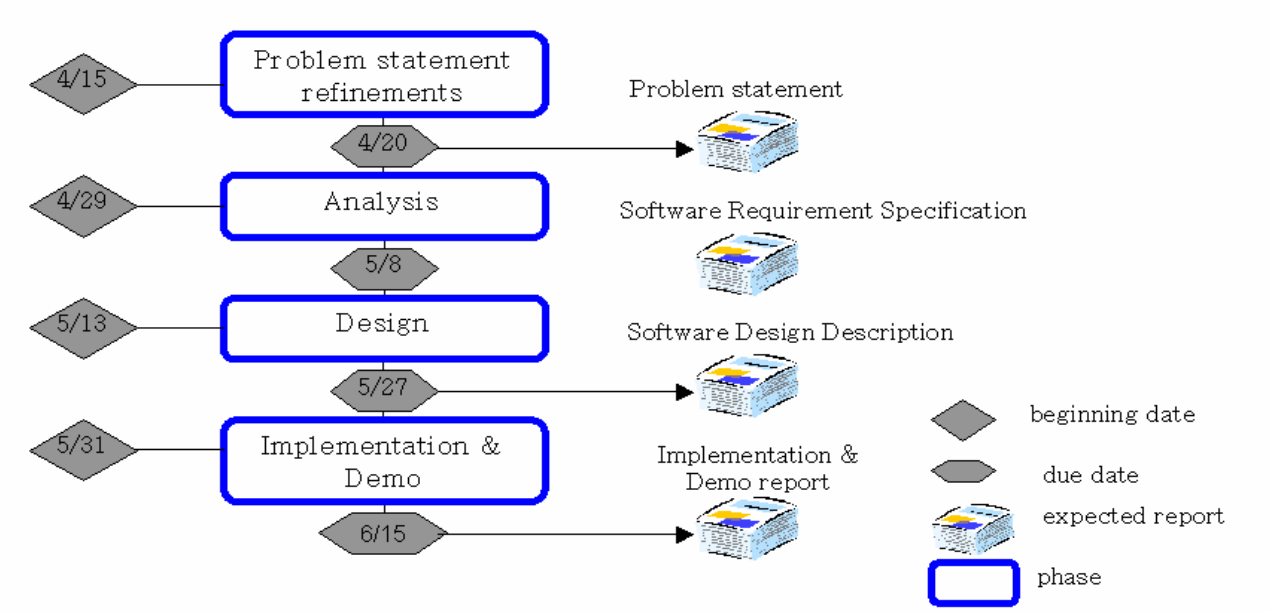


Abbildung 2- 2: SA/SD [4].

3. **Entwicklungsumgebung**

Die Entwicklung ist auf einem Rechner mit GX1-AMD- k6 300MHz Prozessor, 32MB RAM, 16 MB Flash Disk erfolgt, indem die Hardware (Steuerungskarten) untereinander mit dem Board Computer via R232 verbunden wurde.

Die folgenden Software-Werkzeuge sind für die Entwicklung benutzt worden:

- ✓ Suse und RedHat Linux.
- ✓ KDevelop: C/C++ Entwicklungsumgebung unter Linux.
- ✓ „GCC“ und „GDB“ Compiler und Debugger für C-Programme unter Linux.
- ✓ Microsoft Word zu Textverarbeitung und zur Darstellung der Designstruktur.
- ✓ Zwei Rechner jeweils mit einem RS232 für Testzwecke.
- ✓ Nullmodemkabel für die Verbindung der RS232-Schnittstellen.
- ✓ Hyperterminal unter Windows und minicom unter Linux.

4. **Architekturdesign und Implementierung**

Die Aufgabe besteht darin, eine Softwareschnittstelle zur Integration zu entwickeln. Mit Integration ist die Kommunikation zwischen den Steuerungskarten gemeint, also die Sensorik, die Aktorik und der Transceiver. Diese Kommunikation soll in Form eines C-Programms auf dem Board Computer stattfinden, der dann den Datenaustausch zwischen den Steuerungskarten überwacht und steuert. Diese vier Bauteile befinden sich auf der „alternativen Lotte“ und werden durch die serielle Schnittstelle miteinander verbunden.
Jede dieser Steuerungskarten wurde für sich entweder als Diplomarbeit, Studienarbeit oder Ingenieurpraktikum entworfen, entwickelt und getestet – bis auf den Board Computer, der von der Firma Arcom gekauft wurde [5]. Die Karten enthalten jeweils einen Mikrokontroller, der entweder Daten für RS232 zum Senden bereitstellt oder bereit ist, Daten von RS232 zu empfangen.

Die Sensorik sammelt Daten und schickt sie an den Board Computer weiter. Es werden die Sonnenstrahlstärke, Geschwindigkeit und Richtung des Luftschiffes ausgemessen (Sonnenstrahlung in Abhängigkeit der Koordinaten und der Jahreszeit), indem z.B. Sensoren für jede Koordinate eingesetzt werden: x-Richtung, y-Richtung und z-Richtung. (Die Temperatur, Winkelgeschwindigkeit und Azimutwinkel sollen auch ermittelt werden).
Die Aktorik dient der Steuerung der „alternativen Lotte“, also Richtung, Geschwindigkeit und Winkel anzugeben, abhängig von den Informationen vom Board Computer, der dann diese Daten entweder von der Sensorik oder aber vom Transceiver empfängt.
Die Transceiverkarte ihrerseits soll den Datenaustausch zwischen der „alternativen Lotte“ und der Bodenstation gewährleisten und zwar kabellos, indem ein Modem mit Antenne zum Einsatz kommt. Dieses wurde bereits erfolgreich getestet. Da die Transceiverkarte über eine graphische Benutzerschnittstelle verfügt, ist es möglich, Befehle manuell von dem Boden einzugeben und Korrekturen durchzuführen.

Die vollständigen Systemanschlüsse zwischen den Steuerungskarten mit dem Luftschiff und der Bodenstation (Benutzerschnittstellencomputer) werden in der folgenden Abbildung gezeigt:

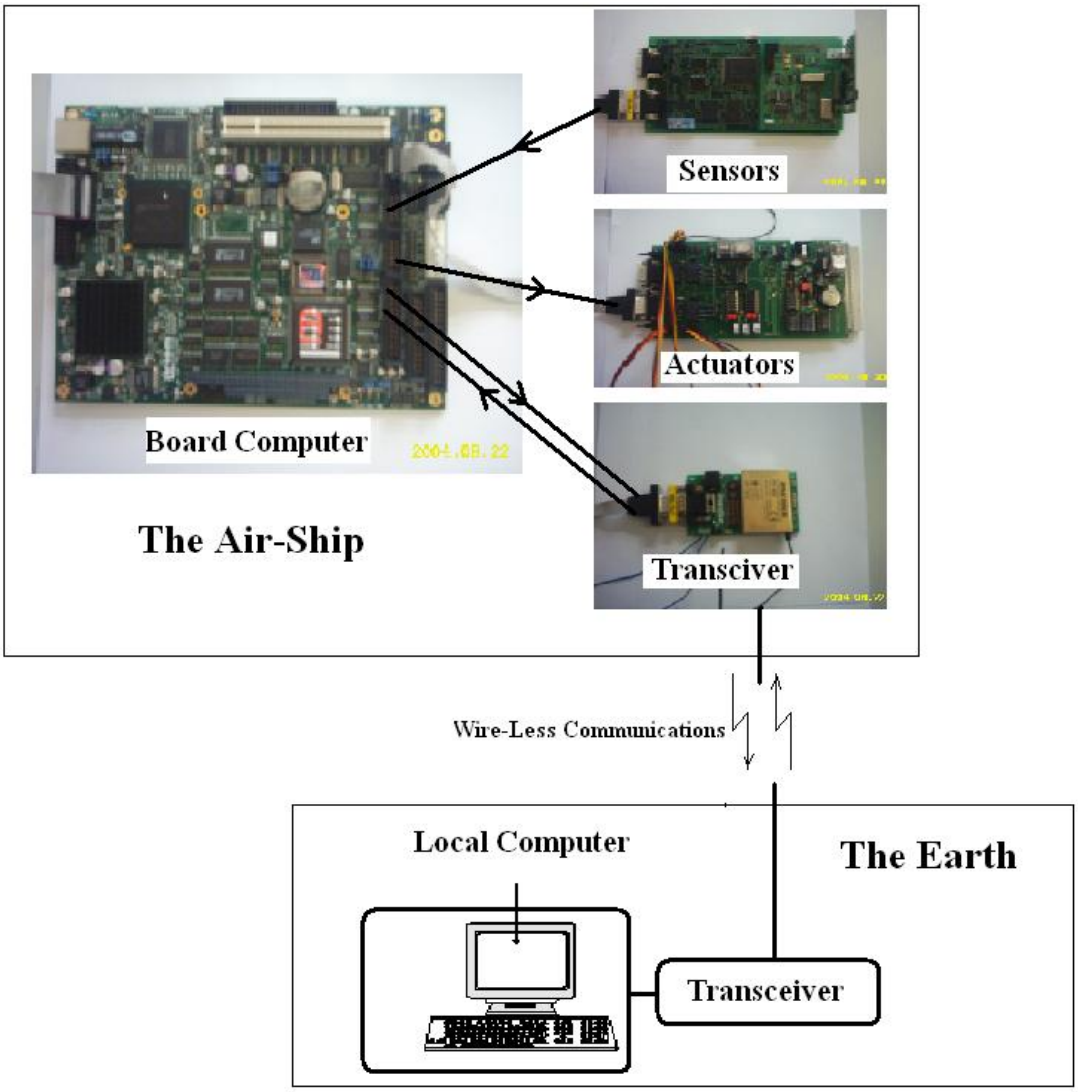


Abbildung 4- 1: Verbindungen des Systems.

Die Abbildung beschreibt sowohl die Anschlüsse zwischen dem Board Computer und den drei Steuerungskarten (Sensorik, Aktorik und Transceiver) in der „alternativen Lotte“ als auch den Anschluss mit dem Rechner in der Bodenstation.
Ebenso ist der Abbildung zu entnehmen, dass das Gesamtsystem in zwei Teile gegliedert ist:

1. Die Bodenstation: mit einem Computer, der das Luftschiff vom Boden überwacht und steuern kann. Für diese Aufgabe wird eine graphische Benutzerschnittstelle verwendet. Diese GUI greift hardwaremäßig auf die

Transceiverkarte zurück. Der Transceiver seinerseits basiert auf einem Modem, das mit einem anderen Modem auf dem Luftschiff kabellos (durch Antennen) Daten in beiden Richtungen austauschen kann bzw. soll. Durch diese graphische Benutzeroberfläche soll es möglich sein, Windgeschwindigkeit, Sonnenstrahlung, Temperatur, Luftschiffgeschwindigkeit bzw. Luftschiffkoordinaten zu empfangen und gleichzeitig in die andere Richtung Windgeschwindigkeit, Beschleunigung und Azimutwinkel zu versenden.

2. Das Luftschiff: Hier liegt der eigentliche Aufgabenbereich der vorliegenden Arbeit, die Integration der drei Steuerungskarten im einem Embedded Computer. Dieser wird durch Nullmodemkabel jeweils mit den Steuerungskarten verbunden. Der Embedded Computer enthält vier serielle Schnittstellen. Die Steuerungskarten selber sind mindestens mit einer seriellen Schnittstelle ausgestattet. Im nächsten Abschnitt werden die einzelnen Karten ausführlicher besprochen bzw. vorgestellt.

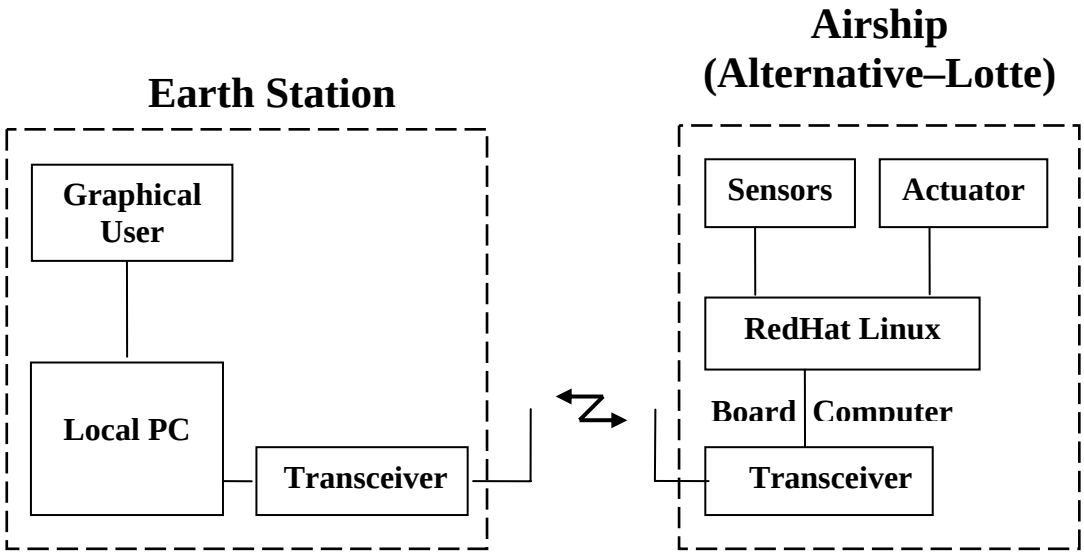


Abbildung 4- 2: Gesamtsystem der „alternativen Lotte“.

4.1. Die Implementierung des Basisstationsprogramms

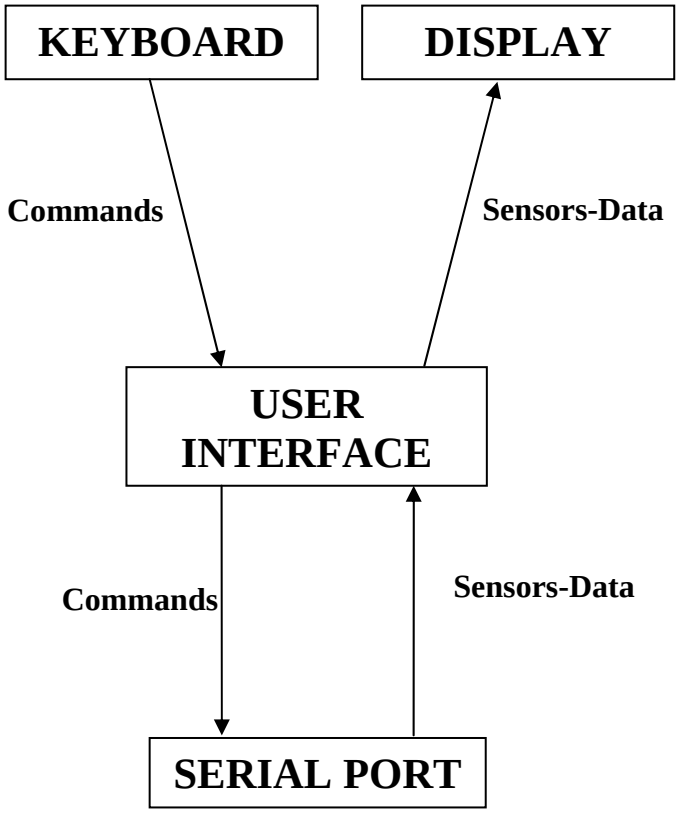


Abbildung 4- 3: Diagramm der Zusammenhänge.

- ✓ Das DISPLAY ist der Bildschirm eines Rechners in der Bodenstation. Es wird benutzt, um die empfangenen Sensor-Daten zu beobachten. Diese Sensor-Daten werden von der Sensorik in der „alternativen Lotte“ erfasst.
- ✓ Das KEYBOARD wird vom Benutzer verwendet, um die Befehle zu schreiben, die er zur „alternativen Lotte“ schicken möchte. Diese Befehle werden zur „alternativen Lotte“ durch die serielle Schnittstelle geschickt. Wann immer der Benutzer Befehle zur „alternativen Lotte“ schicken möchte, werden diese Daten dem Board Computer vom Transceiver übergeben.
- ✓ Der SERIAL PORT ist die Peripherie zwischen dem USER INTERFACE und der „alternativen Lotte“. Indem wir die Benutzerschnittstelle verwenden, können wir sagen, dass der Benutzer Daten erhält (von der „alternativen Lotte“) und Daten oder Befehle (zur „alternativen Lotte“) schickt.
- ✓ Die Sicherheitsschicht wird dem Kommunikationsprotokoll hinzugefügt. Die Befehle, die die Bodenstation zur „alternativen Lotte“ schickt, werden bestätigt. Wenn die Bestätigung ausfällt, werden die Befehle zurückgesendet.

4.2. Die Implementierung des Luftschiffprogramms

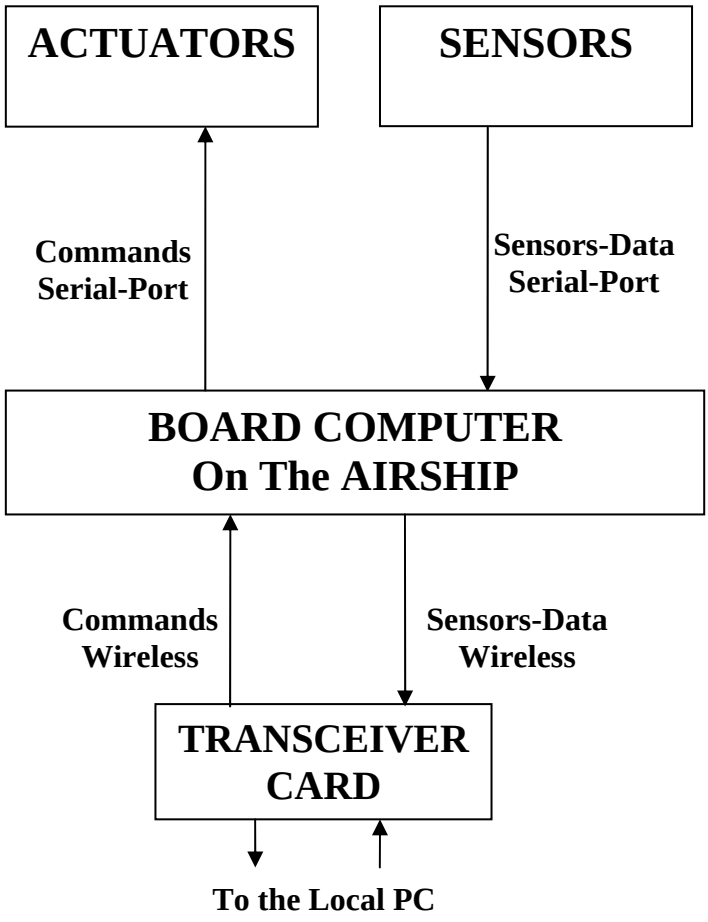


Abbildung 4- 4: Luftschiff als Embedded System.

4.2.1. Die Sensorik

Diese Karte stellt eines der wichtigsten Bestandteile der „alternativen Lotte“ dar. Die Sensorik überwacht das Gesamtsystem, dadurch dass Sensoren an dieser Karte angebracht sind. Diese erfassen die Beschleunigung, die Geschwindigkeit, die Koordinaten (die Position des Luftschiffes), die Winkelgeschwindigkeit, den Azimutwinkel und die Richtung, die dann dem Board Computer über die serielle Schnittstelle übertragen werden. Dieser leitet seinerseits die Daten bzw. Informationen kabellos an die Bodenstation weiter. Außerdem liefert uns die Sensorik andere Daten, wie die Sonnenstrahlung (Solarstrahlung und Windenergie) und die Temperatur.

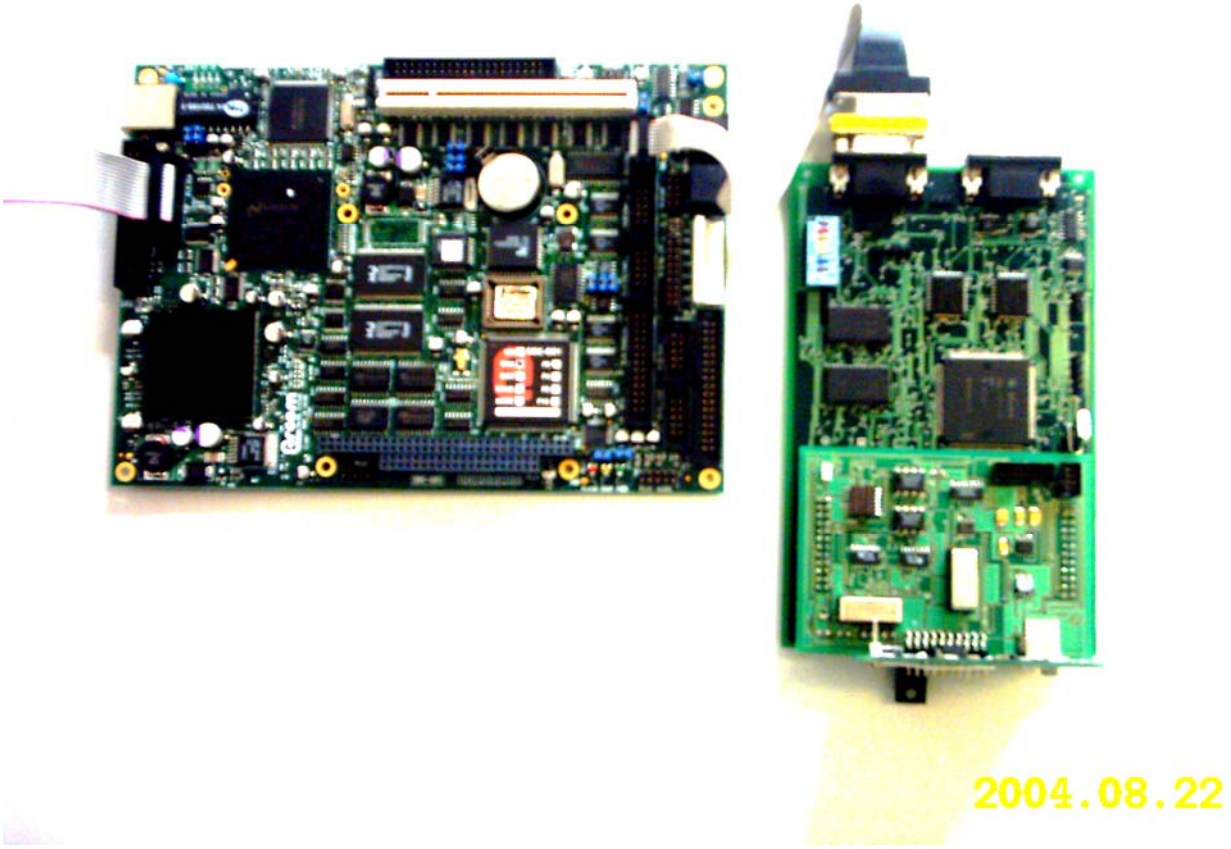


Abbildung 4- 5: : Die Sensorik durch die serielle Schnittstelle mit dem Board Computer angeschlossen.

4.2.2. Die Aktorik

Die Sensorik ermöglicht es, Fluginformationen zu sammeln. Um der Veränderlichkeit der Daten tragen zu können, muss die Aktorik eingesetzt werden. D.h. diese Steuerungskarte enthält z.B. einen Schrittmotor und einen Servomotor, um einerseits die Geschwindigkeit des Luftschiffes und andererseits die Richtung zu steuern. Die Bewegungs- und Flügelbefehle werden vom Board Computer empfangen. Diese Befehle werden entweder von der Sensorik oder von der Bodenstation dem Board Computer und dann der Aktorik zur Verfügung gestellt.

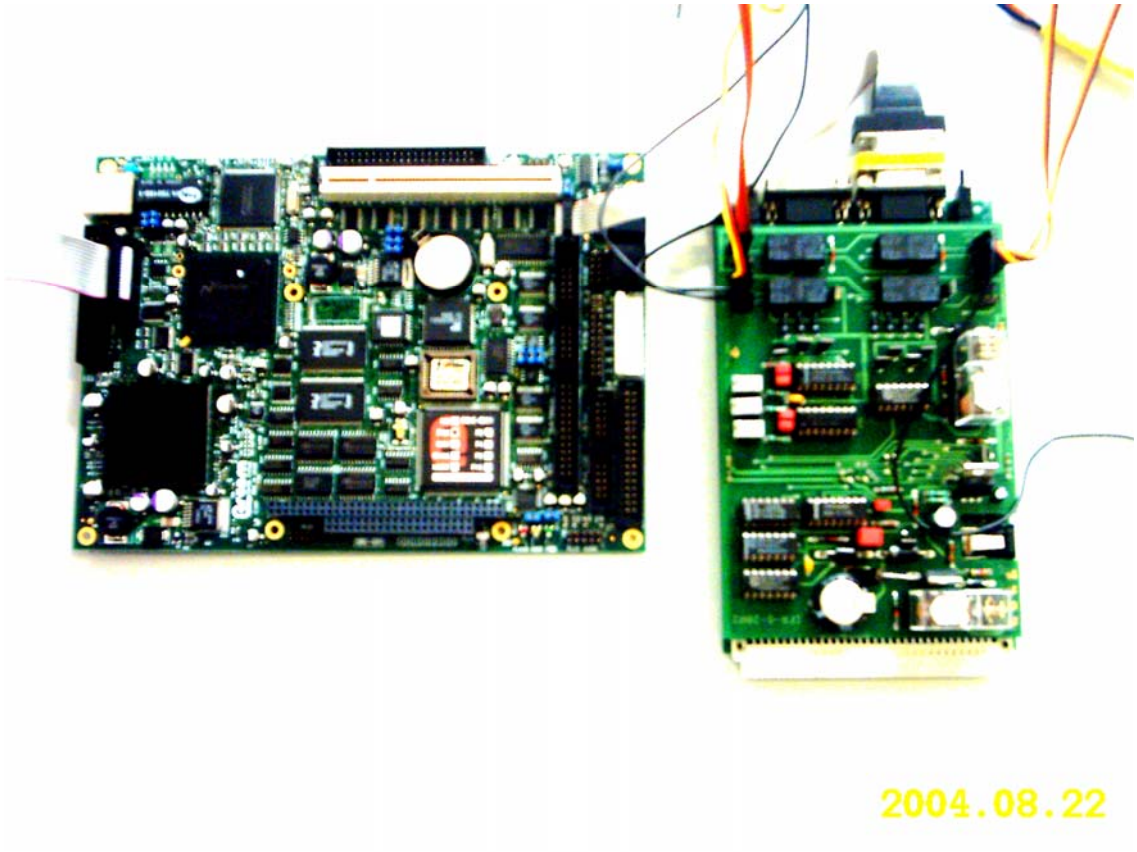


Abbildung 4- 6: Die Aktorik durch die serielle Schnittstelle mit dem Board Computer angeschlossen.

4.2.3. Der Transceiver

Diese Karte besteht sowohl aus Hardware als auch aus Software. Die Hardware stellt sich in Form eines Modems dar, das die kabellose Kommunikation zwischen dem lokalen Computer auf der Bodenstation und dem Board Computer auf der „alternativen Lotte“ gewährleistet. Diese Kommunikation bedeutet Datenaustausch zwischen der Bodenstation und dem Luftschiff, indem eine graphische Benutzerschnittstelle (die Software s. Abb.?) zur Verfügung gestellt ist, um z.B. die Geschwindigkeit einzugeben. Diese wird durch den Board Computer an die Aktorik weitergeleitet und entsprechend die Geschwindigkeit erhöht oder gesenkt, je nachdem welcher Wert eingegeben wurde bzw. was sinnvoller ist.

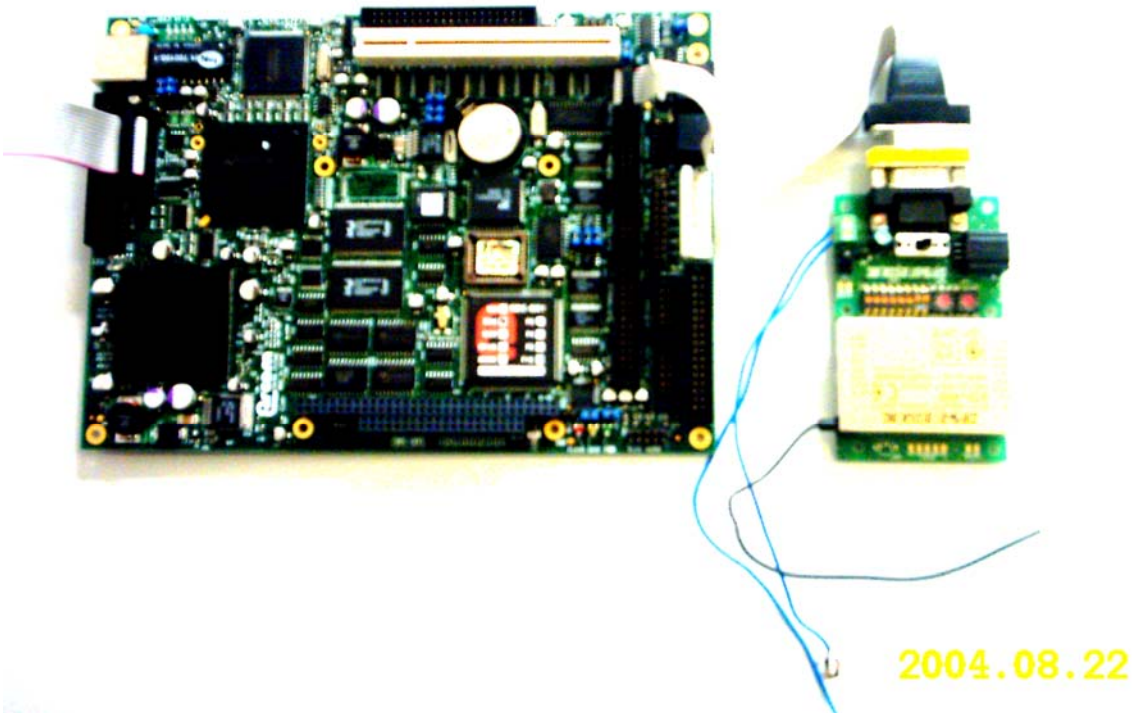


Abbildung 4- 7: Der Transceiver durch die serielle Schnittstelle mit dem Board Computer angeschlossen.

Wenn der Transceiver Sensor-Daten vom Embedded System empfängt, werden diese Daten (in dem Receive Data Frame angezeigt) zur Bodenstation mit einer Frequenz zwischen 433,200 und 434,775 MHZ geschickt. Das Gleiche passiert, wenn der Transceiver Daten von der Bodenstation an die „alternative Lotte“ und dann zur Aktorik senden möchte (in der anderen Richtung sendet man Befehle, indem im Sending Data Frame geschrieben wird).

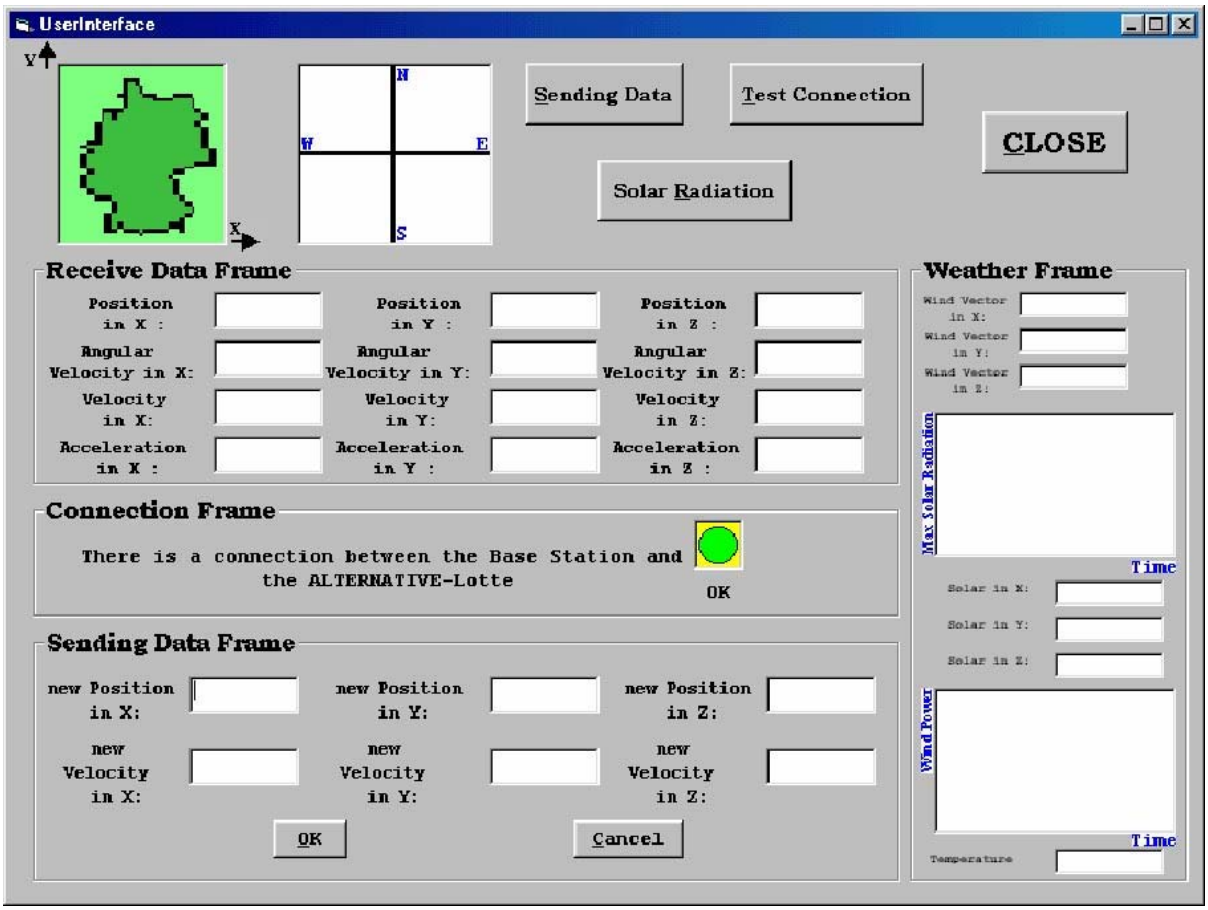


Abbildung 4- 8: Graphische Benutzerschnittstelle.

4.2.4. Der Board Computer

Der Board Computer ist der Embedded Computer in der „alternativen Lotte“. Dieser Computer schließt die drei Steuerungskarten über die seriellen Schnittstellen (RS232) miteinander an.

Spezifikation des Embedded Computers [5]: EBX AMD Geode® GX1 Embedded Computer
Das SBC-GX1 ist ein niedriges Profil, ohne Ventilator, das EBX Format Brett, basiert auf dem 300MHz MMX-erhöhten AMD Geode GX1 Prozessor. Es umfasst alle Rechnerstandard-Schnittstellen sowie eine volle Strecke der Multimediaeigenschaften: VGA-Bildschirm (nationales XpressGraphics), Ethernet 10/100baseTx (PCI 2,1 kompatibel), integriertes 16Mbyte Flash, Doppel-USB, kompatible Schnittstelle des Sounds und vier serielle Schnittstellen.

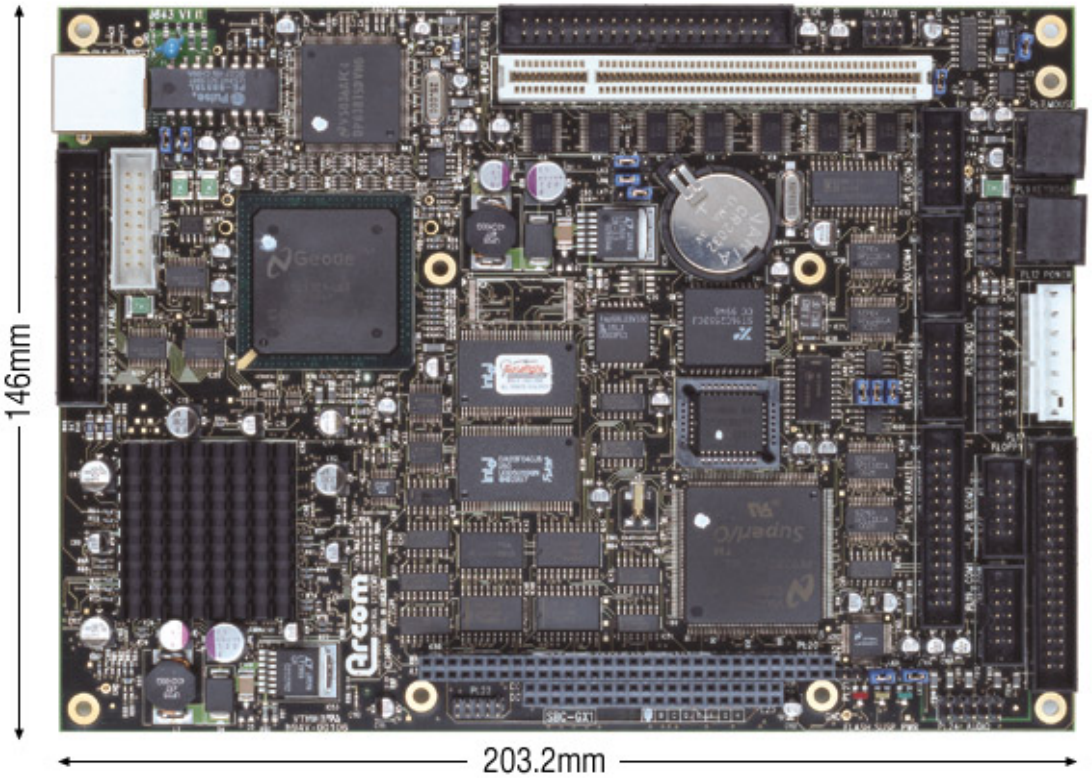


Abbildung 4- 9: Der Board Computer.

Spezifikation	
Prozessor	Ohne Ventilator, Pentium Klasse, 300MHz AMD Geode™ GX1
Speicher	SDRAM: 144-pin low profile SODIMM socket, 256Mbytes max. Flash: 16Mbytes Intel Strata Flash SRAM: 128K battery backed (factory fit option)
Cache	16k L1 write-back cache
Video	TFT Flat Panel & CRT XVGA, 1 - 4MB SDRAM Video Memory
Laufwerkunterstützung	FDD, HDD, Silicon Disk in Flash, Compact Flash
Netzwerkunterstützung	10/100baseTx Ethernet via RJ-45
USB Schnittstellen	2 x USB Ports (USB 1.1)
I/O Schnittstellen	4 x 16550 schnelle serielle Ports (3 x RS-232, 1 x RS-232/422/485)
PC peripheres	Keyboard, Mouse, Druckerport
Zusätzliche I/O	8 general purpose digital I/O signals and two user defined jumpers
Watchdog timer	Echte Hardware watchdog (2 oder 8 Sekunden) mit Reset-Ausgabe.
Erweiterung	Compact Flash socket, PC/104 bus, PCI slot
Sound	16-bit SoundBlaster/Pro compatible interface
BIOS	Award BIOS (gemäß Millennium)
MTBF	90,000 hrs
Format	EBX Format, Größe 5.75“ x 8.00“ (146 x 203 mm)
Leistungsanforderung	+5V nur bei Betrieb (@ 1.5A Typ)
Temperaturbereich	Von -20 bis 60°C Prozessor eingespeist mit niedrigem Profil (ohne Ventilator) Temperaturabfall.

5. Betriebssysteminstallation

5.1. Problemstellung

Der Board Computer soll die Kommunikation zwischen Steuerungskarten verwalten. Er stammt von der Firma Arcom [5]. Auf ihm ist kein Betriebssystem installiert. Da nur ein 16Mb Flash Disk vorhanden ist (Embedded Computer), erscheint es sinnvoll, Linux als Embedded Betriebssystem einzusetzen. Linux bietet den Vorteil, dass über den Kern (der Kern des Betriebssystems) die Wunschkonfiguration erstellt werden kann. Dies spart einerseits Platz, andererseits kann das System bzw. die Hardware optimal eingerichtet werden.

5.2. Durchführung der Installation

Am Anfang der Konfiguration wurden die meisten Optionen deaktiviert, da nur ein möglichst kleiner Kern bzw. ein allgemeines Betriebssystem gebraucht wurde. Folgende Optionen wurden deaktiviert: Parallel support, Plug and Play configuration, Enterprise Volume Management System, Multi-device support (RAID and LVM), Cryptography support (CryptoAPI), Networking options, Telephony Support, ATA/IDE/MFM/RLL support, SCSI support, Fusion MPT device support, IEEE 1394 (FireWire) support (EXPERIMENTAL), I2O device support, Network device support, Amateur Radio support, IrDA (infrared) support, ISDN subsystem, Input core support, Multimedia devices, Sound, USB support, Bluetooth support, Kernel hacking, Library routines, Plug In CPU Schedulers, Support Resource Management Modules und Build options.
Insgesamt wurden 33 Optionen deaktiviert. Sie gliederten sich teilweise in mehrere Untermenüs, teilweise auch nur in ein einziges Untermenü. Die Mehrzahl an Optionen enthielt zwei bis drei Untermenüs. Die Bilder zeigen einige Screenshots, die einen Eindruck von der Zahl der Optionen vermitteln soll.
Obwohl der auf diese Weise konfigurierte Kern klein ist, ließ er sich jedoch erst nach einigem Aufwand kompilieren.
Auf Grund der großen Anzahl an Untermenüs mussten die Optionen in Ruhe angesehen werden, damit sie genauer verstanden werden können.

Die Konfigurationsschritte sind im Internet überall erhältlich, allerdings ist die Konfiguration vom Kern nicht vergleichbar mit der Installation von Windows. Selbst Leute, die Erfahrung mit der Installation von Linux haben, hätten mit Sicherheit Schwierigkeiten mit der Kernkonfiguration.
Folgende Bespiele sollen Embedded Betriebssysteme veranschaulichen. So sind Embedded Betriebssysteme z.B. in PDAs, Organizer und Handys vorhanden, unter anderem WindowsCE oder aber auch Embedded Linux. Auf dieser Hardware ist z.B. keine Festplatte vorhanden, sondern nur ein Flash Disk, auf den das Betriebssystem aufgeladen wurde.
Die Installation des Kerns ist eigentlich eine Reihenfolge von sechs Kommandos. Dabei ist der erste Befehl derjenige, der am meisten Verarbeitungszeit bzw. Konfiguration in Anspruch nimmt. Die anderen benötigen dann je nach Rechnerleistung und Arbeitsspeicher bis zu einigen Tagen Verarbeitungszeit.
Diese Befehle sind wie folgt:

- make menuconfig
- make dep
- make bzImage
- make modules
- make modules_install
- den Bootloader konfigurieren.

5.2.1. Konfiguration des Kerns

Nur der Administrator (root) bzw. ein Benutzer, dem diese Rechte erteilt wurden, hat die entsprechenden Zugriffsrechte, um diese Konfiguration durchzuführen. Das geschieht mit dem Befehl sudo „Kommando“. Vorher muss der Administrator dem Benutzer alle Rechte in der Datei /etc/sudoers zuweisen.

Falls die Kern-Quelldateien noch nicht installiert sind, sollte dies als erstes geschehen, da sonst die Konfiguration nicht begonnen werden kann. Die vorliegenden Quelldateien waren 208,19 MB groß, wobei auch die Hilfedateien enthalten sind und zwar im Verzeichnis /usr/src/linux-2.4.21.99/Documentation. Dort

befindet sich eine große Hilfedatei mit 1,2MB für die Kernkonfiguration und jeweils kleine Hilfedateien zu fast jeder Option.

Um nun im richtigen Verzeichnis (/usr/src/linux-2.4.21.99 im vorliegenden Fall) einen der drei möglichen Befehle einzugeben, stehen weit mehr als 1000 Optionen zur Auswahl. Mit diesen Optionen kann man beeinflussen, welche Funktionen direkt in den Kern integriert werden, welche als Modul und welche gar nicht zur Verfügung stehen sollen:

- make config: Ein paar mit y(ja)/n(nein) bzw. M(Module) zu beantwortende Fragen nacheinander bearbeiten (s. Abbild. ? und ?), ohne dabei die Möglichkeit zu haben, rückgängig zu machen oder umzuändern (Textbasiert), en paar Fragen hängen mit vorher beantworteten Fragen zusammen!

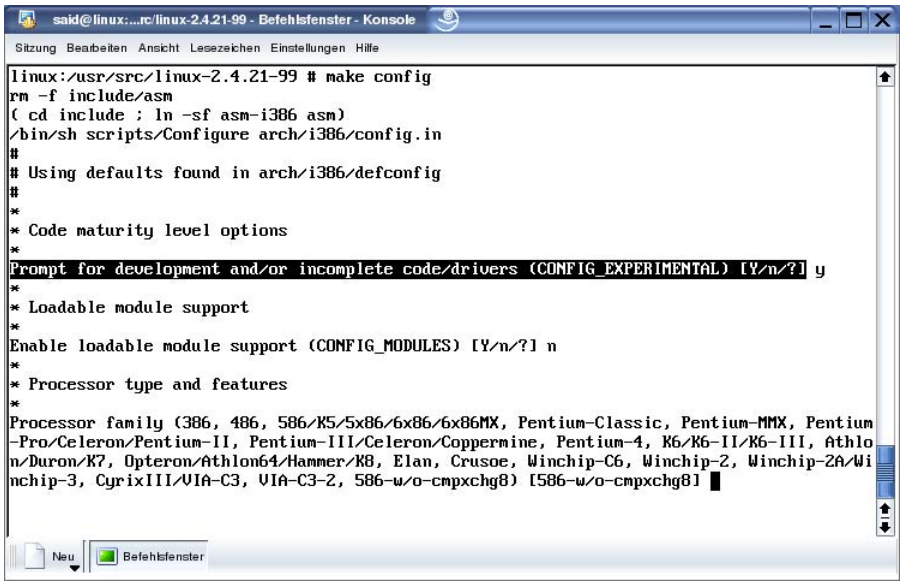


Abbildung 5- 1: make config(1. Option: Code maturity level).

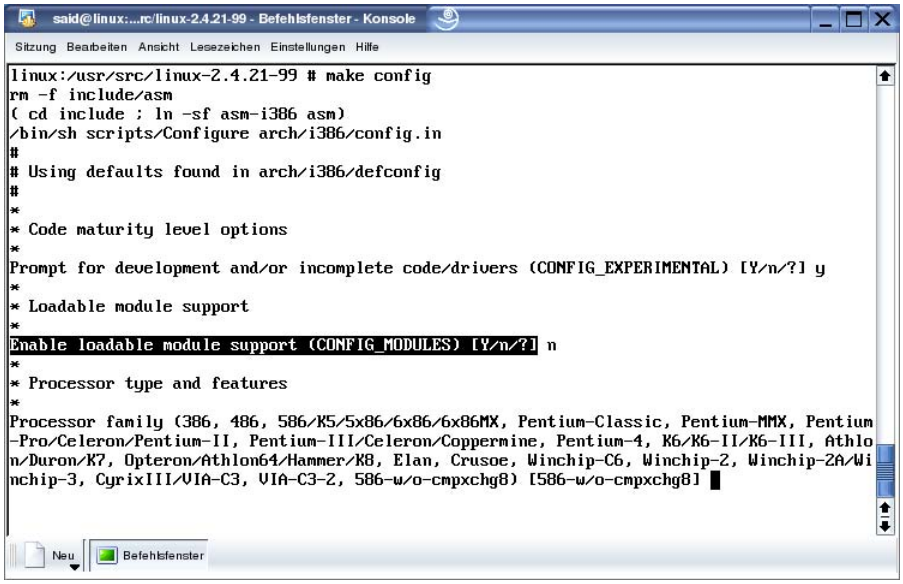


Abbildung 5- 2: make config(2. Option: Loadable module support).

- make menuconfig: Ist auch textbasiert. Dabei besteht aber die Möglichkeit, vor- und rückwärts zu springen und zu korrigieren, d.h. die anfangs vorgenommenen Änderungen rückgängig zu machen oder sogar neu zu bearbeiten (s. Abbild. 5.3 und 5.4).

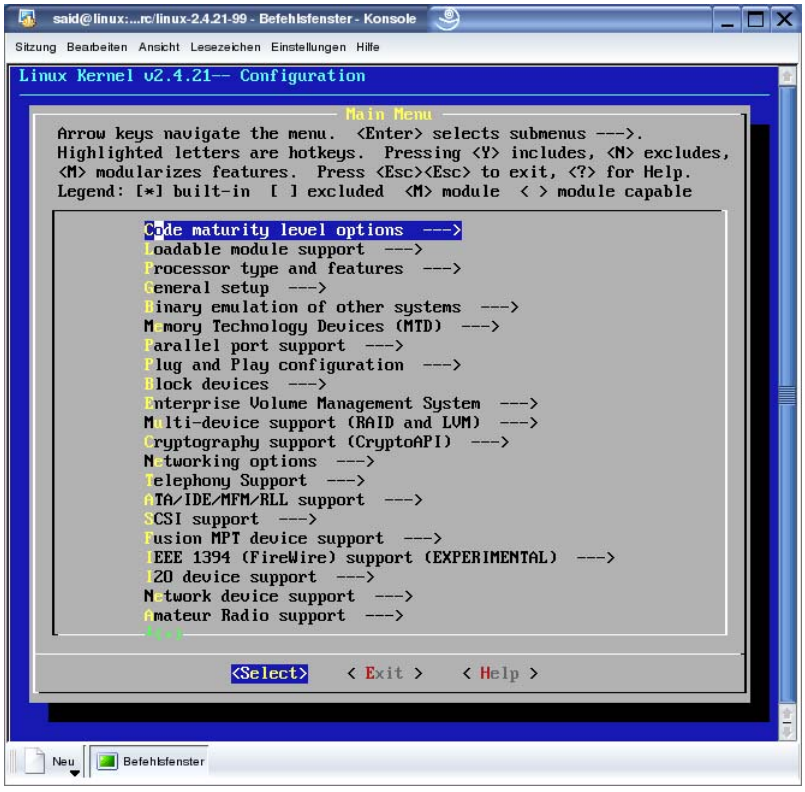


Abbildung 5- 3: make menuconfig(1. Hälfte der Optionen).

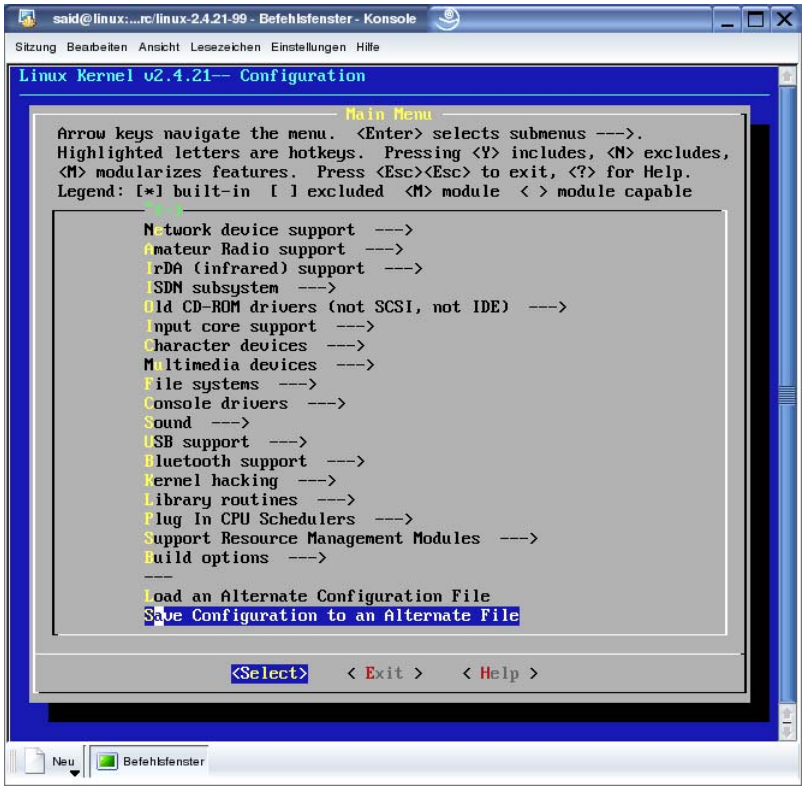


Abbildung 5- 4: make menuconfig(2. Hälfte der Optionen).

- make xconfig: Ist nicht mehr textorientiert ist, sondern einfach mit der Maus bedienbar (s. Abbild. 5.5), (xconfig nur dann, wenn die Programmiersprache Tcl/Tk installiert ist). Dafür braucht man aber die X11-Oberfläche, d.h. die erforderlichen Bibliotheken und die Unterstützung für das Graphical User Interface bzw. KDE, deswegen wird diese Alternative meistens nur am Anfang zu Test- bzw. Lernzwecke verwendet, weil der Kern einerseits nur deswegen kompiliert wird, um einen minimalen Bedarf an Speicherplatz zu gewährleisten. Aus diesem Grund ist dies keine zu empfehlende Alternative, es sei denn, man will den Kern kompilieren, um seiner Linux-Distribution (Suse, RedHat, …) an seine Spezialanforderungen von Hardware und Sicherheit anzupassen bzw. zu optimieren. Ohne genügende Grundkenntnisse kann ein Spielen an der Konfiguration des Kerns die vorhandene Installation oder sogar das ganze System funktionsuntüchtig machen.

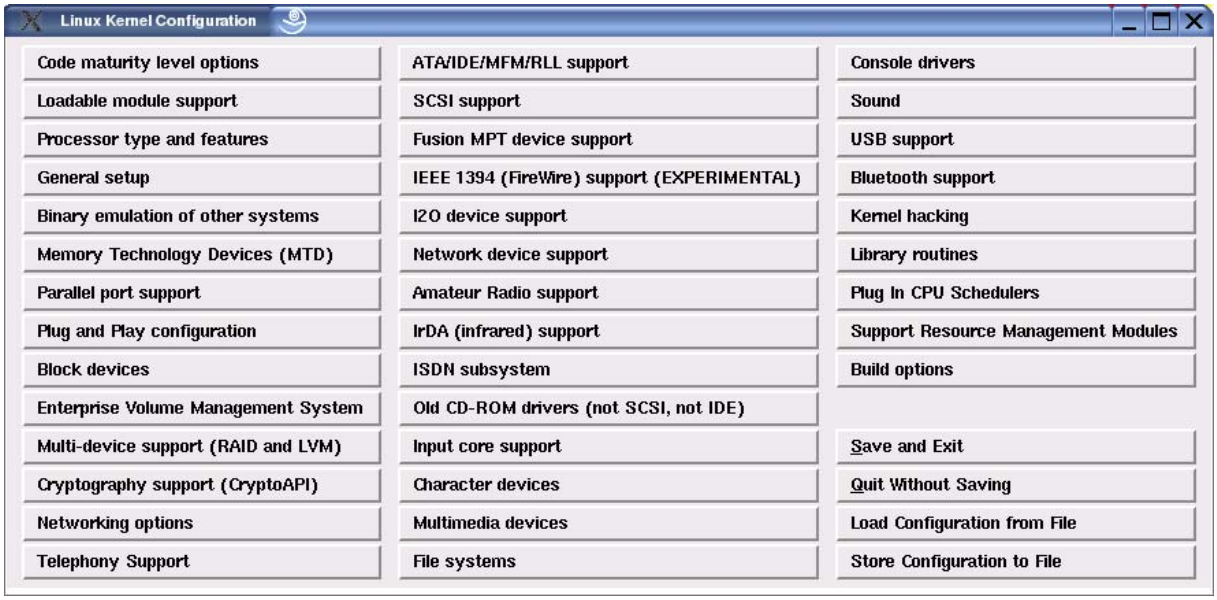


Abbildung 5- 5: make xconfig(Alles auf Einmal).

Nachdem menuconfig gestartet wurde, stehen Optionen zur Verfügung, die mit „y“ für Ja oder mit „n“ für Nein zu beantworten sind. Teilweise kann man mit „M“ für Modul antworten und teilweise müssen Werte angegeben werden.

Als erstes sollte man die aktuelle Konfiguration unter einem anderen Namen vorsichtshalber abspeichern (Letzte Option: Save Configuration to an alternative file), da es vorkommen kann, dass die Änderungen in der vorkonfigurierten Datei gespeichert werden. Dies kann zu Problemen führen, wenn die Datei sich z.B. wegen Unvollständigkeit nicht kompilieren lässt, d.h. es kommt zu Problemen des Kernladens beim Neustart des Computers.

Was sind Module und deren Vorteile?

Ein Modul kann, soweit notwendig, in den Kern eingebunden werden. Diese Module stehen jedoch erst dann zur Verfügung, wenn der Kern geladen ist. Module können zum Kern geladen bzw. davon entladen werden. Module versuchen beim Start ihre Hardware zur finden (Autoprobing).

Beim Hochfahren des Rechners wird ein Basiskern geladen, der nur jene Funktionen enthält, die zum Start erforderlich sind. Diese gehören fest zum Kern. Wenn im laufenden Betrieb Zusatzfunktionen benötigt werden (z.B. für neue Hardware), wird der erforderliche Code als Modul mit dem Kern verbunden. Wenn diese

Zusatzfunktionen eine Weile nicht mehr benötigt werden, kann das Modul wieder aus dem Kern entfernt werden. Dieses modularisierte Konzept hat viele Vorteile:

- Kern-Module können nach Bedarf eingebunden werden. Wenn ein bestimmtes Modul nur selten benötigt wird, kann auf diese Speicher gespart werden, d.h. der Kern ist nicht größer als unbedingt notwendig und optimal an die Hardware des Nutzers angepasst.
- Bei einer Änderung der Hardware (z.B. einer neuen Netzwerkkarte) muss kein neuer Kern kompiliert werden, sondern nur das neue Modul eingebunden werden.
- Bei der Entwicklung eines Kern-Moduls muss nicht ständig der Rechner neu gestartet werden, um die vorgenommenen Änderungen des Kerns zu testen. Es reicht, das Modul neu zu kompilieren. Anschließend kann es bei laufendem Betrieb getestet werden.

Unter den Optionen, findet man z.B. physikalische Ressourcen wie Laufwerke, serielle/parallele Ports, Audio- und Videogeräte, Netzwerkschnittstellen, Tastatur, Maus, etc…

1. **Code maturity level options**: Hier hat man die Möglichkeit, auch die Kern-Komponenten, die als noch nicht ausgereift gelten, bei der Kern-Konfiguration zu berücksichtigen. Im anderen Fall sind diese Optionen nicht aktiv oder werden gar nicht angezeigt. Diese Option wurde gewählt, als Modul lässt sie sich nicht wählen.

2. **Loadable module support**: Hier legt man fest, ob dieses modularisierte Konzept unterstützt werden darf, ob einige Optionen als ladbare Module zur Verfügung stehen dürfen oder nur mit ja/nein dem Kern festgebunden werden sollen.

3. **Processor type and features** (s. Abbild. 5.6): Hier hat man den Prozessortyp anzugeben, was die Geschwindigkeit und Größe des Codes beeinflusst. Der Code ist in diesem Fall auf-, aber nicht – wie gewohnt – abwärtskompatibel (s. Abbild. 5.7), z.B. ein K6-Prozessor kann einen K7-Prozessor unterstützen

aber nicht umgekehrt. Hier wurde ein Athlon/Duron/K7-Prozessor gewählt. Math emulation und Symetric-multiprocessing support wurden nicht gewählt.

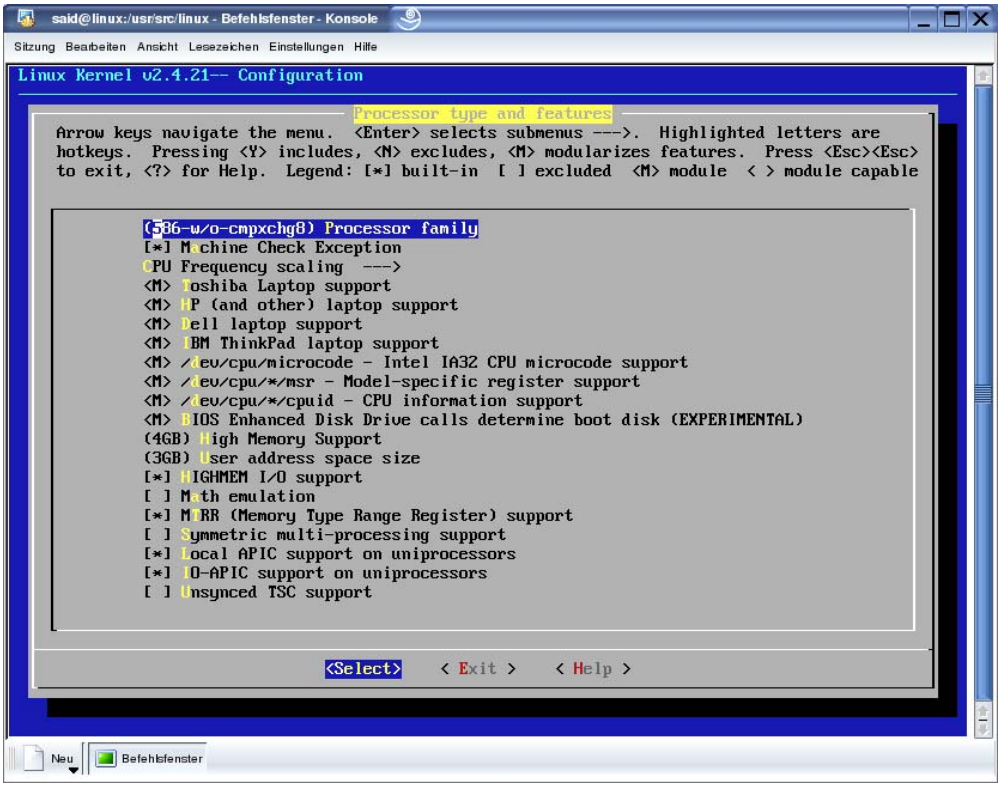


Abbildung 5- 6: Option Processor type and features.

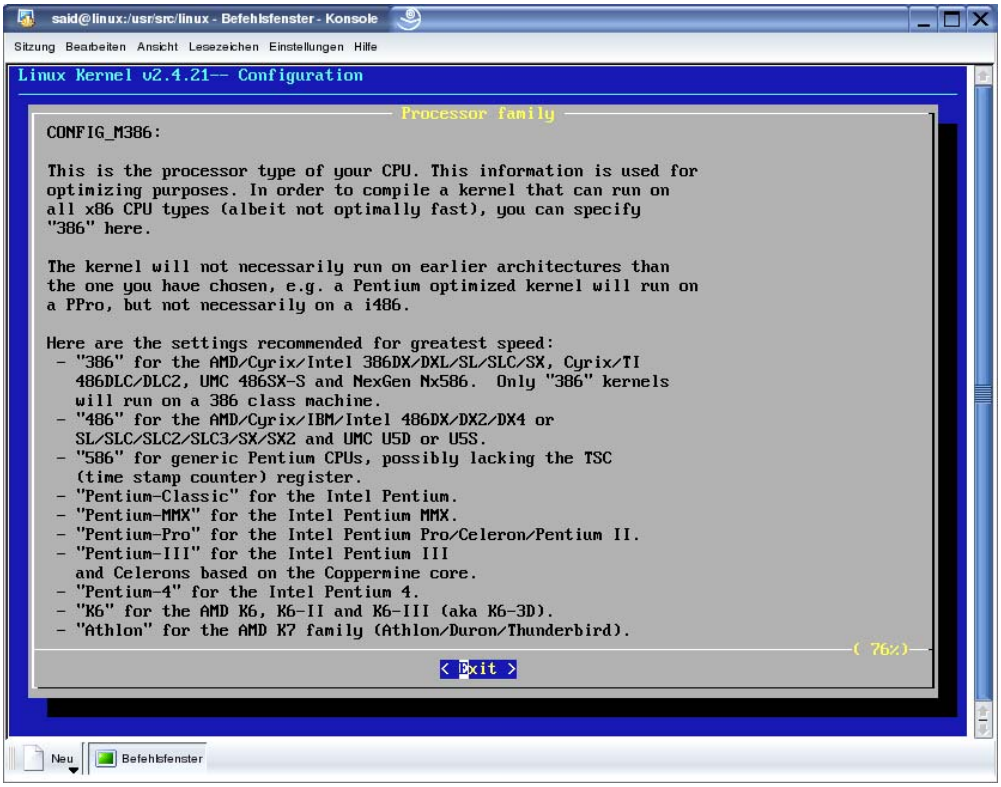


Abbildung 5- 7: Aufruf der Hilfe des Untermenüs Processor family.

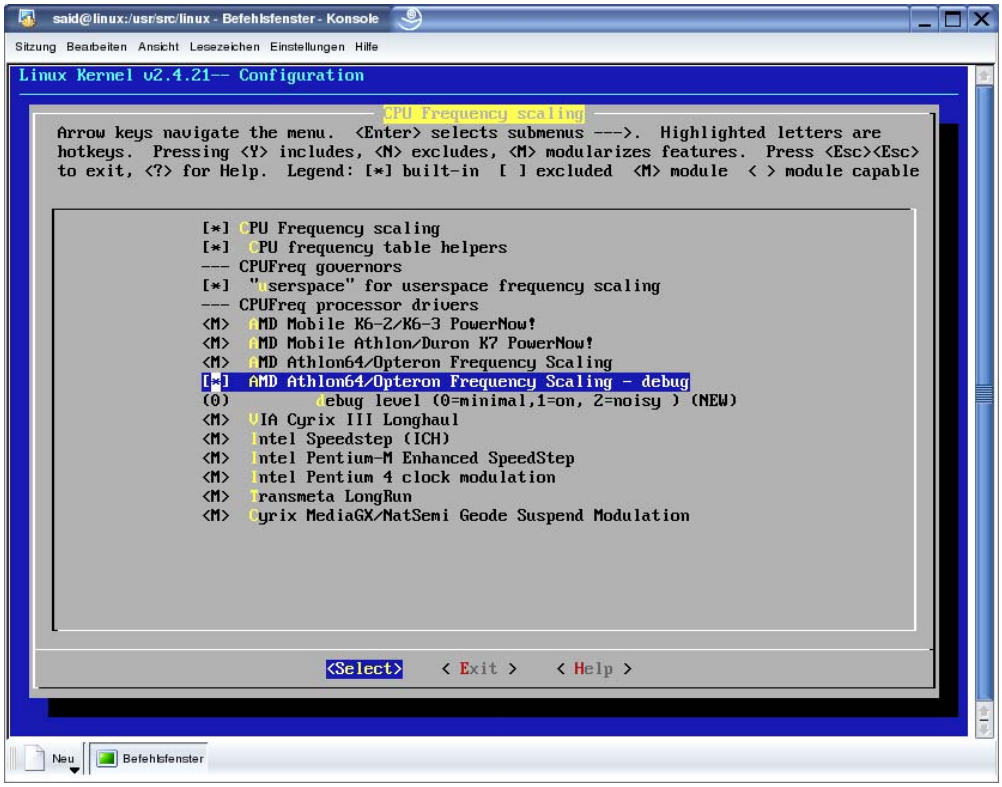


Abbildung 5- 8: Untermenü (CPU frequency scaling) von Option 3.

4. **General setup: Networking support** (s. Abbild. 5.9 und 5.10): Diese Option wird auf jeden Fall benötigt, da einige interne Kommandos auf dem Netzwerkprotokoll aufbauen. Die korrekten Netzwerkoptionen werden erst bei Punkt 13 (Networking options) besprochen. PCI support wurde gewählt (s. Abbild. 5.11), auf PCMCIA/CardBud support (s. Abbild. 5.12) wurde verzichtet. SystemV IPC musste auf jeden Fall wegen der benötigten Inter Process Communication bei der Schnittstellenprogrammierung ausgewählt werden. ELF und a.out Kernel Core format werden auch auf „ja“ gesetzt, Power Management support für Laptops mit „ja“ beantwortet, jedoch ACPI support verneint.

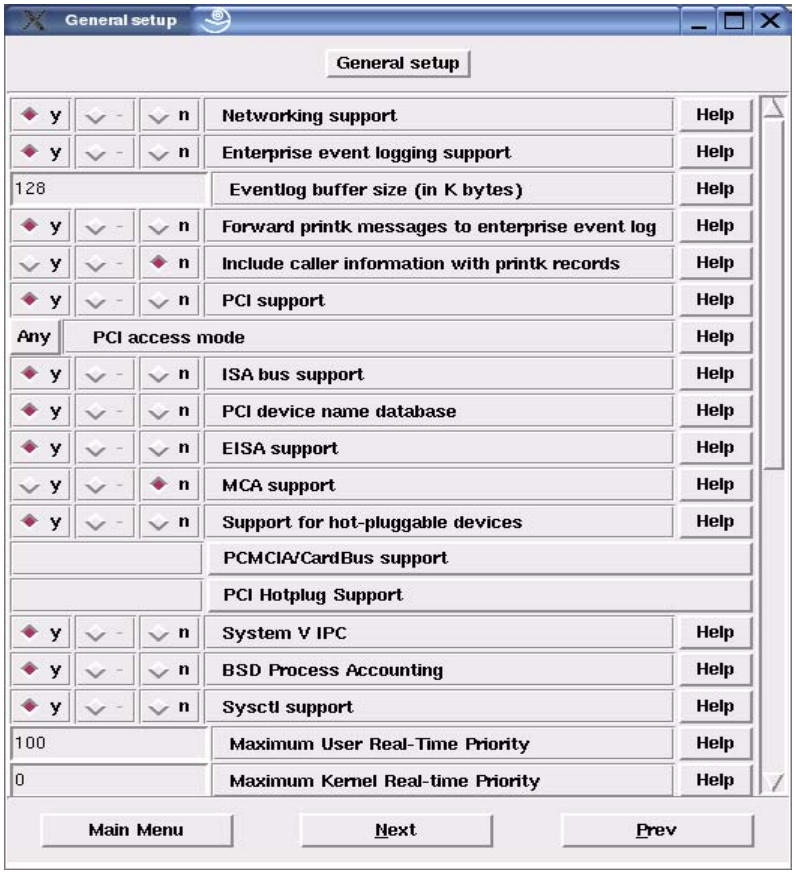


Abbildung 5- 9: General setup(1. Hälfte).

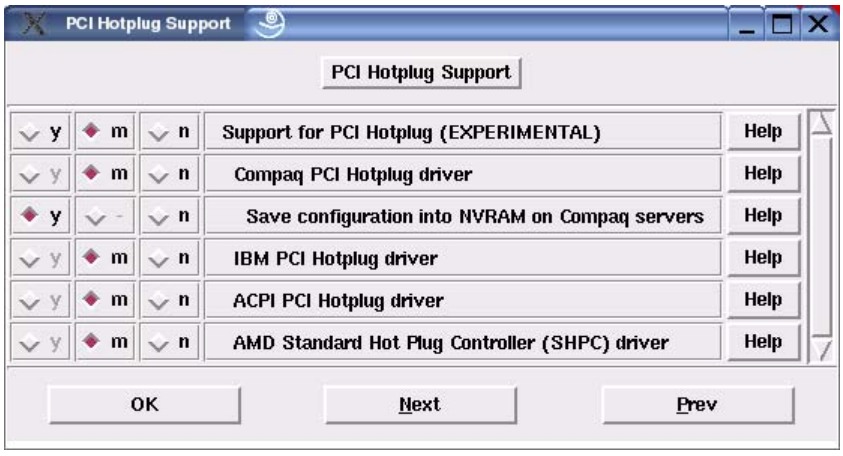


Abbildung 5- 10: Untermenü PCI Hotplug support.

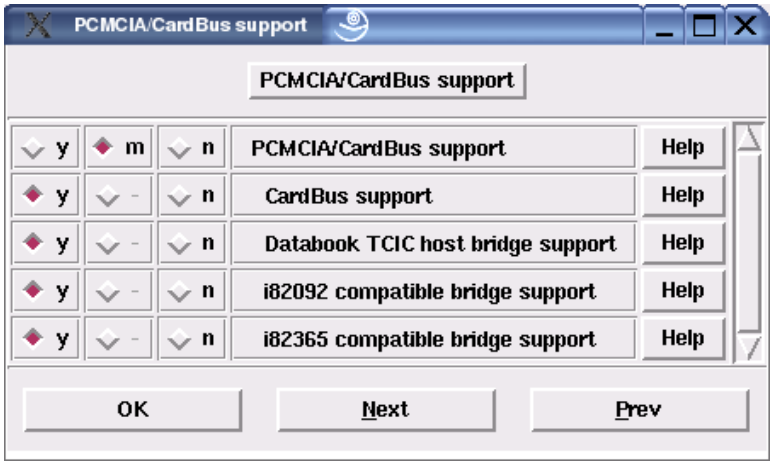


Abbildung 5- 11: Untermenü PCMCIA/CardBus support.

5. **Binary emulation of other systems**: Diese Option ist notwendig, um z.B. SOM- bzw. Solaris-binaries direkt zu laden und durchzuführen. SOM ist ein binär ausführbares Format, das von HP/UX übernommen wird. Auf diese Option wurde verzichtet.

6. **Memory Technology Devices (MTD)**: Damit sind z.B. Flash, RAM gemeint. Meistens werden sie für stabile Systeme bei Embedded Systems benutzt, deswegen ist diese Option im vorliegenden Fall sehr wichtig und musste mit „ja“ beantwortet werden. Dafür wurde keine Partition ausgewählt, da nur ein Flash Disk vorhanden ist. Sogar der Flash Disk von der Firma Arcom wird hier zum Aktivieren angegeben. Der Treiber für die SBC-GXn Board-Familie wurde gleichfalls aktiviert.

7. **Parallel port support**: Hier geht es um die parallele Schnittstelle, an die normalerweise der Drucker angeschlossen wird. Im vorliegenden Fall ist nicht geplant, einen Drucker einzusetzen, deshalb wurde „nein“ angewählt.

8. **Plug and Play configuration**: Ist ein Standard für Plug and Play-Karten, die bei der Erkennung softwaremäßig konfiguriert werden können.

9. **Block devices**: Ein Diskettenlaufwerk wird im vorliegenden Fall nicht gebraucht, ein RAID-Controller ebenfalls nicht. Normale (nicht SCSI) Festplatten und CD-ROM-Laufwerke (unter anderem) werden unter Linux als Block devices erkannt.

10. **Enterprise Volume Management System**: Dieses ist ein plugin-basiertes Framework für Datenträgermanagement und kombiniert die Unterstützung für das Partitionieren, Software, RAID, LVM und vielem mehr in einer einzigen Schnittstelle. Benutzer-Tools sind nötig, um die Administration von EVMS logischen Datenträgern durchzuführen, entsprechend wurde diese Option aktiviert.

11. **Multi-device support (RAID and LVM)**: RAID und Logical Volume Management werden bei unserem Board Computer nicht benötigt, also wird die Option „nein“ gewählt.

12. **Cryptography support (CryptoAPI)**: Gewährleistet die Authentifizierung und die Verschlüsselung und vermutlich die verschlüsselte Authentifizierung und Übertragung über das Netzwerk (kein Hilfetext), entsprechend wird „nein“ angewählt.

13. **Networking options**: Network packet filtering ist für die Firewalls zuständig. Im vorliegenden Projekt werden sie nicht benötigt, also „nein“ angewählt. Die Option TCP/IP networking wurde mit „ja“ beantwortet.

14. **Telephony Support**: Wurde nicht gewählt. Für den Fall, dass telefoniert werden soll, muss ein Hardware-Telefon angeschlossen werden.
15. **ATA/IDE/MFM/RLL support**: Hier bietet sich die Möglichkeit, Geräte zu unterstützen, die am IDE-Bus angeschlossen werden wie z.B. IDE-Festplatten und ATAPI-CD-ROMs.

16. **SCSI support**: Eine Unterstützung für die SCSI Festplatte ist nicht notwendig. Im Gegensatz dazu wird die Option SCSI-CD-ROM mit „ja“ beantwortet. SCSI low-level drivers soll nicht unterstützt werden, da sonstige SCSI-Hardware nicht vorhanden ist.

17. **Fusion MPT device support**: Die Option Message Passing Technology bietet als Schnittstellendienst die Möglichkeit einen Systemhost zu aktivieren

entweder als leistungsstarker SCSI Host Initiator oder als LAN (aber nicht an parallele SCSI-Medien). Die Fusionsarchitektur ist in der Lage, diese Protokolle bidirektional in Hochgeschwindigkeitslaser (bis 2GHz*2Ports = 4GHz) und parallele SCSI (bis zu Ultra-320) physikalische Medien abzuarbeiten. Diese Treiber benötigen einen im Systemhost installierten MTP-kompatiblen PCI-Adapter, diese Adapter enthalten spezielle I/O-Prozessoren. Folglich wurde diese Option mit „nein“ beantwortet.

18. **IEEE 1394 (FireWire) support (EXPERIMENTAL)**: FireWire-Geräte sind nicht vorhanden, daher soll der FireWire-Anschluss nicht unterstützt werden.
    Diese Option wäre z.B. inaktiv, falls die erste Option(Code maturity level) mit „nein“ beantwortet worden wäre, weil die FireWire unter Linux bis zum Kern 2.4. noch nicht für ausgereift deklariert wurde.

19. **I2O device support**: Ist eine intelligente Input/Output Architektur, die es erlaubt, Hardwaretreiber in zwei Anteile zu teilen: Ein betriebssystemspezifisches Modul (OSM) und ein hardwarespezifisches Modul (HDM). Allerdings benötigt man dafür eine I/O-Schnittstellenadapterkarte im Computer. Diese Karte enthält einen speziellen I/O-Prozessor, der eine hohe Geschwindigkeit erzeugt, da die CPU sich nicht mehr mit dem I/O beschäftigen braucht. Weil die I/O-Schnittstellenadapterkarte nicht vorhanden ist, wurde „nein“ angewählt.

20. **Network device support**: Wenn man in irgendeiner Form mit einem anderen Rechner kommunizieren möchte, dann ist die erste Option (Network device support) mit „ja“ zu beantworten. Im Anschluss sollen die Art der Kommunikation und die Hardware spezifiziert werden. Alle folgenden Untermenüs wurden mit „nein“ beantwortet: ARCnet devices, Ethernet (1000Mbit), FDDI driver support, HIPPI driver support (EXPERIMENTAL), Wireless LAN (non-hamradio), TockenRing devices, Fibre Chanel driver support, Red Creek Hardware VPN (EXPERIMENTAL), WAN interfaces. Im Gegensatz dazu wurden PPP (point-to-point protokoll) support, SLIP (serial line) support auf „ja“ gesetzt. Unter der Option Ethernet (10 or 100Mbit) wurden die Angaben zur vorliegenden Netzwerkkarte bejaht.

21. **Amateur Radio support**: Hierdurch kann die Computer-Kommunikation über Amateurfunk ablaufen. Eines dieser Protokolle kann entweder für point-to-point oder als Träger für TCP/IP verwendet werden. Die Option wurde nicht bejaht, da erstens keine entsprechende Hardware dafür vorhanden ist, und zweitens nicht über Funk mit anderen Computern kommuniziert werden soll.

22. **IrDA (infrared) support**: Hier findet man Treiber für die Infrarot-Schnittstelle. Diese wird nicht benötigt, entsprechend wurde die Option verneint.

23. **ISDN subsystem**: Hier sind Treiber für die unter Linux unterstützte Hardware und Optionen für diverse ISDN-Geräte zusammengestellt, daher wurden diese Optionen mit „nein“ beantwortet.

24. **Input core support**: Human Interface Device (HID) bezeichnet eine bestimmte Geräteklasse des USB-Standards für Computer. Mit der Bezeichnung werden solche Geräte beschrieben, die direkt „mit dem Menschen interagieren“. Gemeint sind damit Geräte, mit denen der Benutzer direkt mit dem Computer kommunizieren kann, etwa Mäuse und Tastaturen, aber auch Gamepads, Grafiktabletts und Joysticks. HID-Gerätetreiber sind normalerweise in den gängigen Betriebssystemen enthalten. Wird ein HID-Gerät (während des Betriebs) angeschlossen, wird es meist direkt als Gerätetyp Eingabegeräte (Human Interface Devices) erkannt und dann im Gerätemanager angezeigt. Die Unterstützung solcher Eingabegeräte wird nicht benötigt und deshalb „nein“ angewählt.

25. **Character devices**: Um überhaupt irgendetwas auf dem Monitor angezeigt zu bekommen, müssen die beiden ersten Optionen (Virtual terminal & Support for console on virtual terminal) mit „ja“ beantwortet werden. Das Gleiche gilt für die Tastatur. Standard/generic (8250/16550 and compatible UARTs) serial support sollte auf jeden Fall „bejaht“ werden, da im Rahmen der vorliegende Arbeit die Kommunikation über serielle Schnittstellen gewährleistet werden soll. Eine Maus wird nicht benutzt, deshalb ist die Option „Mice“ mit „nein“ beantwortet worden.

26. **Multimedia devices**: Hier hat man die Möglichkeit, Audio/Video und FM-Karten zu unterstützen, verschiedene Videoadapter und –Treiber werden normalerweise im Kern gebunden. Diese Unterstützung und auch jene für Funk werden nicht benötigt.

27. **File systems**: Die erste Option (Quota support) dient dazu, den Platzbedarf einzelner Benutzer einzuschränken, entsprechend kann sie verneint werden, da nur ein Benutzer vorhanden ist. Es folgt eine Reihe von Optionen, die Dateisysteme verschiedener Plattformen und Betriebssysteme unterstützen können, wie ADFS, Amiga FFS, Apple HFS/HFS+, NTFS (read only), OS/2 HPFS file system support und DOS Fat fs support. Diese Optionen wurden deaktiviert. Nur das Linux Dateisystem wurde bejaht, also Reiserfs support und Ext3 journalling file system support. Second extended fs suport ist das Linux Dateisystem, das Minix fs support abgelöst hat.
    Die Option ISO 9660 CDROM file system support ist mit „ja“ zu beantworten, so dass CD-ROM-Laufwerke verwendet werden können. Die Microsoft Joliet CDROM extensions wird nicht gebraucht, die Transparent decompression extension ebenfalls nicht. Dieser Punkt bedarf eine Erläuterung:
    Die Option ist eine Linux-spezifische Erweiterung, um Daten in komprimierter Form auf CD-ROMs zu speichern und dann diese CDs transparent und unkomprimiert angezeigt zu bekommen!
    Unter der Option Native Language Support wurde alles, bis auf Codepage 437 (United States, Canada), Codepage 850 (Europe) und NLS UTF8 verneint, damit sind Zeichensätze gemeint, die für den Zugriff auf fremde Dateisysteme unterstützt werden sollen. Die Option Default NLS Option wurde mit „iso8859-1“ angegeben, diese Standard, der aus der HTML-Welt bekannt ist.
    Network File Systems: Die erste Option (Coda file system support (advanced network fs)) ermöglicht es, Dateisysteme fremder Rechner via Netz zu nutzen. Dies bietet im Vergleich zu NFS eine Reihe von Vorteilen: Unterstützung von nichtgebundenen Operationen (z.B. für Laptops), Sicherheitsmodelle durch Authentifizierung und Verschlüsselung gleichfalls. Beide Optionen wurden als Module aktiviert. Root file system on NFS wurde ebenfalls aktiviert, ebenso auch CIFS support (advanced network file system for Samba, Window and other CIFS compliant). SMB file system support (to mount Windows shares

etc.) und NCP file system support (to mount NetWare volumes) sind im vorliegenden Fall nicht von Nutzen.
Der letzte Untermenü: Partition Types wurde nicht gewählt. Bedeutet aber, wenn man Festplatten benutzen möchte, die unter einem anderem zu wählenden Betriebssystem(e) partitioniert worden sind.

28. **Console drivers**: Hier muss die Option VGA text console auf „ja“ gesetzt werden, um Text auf VGA-Karten darstellen zu können, MDA text console (dual headed) (EXPERIMENLAL) sollte ebenfalls bejaht werden, wenn man z.B. mit zwei Monitoren arbeiten möchte. Im Rahmen der vorliegenden Arbeit sollte dies jedoch nicht der Fall sein.

29. **Sound**: Hier werden sowohl verschiedene Soundkarten als auch Treiber zur Sounduntersützung angeboten. Diese werden jedoch nicht benötigt.

30. **USB support**: Zunächst sollen hier USB-Bezeichnungen kurz definiert werden. USB steht für Universal Serial Bus. Bis zu 127 USB-Geräte können an einem USB-Port in einer Baumstruktur angeschlossen werden. Der erste USB-Port ist die Wurzel (root) des Baumes, die Peripherie umfasst die Äste und die inneren Knoten sind spezielle USB-Geräte, die so genannten hubs. Diese Definition ist der Hilfedatei entnommen worden. Der vorliegende Embedded Computer enthält zwar zwei USB-1.1-Ports, sie werden aber nicht eingesetzt, entsprechend wird die Option verneint.

31. **Bluetooth support**: Ist die aktuelle Alternative zur Infrarotkommunikation bzw. –übertragung. Hier sind bis zu 10 Meter Abstand im Freien möglich und 5 Meter Abstand mit Gegenständen in der Übertragungsstrecke. Da eine solche Kommunikation hier nicht relevant ist, wurde die Option deaktiviert.

32. **Kernel hacking**: Hiermit werden ausführliche Fehlermeldungen für die Personen, die Gerätetreiber schreiben, unterstützt.
Kernel profiling support: stellt unter /proc/profile Informationen zur Verfügung, mit denen bestimmt werden kann, wieviel Zeit der Kern für bestimmte Funktionen benötigt.

Profile shift count: Bestimmt den Adressabstand, mit dem die einzelnen vom Kern ausgeführten Befehle in /proc/profile aufgelistet werden.

Hierzu ein Zitat aus der README-Datei von Linux:
Die Aktivierung der „Kernel hacking“ Option führt im Normalfall zu einem größeren oder langsameren Kern (oder sogar beides). Dadurch kann der Kern sogar definitiv instabiler werden, da manche Routinen dann etwas unterschiedlich kompiliert werden, um gezielt schlechte Routinen zum Absturz zu bringen und dadurch Fehler im Kern aufzudecken (kmalloc()).
Soll der Kern ganz normal verwendet werden, sollte man deshalb hier mit „nein“ antworten.

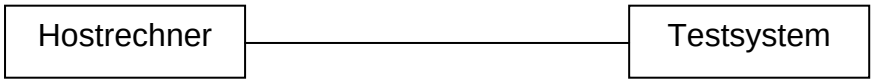
Die Optionen **Library routines**, **Plug In CPU Schedulers**, **Support Resource Management Modules**, **Build options** waren im Kern dieser Arbeit nicht vorhanden und sind entsprechend in den Hilfedateien nicht vertreten. Folglich waren alle zu deaktivieren.

5.2.2. Module erzeugen und installieren

Damit ist die Konfiguration vollendet worden, nun müssen die Abhängigkeiten mit dem Befehl „make dep“ hergestellt werden, im Anschluss die Module erzeugt (make modules) und installiert (make install_modules) werden. Diese Kommandos kann man hintereinander mit Semikolon getrennt ausführen.
Da alles problemlos gelaufen ist, wurde der Kern (make bzImage) kompiliert. Der erzeugte Kern heißt bzImage und liegt im Unterverzeichnis arch/i386/boot/, er wurde dann ins Bootverzeichnis mit dem Namen vmlinuz verschoben (man kann auch das Wurzelverzeichnis dafür verwenden). Von dort aus kann er durch Hinzufügen in der GRUB-Konfigurationsdatei (GRUB steht für Grand Unified Bootloader und ist die andere Alternative zur LILO) gebootet werden. GRUB muss nochmals aufgerufen werden, damit diese Änderungen übernommen werden. Dieser Schritt konnte unter „yast“ bequem verarbeitet werden, indem der Eintrag für Linux kopiert und eingefügt wurde, und dann der neue Eintrag dem vorliegenden Kern angepasst wurde, also der Pfad der rootpartition und der Namen des neuen Kerns.

Zu Beginn war geplant, dem Embedded Computer eine IDE-Festplatte hinzuzufügen. Auf dieser sollte erstmals das Ganze konfiguriert und installiert werden. Weil dies unter Suse nicht erreicht werden konnte, wurde auf RedHat umgestiegen. Hier waren das Flash Disk und der spezielle Prozessor unterstützt. Es existiert eine andere elegante und professionelle Alternative. Hierbei wird der Kern über einen Host-Rechner übertragen. Im nächsten Kapitel soll diese Alternative genauer erläutert werden.

5.3. Entwicklungsumgebung



Das Testsystem besteht aus einem Desktop-PC als Embedded-Rechnerersatz, auf dem auch andere Software oder sogar ein anderes Betriebssystem installiert sein darf. Natürlich soll auf dem Hostrechner eine Linuxdistribution (im vorliegenden Fall war es RedHat) mit allen Entwicklungstools installiert sein, wo dann der Kern vorbereitet, getestet und für das Testsystem aufbereitet werden soll.
Durch diese Variante wird der Kern nicht physikalisch auf dem Datenträger des Testsystems vorhanden sein, sondern über die Netzwerkverbindung übertragen. Hierfür wird NFS zur Hilfe genommen. Das Testsystem dient lediglich in der Testphase als Ersatz für den Embedded Rechner, da er über Tastatur, Graphikkarte und Monitor verfügt.
Zusätzlich zur Umgebung wird eine leere Diskette benötigt, auf der der Bootloader installiert wird, der den Kern in dem Speicher laden soll.

1. Schritt: Als Bootloader wurde syslinux gewählt. Drei Gründe sprachen dafür:

1 – Syslinux sollte auf jedem Linux bereits vorhanden sein.

2 – Syslinux lässt sich auf DOS-formatierten Disketten mit FAT-Dateisystem installieren.

3 – Der Bootloader eine Konfigurationsdatei zur Verfügung stellt, die vom großen Nutzen ist.

Der Kern selbst muss auf der Diskette kopiert werden. Dies geschieht auf dem Hostrechner, um dann das Testsystem von der Diskette zu starten.
Leere Diskette kann mit „mformat a:“ formatiert werden, falls dies nicht der Fall ist und dann soll folgendes Kommando „syslinux /dev/fd0“ als root eingeben werden, was syslinux auf der Diskette installiert, da /dev/fd0 die Bezeichnung für das Diskettenlaufwerk unter linux ist. Falls syslinux noch nicht installiert ist, einfach „yast“ aufrufen und syslinux mit Quellcode installieren.
Wenn man jetzt schon das Testsystem von dieser Diskette bootet (BIOS entsprechend einstellen, dass der Rechner vom Diskettenlaufwerk startet), dann bekommt man eine Fehlermeldung „Could not find Kernel Image“. Dies weist darauf hin, dass syslinux problemlos auf der Diskette installiert wurde, da syslinux eine Datei namens linux (Kernel Image) zum Booten erwartet. Falls man den Kern anders genannt hat, gibt es die Möglichkeit, dem Prompt bei gedrückter Shift-Taste oder Alt-Taste während des Bootvorgangs einen anderen Namen bzw. Kernparameter zu übergeben. Eine andere Alternative, die hier auch benutzt wurde, ist die Konfigurationsdatei syslinux.cfg, wo dann diese Bootparameter angegeben worden sind.
Folgende Bootparameter wurden angegeben:
Das Root-Dateisystem wird nicht auf die Diskette geschrieben, sondern wie schon erwähnt, per Network File System (NFS) vom Hostrechner geladen. Syslinux wurde beauftragt, das Root-Dateisystem vom NFS-Server zu holen. Die IP-Adresse des Testsystems wird über DHCP ermittelt: append root=/dev/nfs ip=dhcpcd.

2. Schritt: Nun muss noch der Kern selber auf die Diskette übertragen werden. Wie der Kern kompiliert wird, haben wir im letzten Abschnitt gesehen. Die Datei bzImage war das Ergebnis der Kernkonfiguration, des Testens der Abhängigkeiten, der Erzeugung und Installation von Modulen und zuletzt der Kompilation des Kerns. Diese habe ich auf der Diskette mit dem Namen linux gespeichert, um dann von der Diskette starten zu können. Dieser Bootvorgang führt dann erneut zu einer anderen Fehlermeldung: „Kernel Panic: VFS: unable to mount root“, wozu diese Konfigurationsdatei syslinux.cfg auf der Diskette gespeichert wurde.
Damit das Booten vom Hostrechner möglich wird, muss der Hostrechner sowohl als NFS-Server, um das Root-Dateisystem zur Verfügung zu stellen, als auch DHCP-

Server konfiguriert werden, um dem Testsystem eine dynamische IP-Adresse zu übergeben.
Wenn diese Schritte vollzogen werden, sollte das Testsystem von der Diskette einwandfrei gebootet, den Kern von der Diskette geladen, rootfs vom NFS-Server gemountet und die IP-Adresse vom DHCP-Server zugestellt werden können. Es war nicht mehr notwendig, die genannten Punkte durchzuführen.
Der letzte Schritt zur Installation des Betriebssystems ist die Echtzeiterweiterung. Diese wird im folgenden Abschnitt genauer erläutert.

5.4. Echtzeiterweiterung

Moderne Applikationen haben meistens die Anforderung, dass alle Vorgänge in der Automatisierungseinrichtung zeitlich zu einander synchronisiert werden müssen.
Deswegen finden Echtzeiterweiterungen vor allem von Linux zunehmende Popularität in der Industrie sowie in den Universitäten.

Mit Echtzeitfähigkeit sind Anforderungen gemeint, die mit Zeitabläufen auf einem Rechner zu tun haben. Echtzeitanwendungen dürfen unabhängig von der Ursache nicht verzögert bzw. verhindert werden. Im Rahmen dieser Arbeit muss auf Ereignisse, die von der Transceiverkarte (Sende-/Empfangskarte) z. B. ausgelöst werden, in genau definierter Zeit eine Reaktion der Aktorik, d.h. des gesamten Luftschiffes, erfolgen.

Die Echtzeitfähigkeit ist standardmäßig bei den Rechnern und den gängigen Betriebssystemen nicht vorhanden. Doch mit RTAI und RTLinux besteht die Möglichkeit, Linux so zu erweitern, dass ein kleiner Echtzeitkern zur Verfügung steht, der garantierte Antwortzeiten gewährleistet. Ein Vorteil von diesen Linux-Echtzeiterweiterungen besteht darin, dass Linux im Normalfall (wenn der Echtzeitkern gerade nichts zu tun hat) mit dem normalen Standardkern arbeitet.

5.4.1. Latenz und Fluktuation

Das Resultat dieser Änderungen ist die Unterbrechungslatenz und –fluktuation (s. Abb. 5-12 und 5-13) zwischen periodischen Unterbrechungen im Mikrosekunden-bereich zu verringern und so schnellere Antworten zu externen Ereignissen und eine höhere Timing-Auflösung zu ermöglichen.
Der Standardkern von Linux zeigt Latenz (latency) und Fluktuation (jitter) von einer Millisekunde. Die Echtzeitversionen von Linux haben Latenz und Fluktuation einiger Mikrosekunden auf den Prozessoren, die mit mehreren Hundert MHz laufen.

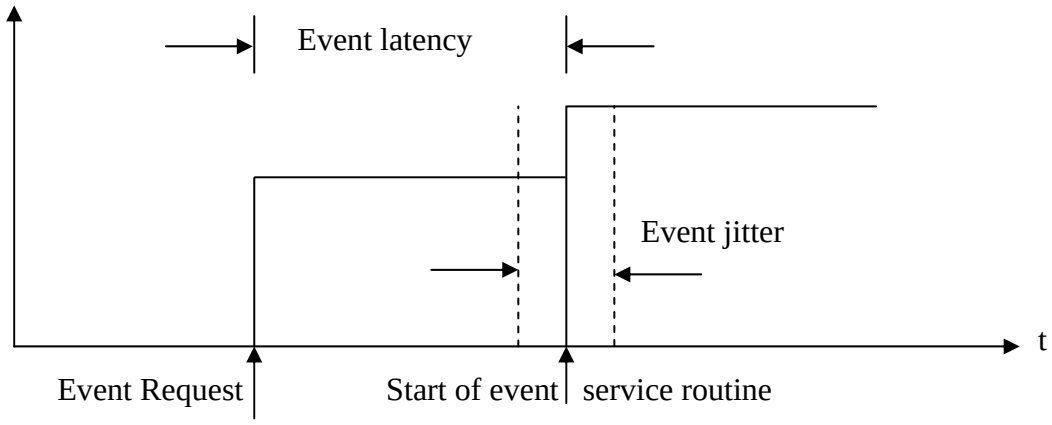


Abbildung 5- 12: Die Ereignislatenz.

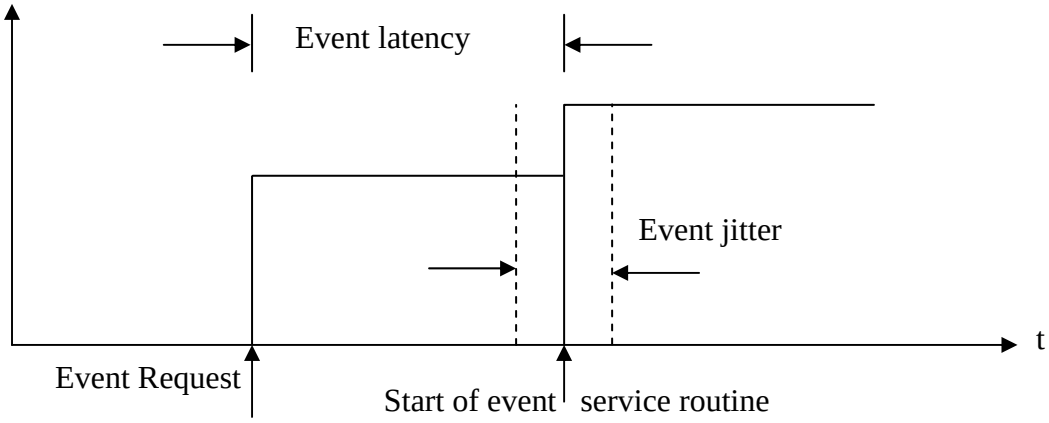


Abbildung 5- 13: Periodische Fluktuation.

Die Latenz ist die Verspätung von einer Unterbrechung zum Anfang der Verarbeitung dieser Unterbrechung. Die Abweichung zwischen vorgegebenem und tatsächlichem Wert wird als Latenzzeit bezeichnet. (In der Praxis wird eine gewünschte Zeit nicht immer exakt identisch sein, da das System auch Zeit für interne Abläufe benötigt).
Die Fluktuation ist die Veränderung des Timings der periodischen Fälle bzw. ist die statische Verteilung der Latenzzeit.

5.4.2. Weiche und harte Echtzeitbedingungen

Man unterteilt die zeitkritischen Aufgaben in folgende Punkte (s. Abb. 5-14):

⟹ Weiche Echtzeitbedingungen: Reaktionszeiten müssen gering sein, jedoch hat eine kurzzeitige Überschreitung gewisser Grenzen keine Größeren Folgen. Das heißt, dass die Zeitvorgabe im Mittel einzuhalten ist.

⟹ Harte Echtzeitbedingungen: Gewisse Reaktionszeiten müssen vom System unter allen Umständen garantiert werden. Falls diese überschritten werden, treten Schäden an der Anlage auf.

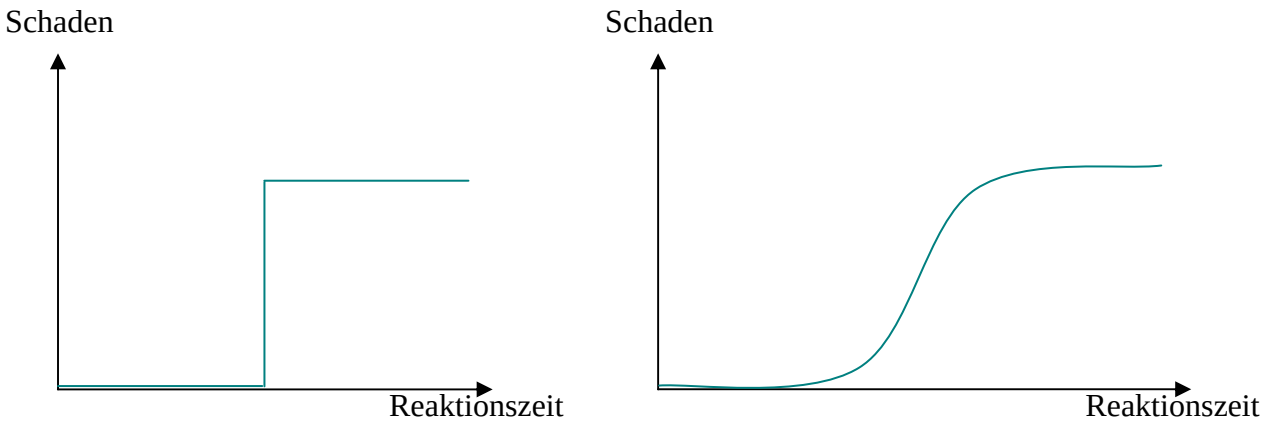


Abbildung 5- 14: Schadensänderung in Abhängigkeit von der Reaktionszeit.

Normalerweise ist Linux wie jedes andere Betriebssystem, nur für weiche Echtzeitaufgaben geeignet (s. Tabelle 5-1).

Tabelle 5- 1: Vergleich der Leistung von Linux mit den kommerziellen Echtzeitkernen.

	Applikation	Latenz/Fluktuation
Standard BS.	Keine Echtzeit	10 µs – 100 ms
Standard Linux	Weiche Echtzeit	1 µs
IEEE 1003.1d	Harte Echtzeit	10 – 100 µs
Linux Mikrokern	Harte Echtzeit	1 – 10 µs
RTOS-Kerne	Harte Echtzeit	1 – 10 µs

Und zwar ist die Antwortzeit aus verschiedenen Gründen nicht deterministisch:

- Die Kernfunktionen des Betriebssystems sind nicht unterbrechbar, was dazu führt, dass andere Anforderungen bis zum Abschluss der Operation warten müssen.
- Die Zeitauflösung auf der x86-Plattform beträgt 10ms und ist damit gering z.B. für den Scheduler, der sehr wichtig, sogar notwendig für Echtzeitaufgaben ist.
- Der Zugriff auf Festplatten und auf Kommunikationseinrichtungen hat grundsätzlich keine definierte Antwortzeit, kann sogar die aufzurufende Funktion solange blockieren, bis alle Daten vorliegen [?].

Es ist möglich, Standard Linux für weiche Echtzeitsteueranwendungen zu verwenden, in denen die Abtastzeit von ca. 10 ms verhältnismäßig lang ist und die Anwendung vermisste Zeitpläne zulassen kann. Ein Bsp. Dafür ist die Videoverarbeitung, wo ein gelegentliches Aufflackern häufig nicht wahrgenommen wird.
Wenn Datenverluste zugelassen werden können, dann kann Standard Linux (besonders die neuesten Kerne) für weiche Echtzeitanwendungen verwendet werden, die die Unterhälfte (preemption logic) besitzen.

Unterbrechungsverarbeitung im Standardkern wird in zwei Hälften geteilt, die als Oberhälften- und Unterhälftenaufgaben gekennzeichnet werden. Die Unterhälfte erfüllt die Aufgaben, die Unterbrechung zu bearbeiten und Daten von dem physikalischen Medium in einem Zwischenspeicher unterzubringen. Die Oberhälfte liest vom Zwischenspeicher und übermittelt die Daten zu einem zugänglichen Puffer des Kerns. Im Standardkern ohne Unterbrechungssteuerung, sind alle

Unterbrechungen untauglich, wenn die Unterhälfte Aufgaben ausführt. Das heißt, dass es eine unvorhersehbare Verzögerung geben kann (die Latenz), bevor eine zweite Unterbrechung vorgenommen werden kann.

Wenn das nicht der Fall ist, dann werden die harten Echtzeitimplementierungen benötigt.

Die Echtzeitvarianten von Linux, die für solche Anwendungen verwendbar sind, sind seit 1997 erschienen. Und zwar existieren hauptsächlich zwei Implementierungen zur Realisierung harter Echtzeitfähigkeit unter Linux:

- RT-Linux wurde in New Mexiko von einer Arbeitsgruppe um Professor Victor Yodaiken und Michael Barabanov entwickelt. Diese Variante soll eine neue in hohem Grade leistungsfähige Codeschicht zwischen der Hardware und dem Standardkern bilden. Die zusätzliche Codeschicht, Mikrokern genannt, steuert die ganze (gesamte) Echtzeitfunktionalität, einschließlich der Interrupts und legt hoch auflösendes Timing fest. Dieser Mikrokern lässt den Standardkern im Hintergrund laufen.
- RTAI wurde in Milano (Politecnico di Milano) von einer Arbeitsgruppe um Professor Paulo Mautegazza entwickelt. Diese zweite Variante soll die Echtzeiterweiterungen zu POSIX.1 innerhalb der Struktur des Linux-Standardkerns einführen. Sie fügt Timer, Terminplanung (Scheduling) und Unterbrechungskontrolle (Preemption) direkt in einem einzelnen Kern zusammen.

Diese Implementierungen werden als Ergänzung (Patch) zum Standardkern zur Verfügung gestellt. Der Mikrokern fängt die Hardwareunterbrechungen (Hardware interrupts) ab und legt auch die Echtzeitaufgaben mit der höchstmöglichen Priorität fest.

Der Echtzeitkern wickelt als direkte Schicht über die Hardware die Interruptverwaltung des kompletten Systems ab. Dieser Echtzeitkern garantiert die Antwortzeiten für Echtzeitprogramme. Normalerweise ist Linux doppelschichtig (Kernel Space & User Space), wird jedoch mit der Echtzeiterweiterung um eine dritte Schicht (Realtime Space) erweitert: Diese neue Schicht überwacht dann das Kernel Space und hat damit die Möglichkeit den Kerncode zu unterbrechen.

Nachdem die Echtzeitaufgaben abgearbeitet worden sind, stellt das Realtime Space alle Register wieder her, so dass der Normalkern nichts davon merkt, dass er unterbrochen wurde.
Wenn das Realtime Space keine Echtzeitaufgaben erledigen muss, benötigt es keine Rechenzeit und macht sich somit für den normalen Linux-Kern unbemerkbar.

Gegenüberstellung der Vor- und Nachteile der beiden Varianten:
RTLinux: Das Entwicklungsteam um Victor Yodaiken entscheidet darüber, ob die Patches der Benutzer in das Konzept passen, andernfalls werden diese Patches zurückgewiesen. Zwar wird dadurch ein konsistentes Modell erreicht, jedoch finden innovative Ideen keine Möglichkeit in die Software aufgenommen zu werden.

RTAI: es Widerspricht diesem anderen konsistenten Modell, in dem viele Benutzer Entwicklungen in Form von Kernmodulen zur Software hinzugefügt haben. Andererseits macht diese Vorgehensweise das gesamte System flexibler und innovativer.
Zwar unterliegen beide Varianten der GNU-Lizenz, aber zumindest RTAI ist heutzutage problemlos zu bekommen.

5.4.3. Eingebettete Anwendungen

Echtzeitsysteme können auch eingebettete Systeme sein. Eingebettete Anwendungen laufen gewöhnlich in kleinen Hardwarekomponenten bzw. -karten, häufig ohne Tastaturen, Mäusen oder Monitoren und normalerweise ohne drehende Medien wie Festplatten oder CD-ROMs, die harten Arbeitsbedingungen nicht widerstehen können. Programmierer können Linux zusammenbauen, so dass es ohne diese Peripheriegeräte läuft und es gibt mehrfach populäre eingebettete Linux-Verteilungen, die vorkonfiguriert vorhanden sind und nützliche Werkzeuge für kundenbezogene Spezialanfertigungen mit einschließen [11].

Die üblicherweise verwendete Plattform für Standard Linux ist ein Intel-basierter Rechner oder Laptop, wobei die Anwendungen möglicherweise nicht Echtzeitleistung benötigen und die Latenz und die Fluktuation von Standard Linux keine Beschränkung für diese Anwendungen sind. Doch es gibt eingebettete

Hardwarekomponenten, die deterministische Echtzeitleistung erfordern und für diese Fälle muss eine der Linux-Echtzeitimplementierungen in der eingebetteten Anwendung verwendet werden. So kann eingebettetes Linux auf den Hardwarekomponenten laufen, die vom Desktop Rechnern bis zu Videoarmbanduhren reichen, aber ein eingebettetes Linux ist nicht notwendigerweise Echtzeitlinux.

Da der normale Linuxkern vom Echtzeitkern nichts merkt, muss es die Möglichkeit geben, Echtzeit und normale Prozesse miteinander kommunizieren zu lassen. Dies geschieht mit den folgenden Techniken: FIFOs, Shared Memory, Mailboxen und Message Queues.

5.4.4. Installation von Linux-Echtzeit

Die Schritte, die der Reihe nach ausgeführt werden, um ein Echtzeitlinux zu installieren, sind folgendermaßen zusammen zu fassen:

1. Selbstkonfigurierung des Patchs:
   Bevor man damit anfängt, den RTLinux zu installieren, muss man die entsprechenden Linux- und RTLinux-Versionen herunter zu laden. In der vorliegenden Arbeit wurde mit 2.4 Kernversion gearbeitet, wofür diese und die RTLinux-Version 3.1 herunter geladen wurden (s. Downloaden). Da mußte auch die C-Compiler-Version beachtet werden, da sonst Probleme entstehen können, und zwar ist die gcc 2.7.2.3 eine Mindestanforderung. Danach müssen folgende Arbeitsschritte unternommen werden:
   - Die herunter geladenen Dateien müssen in einem temporären Ordner zwischengespeichert werden, und zwar unter ,,/var/tmp''.
   - Einen Arbeitsordner vorbereiten und leeren, wenn er vorhanden ist: md /usr/src/rtlinux (rm –rf /usr/src/rtlinux), dann von dort aus die herunter geladenen Dateien entpacken mit dem Befehl: tar –xzvf /var/tmp/linux-2.4.4.tar.gz /var/tmp/rtlinux-3.1.tar.gz.
   - Den Kern mit der RTLinux-Ergänzung im Unterordner linux ausbessern: patch –p1 < rtlinux/kernel_patch-2.4.4.

- Dann den Kern wie folgt konfigurieren, kompilieren und installieren, wie im letzten Kapitel:,, make menuconfig; make dep; make bzImage; make modules; make modules_install;''
- Die erstellte Datei kopieren: cp arhc/i386/boot/bzImage /boot/rtlinux, und nachdem der Boot loader angepasst wurde, den Computer neustarten.
- Jetzt muss der Computer mit dem neuen Kern neu gestartet und einfachheitshalber ein symbolischen Link erstellt werden: dem Ordner ln –sf /usr/src/rtlinux/linux /usr/src/linux, in dem dann der RTLinux mit dem Befehl ,,make menuconfig; make dep; make; make devices und make install'' konfiguriert und kompiliert werden muss.
- Um jetzt Echtzeitprogramme ausführen zu können, müssen die RTLinux-Module durch den Befehl rtlinux start geladen werden.

2. Eine vorhandene Ergänzung (Patch) verwenden.
   - In diesem Fall wurde mit der herunter geladenen Datei rtlinux-2.4-prepached.tgz gearbeitet, also konfiguriert, dekomprimiertt, usw. Ansonsten wurde nicht viel geändert.

In beiden Fällen, musste beachtet werden, dass die „harte“ Echtzeit in der Grundlagenkonfiguration gesetzt und im Gegensatz dazu, dass die APM-Unterstützung deaktiviert ist.

6. **Beschreibung von Portzugriffsarten**

Man unterscheidet – wie wir im Folgenden sehen werden – zwischen zwei I/O-Portzugriffsarten: Auf der einen Seite der direkte Zugriff, was auch unter die Bezeichnung Ein-/Ausgabe tiefer Ebene fällt (low level input and output), und der Zugriff über Gerätedateien auf der anderen Seite.

Eigentlich ist hardware-nahe Programmierung bei modernen, komplexen Betriebssystemen auf heutigen Computern ungünstig. Zugriffe auf Adressen außerhalb des zugewiesenen Speichers oder I/O-Ports werden dem „normalen Benutzer“ vom Betriebssystem verwehrt, was früher nicht der Fall war. Man konnte viel einfacher auf die Hardware zugreifen und die maximale Leistung der Programme bzw. einer Komponente erreichen.

Sehen wir uns zuerst die direkte Zugriffsart genauer an. Im Folgenden wird diese direkte Zugriffsart am Beispiel der seriellen Schnittstelle erläutert.

6.1. Direkter Zugriff

Selbstgeschriebene Programme können nicht ohne Weiteres direkt auf Hardware-Komponenten zugreifen, dies wird aus Sicherheitsgründen vom Betriebssystem verhindert. Direkte (Schreib-) Zugriffe auf die Hardware des Computers können zu Systemabstürzen und damit unter Umständen zu Datenverlust führen.
Um trotzdem einen direkten Zugriff zu ermöglichen, müssen Zugriffsrechte erteilt werden. Diese schaffen die Voraussetzung dafür, dass dann die eigentlichen Zugriffe getätigt werden können.

6.1.1. Rechte vergeben

Unter Linux gibt es neben den Zugriffsrechten Lesen (R), Schreiben (W) und Ausführen (X) auch das Recht „set user ID“ (S). Wenn dieses Bit bei einer ausführbaren Datei gesetzt ist, dann erhält der ausführende Prozess für die Dauer der Ausführung die Benutzer-ID dieser Datei. D.h. wenn ein Programm dem Benutzer

„root“ gehört, dann werden dem ausführenden Prozess root-Privilegien verliehen. Z.B. wird mit der Kommandozeile „chown root:root meinprog“ der „Besitzer“ und die Benutzergruppe der Datei geändert und mit „chmod a+s meinprog“ das Bit des Benutzers gesetzt. Wenn jetzt das Programm „meinprog“ vom normalen Benutzer (natürlich sollte der Benutzer das Programm ausführen dürfen) gestartet wird, dann läuft es mit root-Privilegien, und kann wieder entsprechenden Schaden anrichten! Aus diesem Grund sollte ein solches Programm diese Privilegien mit Hilfe des Funktionsaufrufs ,,setuid (getuid()); (#include <sys/types.h> und #include <unistd.h>)“ wieder zurückgeben, sobald sie nicht mehr benötigt werden.

Eine zusätzliche Sicherung gegen „unbeabsichtigte“ Zugriffe auf Hardware-Komponenten des Computers besteht darin, dass dem Benutzer root selber zunächst der Zugriff auf die Hardware verweigert wird (Speicherzugriffsfehler). Die zwei Funktionen iopl() und ioperm() sind dafür gedacht, die I/O-Portzugriffe explizit freizugeben. Diese Funktionen (bzw. eine von beiden) müssen am Anfang (vor jenem I/O-Portzugriff) des Programms aufgerufen werden. Die Zugriffsbibliotheken auf die I/O-Ports werden von der Headerdatei /usr/include/asm/io.h zur Verfügung gestellt. iopl steht für I/O privileged level und ioperm für I/O permissions. Beide Funktionen geben folgende Werte zurück: den Wert 0 bei Erfolg, den Wert -1 bei Misserfolg. Im letzteren Fall wird die Fehlerkonstante auf errno gesetzt.

Übrigens sperrt ein setuid() nicht den Portzugang zu einem normalen Benutzer, der durch ioperm() bewilligt wurde, aber ein fork() (der Kindprozess erhält keinen Zugang, aber der Elternprozess behält ihn) könnte zu diesem Zweck eingesetzt werden.

Die Syntax der ersten Funktion ist int ioperm(unsigned long from, unsigned long num, int turn_on). Dabei ist from die erste Portnummer, auf die zugegriffen werden soll, und num die Anzahl der nachfolgenden Ports. Zum Beispiel würde ioperm(0x3E8, 16, 1) Zugang zu den Ports 0x3E8 bis 0x3F8 erteilen (wobei 3F8 und 3E8 jeweils für die erste und zweite serielle Schnittstelle stehen). Das letzte Argument ist ein boolescher Wert, der entweder dem Programm Zugang zu den Ports gibt (true (1)) oder verbietet (false (0)). Man darf ioperm() mehrfach aufrufen, um nicht-nachfolgenden Ports mehrfachen Zugriff zu ermöglichen.

Man kann die Administratorzugriffsrechte wieder zurückziehen, nachdem ioperm() aufgerufen wurde, um Zugang auf die zu benutzenden Ports zu ermöglichen. Am Ende des Programms wird man nicht aufgefordert, die Portzugangsrechte mit ioperm(..., 0) ausdrücklich zu entfernen. Dieses geschieht automatisch, wenn der Prozess beendet wird.

Die Funktion ioperm() kann nur Zugang zu den Ports 0x000 bis 0x3ff verleihen. Für zusätzliche Ports muss man iopl(), die Zugriff zu allen Ports sofort gewährleistet, nutzen. Eine Beschränkung des Adressbereichs ist leider damit nicht möglich, jedoch kann man den Level der Zugriffsmöglichkeiten auf die I/O-Ports von 0 (keine Erlaubnis) auf 3 (voller Zugriff) einstufen.

6.1.2. Einlesen/Ausgeben

Nachdem die Zugangsbedingungen erfüllt sind, kann man mit dem eigentlichen Ausgeben bzw. Auslesen anfangen. Jedoch sind die Hardware-Komponenten des Computers nicht wie das RAM über „normale“ Speicherzugriffe zugänglich, sondern nur mit Hilfe spezieller (Assembler-)Befehle erreichbar. Für C-Programme werden eine Reihe von Makros zur Verfügung gestellt, die diese Befehle auf C-Funktionen abbilden. Diese Funktionen sind Inline-Makros: d.h. #include <asm/io.h> reicht vollkommen aus, um sie einzubinden. Zudem benötigt man keine zusätzlichen Bibliotheken. Tabelle 2.1 gibt eine Übersicht dieser Funktionen.

Um nun ein Byte (8 Bits) zu einem I/O-Port auszugeben, muss man die Funktion outb(Port, Wert) in der richtigen Reihenfolge der Parameter (z.B. unter DOS sind sie in umgekehrter Reihenfolge einzugeben) aufrufen. Um umgekehrt von einem Port einzulesen, kommt die Funktion unsigned char inb(unsigned short int from) zum Einsatz. Sie gibt das Byte zurück, das sie empfangen hat. Die meisten Geräte sind für byteweisen Portzugang konzipiert. Übrigens alle Portkommunikations-anweisungen benötigen zum Ausführen mindestens ungefähr eine Mikrosekunde. Die mit dem Suffix „_p“ versehenen Makros arbeiten fast identisch zu den anderen Makros, aber sie gewähren eine zusätzliche kurze Verzögerung (ungefähr eine Mikrosekunde) nach dem Portzugang. Man kann die Verzögerung von ungefähr vier Mikrosekunden mit #define REALLY_SLOW_IO vor dem #include <asm/io.h>

erzwingen. Diese Makros (es sei denn man verwendet #define SL0W_I0_BY_JUMPING, das vermutlich weniger genau ist) benutzen normalerweise einen Portausgang zu 0x80 für ihre Verzögerung, also muss man zuerst Zugang zum Port 0x80 mit ioperm() verleihen. Ausgänge zum Port 0x80 sollten keinen Teil des Systems beeinflussen.

Tabelle 6- 1: Makros für den Zugriff auf I/O-Ports.

Makro	Beschreibung
inb (Adresse) inw (Adresse) inl (Adresse) insb (Adrese) insw (Adresse) insl (Adresse)	Der Inhalt des I/O-Bereichs an der Adresse Adresse wird ausgelesen. Dabei wird je nach Endung des Makros ein Byte (b), Word (w) oder Long-Word (1) zurückgegeben, wahlweise auch mit Vorzeichen
outb (Wert, Adresse) outl (Wert, Adresse) outsb (Wert, Adresse) outsw (Wert, Adresse) outw (Wert, Adresse) outsl (Wert, Adresse)	Der Wert Wert wird in den I/O-Bereich an die Adresse Adresse geschrieben. Dabei wird Wert je nach Endung des Makros als Byte (b), Word (u) oder Long-Word (1) interpretiert, wahlweise auch mit Vorzeichen
inb_p(Wert, Adresse) inw_p(Wert, Adresse) inl_p(Wert, Adresse) outb_p(Wert, Adresse) outw_p(Wert, Adresse) Outl_p(Wert, Adresse)	Im Unterschied zu den Makros ohne ,,_p“ wird bei diesen Funktionen die Ausführung etwas verzögert, um „langsamer“ Hardware Rechnung zu tragen

Wegen einer Beschränkung im GNU C-Compiler (vorhanden in allen Versionen, einschließlich der egcs), muss jedes (mögliche) Quellprogramm mit der Option -O für die Optimierung kompiliert werden (gcc - O1 oder höher). Als Alternative kann #define extern Static eingefügt werden, bevor #include <asm/io.h> eingebunden wird (letztendlich darf die Anweisung #undef extern gleich danach nicht vergessen werden).

Für das Debuggen kann gcc mit dem Kommando - g - 0 aufgerufen werden (auf jeden Fall in modernen Versionen von gcc), obwohl sich die Optimierung den Debugger manchmal merkwürdig benehmen lässt. Man kann dies umgehen, indem das Programm, das den I/O-Portzugang verwendet, in einer separaten Quelldatei gespeichert und nur für sich mit der Optimierung kompiliert wird.

Zu Begin der Programmierung konnten die Daten über die serielle Schnittstelle nicht richtig, teilweise überhaupt nicht, versendet und empfangen werden. Dann sollte das Verbindungskabel getestet werden, indem anhand Standardkommunikations-programme versucht wurde, Daten über das Nullmodemkabel auszutauschen. Die Standardkommunikationsprogramme sind zum einen „Hyperterminal“ unter Windows und zum anderen „minicom“ unter Linux. Wenn man zwei Rechner mit Windows-Betriebssystem bzw. Linux über die serielle Schnittstelle verbindet und Hyperterminal bzw. minicom startet, dann hat man Initialisierungseinstellungen vorzunehmen. D.h. baud rate, Datenlänge, Paritätsart, Stopbit und Datenflusskontrolle mit Werten zu versehen, nachdem der richtige COM-Port ausgewählt wurde. Die baud rate erhält Werte zwischen 110 und 921600 Bits pro Sekunde, die Flussteuerung kann entweder hardwaremäßig, softwaremäßig oder aber nicht vorhanden sein. Tatsächlich war das Kabel nicht mehr in Ordnung und sollte ausgetauscht werden. Und zwar lag es daran, dass RS-232 kein USB-Anschluss ist. Was bedeutet, dass das Kabel während des Betriebes nicht umgeschaltet werden darf, also erst wenn der Rechner ausgeschaltet ist, darf man das Kabel von der seriellen Schnittstelle entfernen. Mit dem neuen Kabel konnten die Standardkommunikationsprogramme problemlos Daten versenden und empfangen, sogar „Hyperterminal“ konnte mit „minicom“ Daten austauschen. Man muss nur auf die Initialisierung achten, z.B. Linux unterstützt keine 1,5 Stopbits und wenn die baud rate auf 115200 gesetzt wird, dann werden die Daten von Hyperterminal zu minicom richtig gesendet, jedoch in die andere Richtung läuft das nicht. Bei 6 Datenbits werden nur Zahlen richtig übertragen, die Zeichen (Buchstaben) leider nicht. Die Daten werden zeichenweise übertragen, was auf eine nicht-kanonische und nicht blockierende Kommunikation hinweist.

6.2. Zugriff über Gerätedateien

Die andere Methode I/O-Ports zugänglich zu machen, besteht darin, das „Gerät“ /dev/port zu öffnen – ähnlich wie normale Dateien entweder zum Lesen, zum Schreiben oder gleichzeitig zu beidem. Die stdio-Funktionen f*(), die normalerweise für Blockgeräte (block device) gedacht sind, können hier auch verwendet werden, sind aber wegen einer internen Pufferung zu vermeiden. Dafür stehen die Gerätedateifunktionen wie z.B. open() zur Verfügung, da sich die Ports unter Zeichengeräte (charactere device) gliedern lassen.

Natürlich müssen hier auch Lese-/Schreibrechte auf /dev/port vorhanden sein. Und zwar wird diese Bedingung erfüllt, indem der Zugang zu /dev/port freigegeben wird. Das heißt, dass diese Methode weder Compileroptimierung noch ioperm() und (sogar) keine „root“-Zugriffsrechte benötigt. Aber diese Art der Portfreigabe ist in Bezug auf Systemsicherheit nicht empfehlenswert, da so eine Systemverletzung möglich wird. Es kann sogar Zugang zum root „erschlichen“ werden, indem man /dev/port verwendet, um Festplatten, Netzwerkkarten, etc. direkt zugänglich macht.

Diese Methode ist viel umfangreicher und findet häufige Verwendung in der Entwicklung (jeglichen Hardware-Zugriff) von hardware-basierten Zugriffen. Sie bietet auch viel mehr Möglichkeiten in der Handhabung von Threads und Echtzeit. Es ist nicht möglich select (s. man 2 select) oder poll(man 2 poll) zu verwenden, um /dev/port zu lesen, weil die Hardware keine Möglichkeiten zum Merken der Stati der CPU hat, wenn sich ein Wert in einem Eingangsport ändert.

6.2.1. Ports öffnen und schließen

Um Daten über eine serielle Schnittstelle senden oder empfangen zu können, muss man diese zuerst öffnen. Dazu steht die Funktion int open(const char *pathname, int openflag /*,mode_t mode */) zur Verfügung. Mit dieser Funktion legt man fest, ob von dieser (bzw. zu dieser) Schnittstelle nur gelesen bzw. geschrieben wird oder beides gleichzeitig. Dies geschieht mit den flags: O_RDONLY, O_WRONLY oder O_RDWR.

In diesem Fall muss einmal gelesen werden (der Board-Computer soll die Sonnenstrahlstärke, die Geschwindigkeit, die Richtung, die Temperatur, die Winkelgeschwindigkeit und die Azimutgeschwindigkeit von der Sensorik empfangen, auslesen), einmal gleichzeitig lesen und schreiben (der Transceiver übergibt die Werte dem Board-Computer und liest die gewünschten Werte von der Sensorik aus). Abschließend werden die vom Board-Computer gelesenen Werte der Aktorik mitgeteilt, d.h. es wird nur geschrieben.

Das folgende Codebeispiel zeigt diesen Schritt mit der open()-Funktion:

```
#include <sys/types.h>      // Definition der primitiven Systemdatentypen
#include <sys/stat.h>       // Dateistatus
#include <fcntl.h>          // Elementare E-/A-Operationen
#include <errno.h>          // Fehlerkonstantendefinition mit Konstantennummer
#include <string.h>         // Makros und Funktionen zur  Zeichenkettenbearbeitung

int fd;

fd = open("/dev/ttyS0", O_WRONLY);

if (fd < 0)
        printf("Fehler beim Öffnen von COM1, %d, %s", errno, perror(errno));
else
        printf("COM1 wurde erfolgreich geöffnet. \n");
```

Diese Funktion gibt den file descriptor fd bei Erfolg zurück, im anderen Falle gibt die Funktion den Wert -1 zurück.

Es ist zu beachten, dass dieser Code die serielle Schnittstelle COM1 zum Schreiben öffnet. Geschrieben wird jedoch erst mit der Funktion „write“. Diese Funktion wird im Folgenden genauer erläutert.

Weitere für diese Aufgabe wichtige, aber optionale Konstanten sind die folgenden:
O_NOCTTY: Der geöffnete Terminal sollte nicht der Kontrollterminal des Prozesses werden.

O_NONBLOCK: Die geöffnete serielle Schnittstelle wird bei nachfolgenden E/A-Operationen nicht blockiert.

O_ASYNC: Asynchrone Ein-/Ausgabe, sie generiert ein Signal (SIGIO)

O_SYNC: Nach jedem Schreiben mit der Funktion write() ist darauf zu warten, dass der Schreibvorgang vollständig abgeschlossen ist. Dies bewirkt, dass jeder Aufruf der Funktion write() erst alle Daten vollständig auf das physikalische Medium schreibt, bevor andere Programmschritte möglich sind.

Nachdem die Kommunikation beendet worden ist, muss man die serielle Schnittstelle wieder für andere Anwendungen bzw. Benutzer freigeben. Dies geschieht, indem man die serielle Schnittstelle mit der Funktion int close(int fd) schließt. Diese Funktion gibt -1 zurück, wenn sich der Port nicht schließen lässt, sonst gibt sie 0 bei Erfolg zurück. Hierzu ein kleines Beispiel:

```
#include <unistd.h>        /* Reduzierte UNIX Standard Bibliothek & symbolische
                           Konstanten */
#include <errno.h>         // Fehlerkonstantendefinition mit Konstantennummer
#include <string.h>        // Makros und Funktionen zur  Zeichenkettenbearbeitung

if (( close(fd)) < 0)
        printf(„konnte den Port nicht schließen!\n %d, %s“, errno, perror(errno));
```

Im Rahmen der vorliegenden Arbeit wurde die serielle Schnittstelle zunächst vorsichtshalber geschlossen, um sie dann erneut zu öffnen, da sonst die eingehenden Daten nicht richtig und vollständig empfangen worden wären. Am Ende des Programms darf die serielle Schnittstelle nicht geschlossen werden, da das Programm endlos Daten versenden und empfangen soll.

6.2.2. Kanonische und nicht-kanonische Eingabe/Ausgabe

Die Funktion ssize_t write(int fd, const void *buffer, size_t nbytes) gibt die Anzahl der tatsächlich geschriebenen Bytes bei Erfolg zurück. Übrigens ist diese Anzahl ist nicht unbedingt gleich der Variable „nbytes“, sie kann kleiner sein, was nicht auf Fehler hindeuten soll, sondern nur, dass zur Zeit keine anderen Daten eingetroffen sind. Bei

Fehler gibt diese Funktion -1 zurück. Unter POSIX.1 entspricht ssize_t signed integer und entspricht size_t unsigned integer, was unsere write-Funktion in int write(int fd, char *buffer, int nbytes) umwandelt.

Codebeispiel:

```
#include <stdio.h>          // Standard Ein-/Ausgabefunktionen
#include <unistd.h>         /* Reduzierte UNIX Standard Bibliothek & symbolische
                            Konstanten */

int fd, iBuf;
char buffer[]=”nun die Ausgabe testen”;

if ((iBuf = (write(fd, buffer, sizeof(buffer)) )) < 0)
        printf (“konnte auf fd nicht schreiben!”);
else if (iBuf == sizeof(buffer))
        printf (“Alle Zeichen wurden erfolgreich geschrieben!”);
else
        printf(“konnte nur %d Zeichen schreiben!”, iBuf);
```

Unterschieden werden können dabei zwei Möglichkeiten der Kommunikation: zum einen der kanonische Modus und zum anderen der nicht-kanonische Modus.

Der kanonische Modus bedeutet, dass die ganze Zeile auf einmal gesendet (Schreiben) oder empfangen (Lesen) wird. Bei der Arbeit mit normalen Dateien spricht man von gepufferter Ein-/Ausgabe. Die Terminals, auf die zurückgegriffen wird, sind standardmäßig im kanonischen Modus vorkonfiguriert. Beim Schreiben auf den Port werden die Zeichen zunächst nur in den Puffer abgelegt, bis entweder dieser voll oder die Zeile zu Ende ist. D.h. erst dann wird der Inhalt des Puffers tatsächlich auf das physikalische Medium übertragen, wenn ein Linefeed LF(ASCII), New line NL oder Carriage retrun CR auftritt. Beim Schließen des Ports wird der Rest des Puffer-Inhalts erst automatisch geschrieben, dann der Speicher für den Puffer freigegeben. In diesem Modus muss man das flag ICANON in der lokalen Konstante c_lflag setzen. Man nennt den kanonischen Modus auch blockierend, weil eine read-

/write-Operation der Anwendung die Kontrolle erst zurückgibt, wenn eine Textzeile eingegeben worden ist.

Im Rahmen dieser Arbeit wurde dieses Codebeispiel geschrieben, um den kanonischen Modus einzusetzen:

```
#include <stdio.h>          // Standard Ein-/Ausgabefunktionen
#include <stdlib.h>         /* Reduzierte Standard C Bibliothek & allgemein
                            nützliche Funktionen */
#include <unistd.h>         /* Reduzierte UNIX Standard Bibliothek & symbolische
                            Konstanten */
#include <string.h>         // Makros und Funktionen zur
Zeichenkettenbearbeitung
#include <fcntl.h>          // Elementare E-/A-Operationen
#include <termios.h>        // POSIX Terminal Funktionen und Konstanten
#include <sys/types.h>      // Definitionen der primitiven Systemdatentypen
#include <sys/stat.h>       // Dateistatus
#include <errno.h>          // Fehlerkonstantendefinition mit Konstantennummer

#define FALSE 0
#define TRUE 1

int STOP = FALSE;

void init_port(int fd)
{
struct termios tio;
```

```
        tcgetattr(fd, &tio);
        tio.c_cflag = 0;
        tio.c_iflag = 0;
        tio.c_oflag = 0;
        tio.c_lflag = 0;

        tio.c_cflag |= B38400;
        tio.c_cflag |= CRTSCTS;
        tio.c_cflag |= CS8;
        tio.c_cflag |= (CLOCAL | CREAD);

        tio.c_iflag = IGNPAR | ICRNL;
        tio.c_oflag = 0;
        tio.c_lflag = ICANON;

        tcflush(fd, TCIFLUSH);
        tcsetattr(fd, TCSANOW, &tio);
}

int main()
{
        int fd, iOut;
        char buffer[] = „Mal etwas senden!?!“;

        fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY);
        if (fd < 0)
        {
                perror("/dev/ttyS0");
                exit(-1);
        }

        init_port(fd);
```

```
        while (STOP==FALSE)
        {
                iOut = write(fd, buffer, sizeof(buffer));
                if (iOut < 0)
                {
                        printf(„Kann auf COM0 nicht schreiben: %d, %s\n“, errno,
strerror(errno));
                        exit(-1);
                }

                buffer[iOut] = 0;
                printf("geschrieben wurden %d und ganz genau %s\n", iOut, buffer);
                if (buffer[0] == 'q') STOP = TRUE;
        }

        close(fd);
        return 0;
}
```

Ein Terminal hat 5 verschiedene Modi-Eigenschaften: Eingangsmodus (iflag), Ausgangsmodus (oflag), Kontrollmodus (cflag), Lokalmodus (lflag) und Steuerzeichen (cc). Jeder diese Modi verwaltet eine große Anzahl an symbolischen Konstanten wie die folgende Tabelle zeigt.

Tabelle 6- 2: Die fünf verschiedenen Modi-Eigenschaften eines Terminals.

Komponente	Beschreibung
c_iflag	Eingabeflag
BRKINT	Generieren von SIGINT bei break
IGNBRK / IGNPAR	Ignorieren von break/ Paritätsfehlern
INPCK	Paritätsprüfung erlauben
IGNCR	Ignorieren von Carriage Return
ICRNL	Umwandeln von CR in NL bei der Eingabe
INLCR	Einschalten der Paritätsprüfung bei der Eingabe
IXON / IXOFF	Datenfluss-Steuerung softwaremäßig ermöglichen
PARMARK	Markieren von Paritätsfehlern
c_oflag	Ausgabeflag
OPOST	Ermöglichen einer implementierungsdefinierten Ausgabeart. Wenn diese Option gesperrt ist, dann werden alle anderen Optionen in c_oflag ignoriert.
ONLCR	NL in CR-NL Paare abbilden.
c_cflag	Kontrollflag
B0-B115200	Baudrate des Terminals festlegen.
CLOCAL	Ignorieren der Modemstatuszeichen bzw. den Terminal nur lokal betreiben.
CREAD	Zeichen können empfangen werden.
CRTSCTS	Datenflusskontrolle hardwaremäßig aktivieren
CSIZE	Bitanzahl pro Zeichen: CS5-8 für 5 bis 8 Bits pro Byte
CSTOP	Zwei Stoppbits verwenden. Standardmäßig ein Stoppbit.
PARENB	Paritätserzeugung für die Ausgabe und Paritätsprüfung für die Eingabe ermöglichen.
PARODD	Ungerade Parität aktivieren, sonst ist gerade voreingestellt.
c_lflag	Lokale flags
ECHO	Jedes eingegebene Zeichen auch auf dem Terminal ausgeben.
ICANON	Den kanonischen (zeilenorientierten) Modus einschalten.
ISIG	Entsprechendes Signal generieren, wenn Terminalsteuerzeichen eingetreten (eingetroffen) sind.

c_cc[NCCS]	Steuerzeichen
VMIN	Die minimale zu lesende Byteanzahl bevor eine Leseoperation zurückkehrt.
VTIME	Die minimale Zeit, die gewartet werden soll, bis eine Leseoperation zurückkehrt.

Die Funktion tcgetattr(int fd, struct termios *tio) ermöglicht es, die in tio gespeicherten Terminalattribute zu ermitteln. Die Funktion tcsetattr(int fd, int flag, const struct termios *tio) dagegen führt die vorher gesetzten flags aus, wobei fd für ein file descriptor stehen soll. Wenn beispielsweise die symbolische Konstante TSCANOW verwendet wird, heißt dies, dass die Änderungen sofort aktiviert werden müssen. Dagegen hat TCSAFLUSH zur Folge, dass die Änderungen erst zu aktivieren sind, nachdem alle anstehenden Ausgaben übertragen wurden.
Die Funktion tcflush(fd, flag) entleert den Ein-/Ausgabepuffer der mit fd geöffneten Gerätedatei. Dieser Funktion stehen wiederum verschiedene symbolische Konstanten zur Verfügung: TCIFLUSH, TCOFLUSH und TCIOFLUSH. Sie haben die Funktion, dass entweder jeweils Ein-, Aus- oder aber Ein- und Ausgabepuffer geleert werden. Die dabei noch in den jeweiligen Puffern befindlichen Daten werden ohne Bearbeitung gelöscht.

Die Funktion ssize_t read(int fd, void *buffer, size_t nbytes) zum Lesen ähnelt der Funktion write(). Wenn man die Headerdatei <unistd.h> einbindet, dann ist sie unter POSIX.1 folgendermaßen definiert int read(int fd, char *buffer, int nbytes) und gibt die Anzahl der gelesenen Bytes bei Erfolg zurück, den Wert 0 bei EOF. Ansonsten wird der Wert -1 zurückgegeben.
Beim Lesen aus einem Port (Zeichengerät) mit den Funktionen read() oder pread() ist die Anzahl der tatsächlich gelesenen Zeichen häufig geringer als die mit nbytes angegebene Zahl. Dieses liegt lediglich daran, dass im Moment noch keine weiteren Zeichen verfügbar sind [2].

Beim nicht-kanonischen Modus dagegen werden die Eingangszeichen unverzüglich ausgeführt, d.h. ohne Zwischenspeicherung sofort weitergeleitet. Hierbei wird entweder eine bestimmte Anzahl von Bytes oder nach einer gewissen Zeit die Bytes ausgegeben, die eingetroffen sind. Für diesen Zweck steht das Array c_cc[] (s. Tabelle 6-3) der Struktur termios zur Verfügung.

Tabelle 6- 3: Mögliche Steuerzeichenvarianten für die nicht-kanonische Eingabe.

	MIN > 0	MIN == 0
TIME > 0	Die Leseoperation liefert mindestens MIN Bytes, wenn TIME noch nicht abgelaufen ist. Sie liefert 1 oder MIN Bytes, wenn TIME abgelaufen ist. Die Schaltuhr wird erst beim ersten gelesen Byte gestartet. Deshalb kann hier eine unendliche Blockierung erzielt werden.	Die Leseoperation liefert 1 bzw. die Anzahl benötigter Bytes, wenn TIME noch nicht abgelaufen ist. Ansonsten liefert sie 0 zurück. Die Zeitschaltuhr wird zu Beginn der Leseoperation gestartet
TIME == 0	Die Leseoperation liefert MIN Bytes, wenn diese vorhanden sind. Blockierung auch möglich, wenn keine MIN Bytes geliefert werden.	Die Leseoperation liefert 0 oder die Anzahl geforderter Bytes. Sie kehrt auf jeden Fall ohne jegliches Warten sofort zurück.

Nicht-kanonischer Modus bedeutet auch, dass eine Anwendung andere Aufgaben durchführen kann, solange sie auf I/O wartet. Aus diesem Grund wird diese Verarbeitungsart auch nicht-blockierend genannt.

Das folgende Beispiel veranschaulicht einen Teil des im Rahmen dieser Arbeit entwickelten Codes im nicht-kanonischen Modus:

```
#include <stdio.h>          // Standard Ein-/Ausgabefunktionen
#include <stdlib.h>         /* Reduzierte Standard C Bibliothek & allgemein
                               nützliche Funktionen */
#include <unistd.h>         /* Reduzierte UNIX Standard Bibliothek & symbolische
                               Konstanten */
#include <string.h>         // Makros und Funktionen zur  Zeichenkettenbearbeitung
#include <fcntl.h>          // Elementare E-/A-Operationen
#include <termios.h>        // POSIX Terminal Funktionen und Konstanten
#include <sys/types.h>      // Definition der primitiven Systemdatentypen
#include <sys/stat.h>       // Dateistatus
#include <errno.h>          // Fehlerkonstantendefinition mit Konstantennummer
```

```
struct coordinates   {
        short speed;
        short direction;
        short temperature;
        };

void init_port(int fd)
{
        struct termios tio;

        tcgetattr(fd, &tio);
        bzero(&tio, sizeof(tio));

        cfsetospeed (&tio, B38400);

        tio.c_cflag |= PARENB;
        tio.c_cflag &= ~PARODD;
        tio.c_cflag &= ~CSTOP;
        tio.c_cflag &= ~CSIZE;
        tio.c_cflag |= CRTSCTS;
        tio.c_cflag |= CS8;
        tio.c_cflag |= (CLOCAL | CREAD);

        tio.c_iflag = IGNPAR;
        tio.c_oflag = 0;
        tio.c_lflag = 0;
        tio.c_cc[VMIN] = 5;
        tio.c_cc[VTIME] = 0;

        tcflush(fd, TCOFLUSH);
        tcsetattr(fd, TCSANOW, &tio);
}
```

```
int main()
{
	int fd, iIn;
	struct coordinates *Koordinaten;
	char buffer[255] ;

	fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY);
	if (fd < 0)
	{
		fprintf(stderr, "Kann COM0 nicht öffnen !\n");
		exit(-1);
	}
	else
	{
		init_port(fd);

		iIn = write(fd, buffer, sizeof(buffer));
		if (iIn < 0)
		{
	printf(„Kann auf COM0 nicht schreiben: %d, %s\n“, errno, strerror(errno));
	exit(-1);
		}

		buffer[iIn] = 0;
		printf("gelesen wurden %d und ganz genau %s\n", iIn, buffer);
	}

	close(fd);
	return 0;
}
```

Die Funktion bzero() setzt die ersten n (sizeof()) Bytes ab dem Anfang des Terminals tio auf null.

POSIX.1 bietet die folgenden zusammengefassten Funktionen zum Erfragen und/oder Ändern der Terminaleigenschaften: tcgetattr, tcsetattr, cfgeti(o)speed, cfset(i)ospeed und tcflush.

Tabelle 6- 4: POSIX-Funktionen zum Abfragen und Ändern der Terminalattribute.

Funktion	Beschreibung
tcgetattr	Erfragen der Terminalattribute
tcsetattr	Setzen der Terminalattribute
cfgetispeed	Erfragen der Eingabegeschwindigkeit
cfgetospeed	Erfragen der Ausgabegeschwindigkeit
cfsetispeed	Setzen der Eingabegeschwindigkeit
cfsetospeed	Setzen der Ausgabegeschwindigkeit
tcflush	Leeren der Ein- und/oder Ausgabepuffer

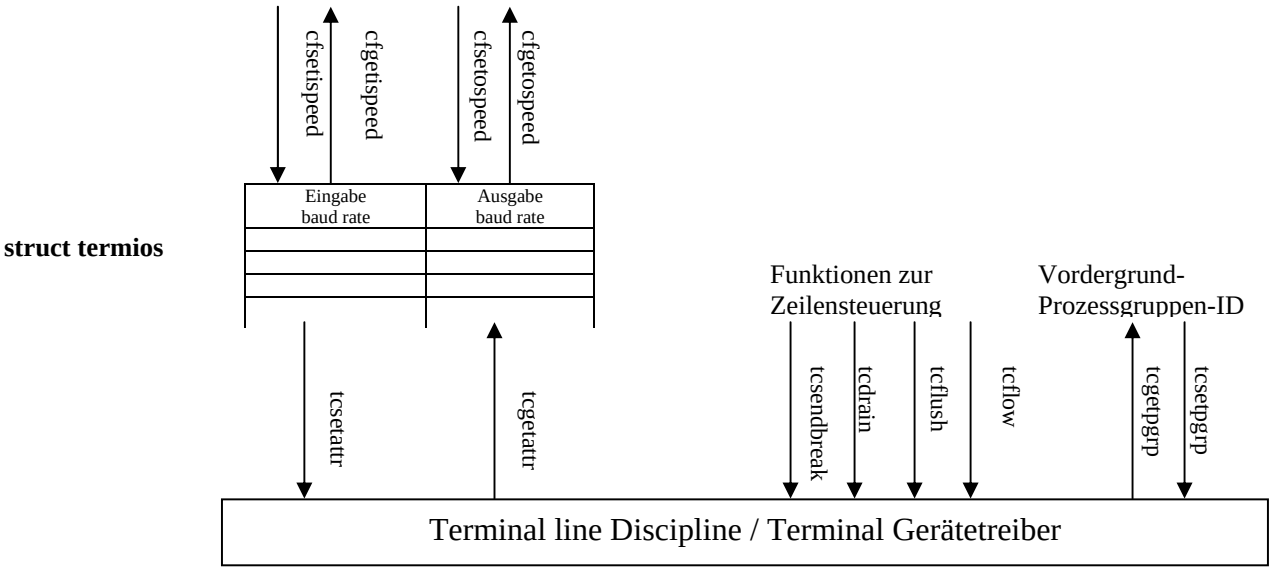


Abbildung 6- 1: Terminal E/A-Funktionen im Überblick.

Die Funktion cfgeti(o)speed erfragt die Ein-/Ausgabegeschwindigkeit. Sie darf aber erst aufgerufen werden, nachdem die termios-Struktur mit der Funktion tcgetattr() ermittelt worden ist.
Die Funktion cfsetospeed setzt die baud rate in diesem Fall auf den Wert B38400. Hier gilt auch, dass die mit cfseti(o)speed-Funktionen gesetzten Änderungen erst eingestellt werden, nachdem die Funktion tcsetattr() aufgerufen wird.

6.2.3. Synchrone und asynchrone Übertragung

Der kanonische bzw. nicht-kanonische Modus kann sowohl synchron als auch asynchron programmiert werden.
Damit der Computer die eingehenden Seriendaten versteht, benötigt er einen Hinweis darauf, wo ein Zeichen endet und das folgende beginnt. Im asynchronen Modus bleibt die Seriendatenleitung im high-Zustand (1) bis ein Zeichen übertragen ist. Ein Startbit geht jedem Zeichen voran und wird unverzüglich von jedem Bit im Zeichen gefolgt – optional von einem Paritätsbit und mindestens von einem oder zwei Stopbits.
Das Startbit ist immer ein low-Signal (0) und teilt dem Computer mit, dass neue Seriendaten vorhanden sind. Daten können jederzeit gesendet oder empfangen werden.
Die Dauer eines einzelnen Bits wird durch einen Taktgeber im Sender bestimmt, indem mit dem nächsten Taktimpuls das nächste Bit ausgegeben wird. Im Empfänger wird das hereinkommende Bitmuster mit einem zum Sender asynchronen Takt abgetastet. Wenn der Empfänger auf das nächste Zeichen wartet, erkennt er bei dieser Abtastung die 1-0-Flanke des Startbits und weiß, dass jetzt ein neues Zeichen kommt. Zwischen Sender und Empfänger müssen die Anzahl der Bits pro Zeichen und mit der Übertragungsrate auch die Breite eines Bits schon bei der Herstellung der Verbindung fest vereinbart werden.
Der Empfänger weiß dann, was er zu erwarten hat, und kann so immer etwa in der Bitmitte abtasten und die Werte der einzelnen Bits eines Zeichens ermitteln. Das optionale Paritätsbit ist eine einfache Summe der Informationsbits, die anzeigen, ob die Daten eine gerade oder ungerade Anzahl von 1-Bits enthalten. Mit gerader Parität ist das Paritätsbit 0, wenn es eine gerade Zahl von Einsen im Zeichen gibt. Mit ungerader Parität ist das Paritätsbit 0, wenn es eine ungerade Zahl von Einsen in

den Daten gibt. Darüber hinaus gibt es noch drei weitere Paritätsvarianten: Raumparität (space parity), wobei das Paritätsbit immer 0 ist. Die zweite wird Markierungsparität (mark parity) genannt, bei ihr ist das Paritätsbit immer 1. Die letzte Variante (no parity) setzt kein Paritätsbit.

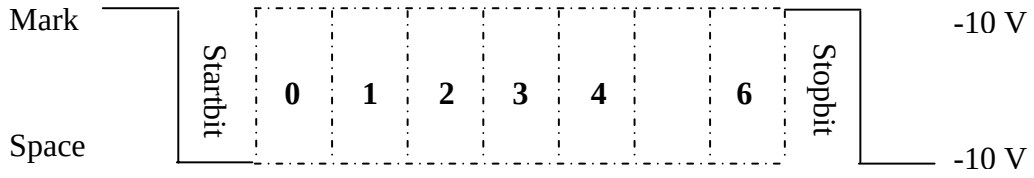


Abbildung 6- 2: Asynchrone Datenübertragung mit 7 bits Datenlänge.

Es kann 1 oder 2 Stoppbits zwischen den Zeichen geben und sie haben immer einen Wert von 1 (high-Signal). Bei dieser Übertragungsart ist zu bedenken, dass für acht Datenbits jeweils drei Bits hinzugefügt werden und dass damit natürlich die Übertragungseffizienz reduziert wird. Die Zeit zum Übertragen der „eingepackten“ acht Bits ist um 3/8 länger als die Zeit, die man für „nackte“ acht Bits gebraucht hätte. Diese Zeit ist für die Übertragung anderer Daten verloren. Bei längeren Telegrammen bietet sich deshalb eine synchrone Übertragung an. Beim asynchronen Transfer (kehrt) wird das Auslesen (unverzüglich) sofort zurückgegeben und sendet ein Signal zum aufrufenden Programm. Folgendes eine Anwendung der nicht-synchronen Kommunikation:

```
#include <termios.h>
#include <stdlib.h>      /* Reduzierte Standard C Bibliothek & allgemein nützliche
                            Funktionen */
#include <stdio.h>       // Standard Ein-/Ausgabefunktionen
#include <unistd.h>      // Reduzierte UNIX Standard Bibliothek & symbolische Konstanten
#include <fcntl.h>       // Elementare E-/A-Operationen
#include <sys/signal.h>  // Makros und Funktionen um Signale abzufangen
#include <errno.h>       // Fehlerkonstantendefinition mit Konstantennummer
#include <string.h>      // Makros und Funktionen zur  Zeichenkettenbearbeitung

#define FALSE 0
#define TRUE 1

int STOP=FALSE;
```

```
int RECEIVED=TRUE;

void signal_handler (int s)
{
   printf("SIGIO empfangen!\n");
   RECEIVED = FALSE;
}

int main(int argc, char *argv[])
{
   int fd, iIn, i, key;
   char cIn, message[90];
   struct termios oldtio, newtio;
   struct sigaction sigio;
   char buffer[255];

   if (argc != 2)
   {
        printf("Bitte Portnummer mitangeben !\n");
        exit (-1) ;
   }
   else
   {
      fd = open(argv[1], O_RDWR | O_NOCTTY | O_NONBLOCK);
      if (fd < 0)
      {
         printf("Fehler beim Öffnen: %d %s\n", errno, strerror(errno));
         exit(-1);
      }

      sigio.sa_handler = signal_handler;
      sigemptyset(&sigio.sa_mask);
      sigio.sa_flags = 0;
      sigio.sa_restorer = NULL;
```

```
sigaction(SIGIO, &sigio, NULL);

    fcntl(fd, F_SETOWN, getpid());
    fcntl(fd, F_SETFL, FASYNC);

    tcgetattr(fd, &oldtio);
    newtio.c_cflag = B38400 | CRTSCTS | CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    newtio.c_lflag = 0;
    newtio.c_cc[VMIN]=1;
    newtio.c_cc[VTIME]=0;

    tcflush(fd, TCIFLUSH);
    tcsetattr(fd, TCSANOW, &newtio);

    tcsetattr(1, TCSANOW, &newtio);

    while (STOP == FALSE)
    {
      if ((key = getc(STDIN_FILENO)) == TRUE)
      {
        switch (key)
        {
          case 0x1b: /* Esc */
            STOP = TRUE;
            break;
          default:
            write(fd, &key, 1);
            break;
        }
      }
      if (RECEIVED == FALSE)
      {
```

```
          iIn = read(fd, buffer, 255);
          if (iIn <= 0)
          {
              printf("Fehler beim Öffnen: %d %s\n", errno, strerror(errno));
              exit (-1);
          }
          else
          {
            for (i=0; i<iIn; i++)  //for all chars in string
            {
               cIn = buffer[i];
               if ((cIn<32) || (cIn>125))
               {
                   sprintf(message, "%x", cIn);
                   fputs(message, STDOUT_FILENO);
               }
               else fputc ((int) cIn, STDOUT_FILENO);
            }
          }

          RECEIVED = TRUE;
        }
      }
      tcsetattr(fd, TCSANOW, &oldtio);
      close(fd);
   }
return 0;
}
```

Standardmäßig ist die Übertragung synchron (auch als blockierend bezeichnet), d.h. mit dem Auslesen wird solange gewartet bis Daten zum Ausgeben bzw. Auslesen vorliegen. Anders als die asynchronen Daten erscheinen die synchronen Daten als konstanter Bitstrom. Um die Daten auf der Datenleitung zu lesen, muss der Computer einen gemeinsamen Bittaktgeber zur Verfügung stellen oder gestellt bekommen, damit der Absender und der Empfänger synchronisiert werden. Da synchrone Protokolle kein zeichenweises Synchronisierungsbit benutzen, liefern sie für gewöhnlich mindestens eine 25%-ige Verbesserung gegenüber der asynchronen Kommunikation und sind für Verbindungen mit mehr als zwei seriellen Schnittstellen besser geeignet.

Trotz der Geschwindigkeitsvorteile der synchronen Kommunikationsmethode unterstützen die meisten RS-232 diese Vorgehensweise nicht wegen des hohen Aufwands an Hard- und Software.
Charakteristisch für die synchrone Datenübertragung ist, dass der Empfänger dieselbe Taktfrequenz wie der Sender benutzt und bei jedem Telegramm eine Synchronisation seines Taktes mit dem des Senders durchführt, um so korrekt abtasten zu können.

6.3. Threadprogrammierbeispiel

Die Threads-programmierung geschieht zu einem mit den POSIX-Threads. Die zweite Möglichkeit ist die Programmierung mit Hilfe der Common API von RTLinux. Jede dieser Varianten hat Vor- und Nachteile. Das RTLinux-API ist leistungsfähiger als POSIX, hat jedoch den Nachteil, dass es sich nicht in anderen Betriebssystemen einsetzen lässt.

Es ist noch zu erwähnen, dass das System abstürzen kann, z.B. wenn der Echtzeitprozess die gesamten Prozessorressourcen auslastet, d.h. keine Zeit mehr zum Ausführen von Linux (User Space) hat. Allgemein ist die Echtzeitprogrammierung (Kernel Space) mit einer sehr großen Sorgfalt zu verrichten, da sie keine Sicherheit zur Systemintegrität anbietet.
Wenn man Threads kreieren möchte, dann steht einem die Funktion
int pthread_create(pthread_t *threadID, pthread_attr_t *attr, void *(*threadFct)(void *),

void *arg), diese Funktion gibt 0 bei Erfolg zurück oder gibt einen Exxx-Wert im anderen Fall zurück. Beim Kreieren eines Threads wird ihm eine eindeutige Thread-ID zugewiesen, die man mit der Funktion pthread_t pthread_self(void) erfragt werden kann. Die Threads sind im Standard POSIX.1b spezifiziert und unter der Herder-Datei pthread.h (vorbereitet).

```
#include <pthread.h>

struct thread_data {
        short speed;
        short direction;
        short temperature;
        };

void *lesen(void *data);
void *schreiben(void *data);

int main(void)
{
        pthread_t Id1, Id2;
        struct thread_data data1, data2;

        data1.speed = “123.4”;
        data1.direction = “35,64”;
        data1.temperature = 40;
        pthread_create(&Id1, NULL, &schreben, &data1);

        pthread_create(&Id2, NULL, &lesen; &data2);

        pthread_join(Id1, NULL);
        pthread_join(Id2, NULL);

        return 0;
}

void *schreiben(void *data)
{
        int fd, iIn;
        char buffer[255];

        fd = open(“/dev/ttyS0”, O_RDWR | O_NONBLOCK);
        if (fd < 0) Fehlermeldung;
        iIn = write(fd, buffer, 255);
        if (iIn < 0) Fehlermeldung;

        buffer[iIn] = 0;
        fprintf(stderr, “geschrieben wurden %d Zeichen \n”, iIn);
        return 0;
}
```

In der vorliegenden Arbeit müssen Parameter dem Thread von der seriellen Schnittstelle übergeben werden, dazu steht der vierte Parameter arg in der Funktion pthread_create() zur Verfügung.

Die Funktion pthread_join(pthread_t Id, void **retval) suspendiert die Ausführung des ausrufenden Thread, bis der Thread mit der Kennung Id beendet ist.
RT-Linux besteht im Wesentlichen aus den folgenden Modulen [?]:

- rtl_sched.o: beinhaltet den zum System passenden Echtzeitscheduler (Single oder Dualprozessor).
- rtl_fifo.o: API zur Verwaltung der Echtzeit-FIFOs.
- rtl_posixio.o: Anwendungs-API zur Geräteverwaltung.
- rtl_time.o: API zur zeitlichen Echtzeitverwaltung (Genauigkeit in Nanoseconden). Stellt, der Zeitgeber zur Verfügung.
- rtl.o: Echtzeit-Mikrokern, das das grundlegende Framework und den Interrupthandler zur Verfügung stellt.

- Eine Echtzeitanwendung wird in Form eines Kern-Moduls programmiert, was auch als Echtzeitprozess bezeichnet wird.
- Ein Echtzeitprozess ist meistens aus verschiedenen Echtzeitteilen aufgebaut, die quasi parallel ablaufen. Die Aufgabe des Schedulers ist, die verschiedenen Rechenzeiten je nach Priorität zuzuordnen.
- Diese Echtzeitteile entsprechen C-Funktionen, die Threads genannt werden und einen Echtzeitcode beinhalten, der rechtzeitig, mit garantierter Antwortzeit, ausgeführt werden muss.

Wie man aus der C-Programmierung kennt, sucht der Prozessor (Präprozessor) zuerst die Funktion main; bei den Echtzeitmodulen sind es zwei Funktionen, die erwartet werden, und zwar init_module(), die beim Laden des Moduls aufgerufen wird und dient dem Einrichten des Echtzeitprozesses und die Funktion cleanup_module(), die das Gegenteil gewährleistet, also beim Entladen des Moduls aufgerufen wird und dessen Aufräumen dient.

7. LITERATURREFERENZ

[1] http://www.sfb501.uni-kl.de/a1doc/glossary/v-model.html

[2] http://www.stsc.hill.af.mil/crosstalk/2000/06/hirschberg.html

[3] http://www.agilealliance.org/articles/articles/onAnalysis

[4] http://salmosa.kaist.ac.kr/~course/DrBae/cs550_2002/project.html

[5] http://www.arcom.com

[6]: http://www.easysw.com/~mike/serial/serial.html

[9]: http://en.tldp.org/HOWTO/IO-Port-Programming.html bzw
http://ibiblio.org/pub/Linux/docs/HOWTO/IO-Port-Programming

[10]: http://www.masoner.net/articles/async.html

[11]: http://www.masoner.net/articles/async.html

[12]: http://eavr.u-strassbourg.fr/library/teaching/rt/siframe.htm

[15]: http://www.linuxdevices.com/articles/AT2760742655.html
, http://www.linuxdevices.com/articles/AT9952405558.html
und http://www.linuxdevices.com/articles/AT8073314981.html

[16]: ftp://ftp.kernel.org/pub/linux/kernel/v2/linux-2.4.4.tar.gz ;
ftp://ftp.rtlinux.com/pub/rtlinux/v3/rtlinux-3.1.tar.gz

[8]: Richard, Stevens (1992): Advanced programming in the UNIX environment. USA: ADDISON-WESLEY.

[7]: Gräfe, Martin (2005): C und Linux. München: Hansen Verlag.

[14]: Embedded Linux: Handbuch für Entwickler v. Robert Schwebel, 1. Auflage : mitp-Verlag. Bonn 2001.

[13]: Herold, Helmut (2004):Linux UNIX Systemprogrammierung. München: ADDISON-WESLEY.

ERKLÄRUNG

Hier bestätige ich, dass ich diese Arbeit selbständig angefertigt und nur die aufgeführten Quellen verwendet habe.

Kiel, den 22. Juni 2006