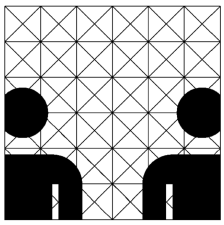


Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Department Informatik
Technische Aspekte Multimodaler Systeme (TAMS)

Diplomarbeit

Entwicklung eines Eingebetteten Systems zur ressourcenschonenden und plattformunabhängigen Anbindung von SICK-Lasermesssystemen

Oktober 2006

Hannes Bistry

0bistry@informatik.uni-hamburg.de
Matrikelnummer: 5317405

Stephan Pöhlsen

0poehlse@informatik.uni-hamburg.de
Matrikelnummer: 5303153

Erstbetreuer: Prof. Dr. Jianwei Zhang
Zweitbetreuer: Prof. Dr.-Ing. Dietmar P. F. Möller

Abstract

Gegenstand dieser Diplomarbeit ist die Entwicklung einer effizienten Methode zur Anbindung zweier SICK-Lasermesssystemen an einen Serviceroboter des Arbeitsbereichs TAMS der Universität Hamburg. Bei den Lasermesssystemen handelt es sich um Sensoren zur Abstandsmessung unter verschiedenen Winkeln, die beim Serviceroboter zur Lokalisierung und Kollisionsvermeidung eingesetzt werden. In der bisherigen Implementierung der Anbindung verursachen diese eine hohe Prozessorauslastung des Steuerrechners im Serviceroboter. Zur Lösung dieses Problems wird ein Eingebettetes System basierend auf einem Rabbit-3000-Prozessor entwickelt, das mit den Lasermesssystemen über serielle Schnittstellen nach dem RS-422-Standard kommuniziert und die Daten über eine Ethernet-Verbindung an den Steuerrechner weiterleitet. Durch die effiziente Verarbeitung von eingehendem Netzwerkverkehr im Steuerrechner sinkt dessen Auslastung deutlich. Für eine weitere Entlastung des Steuerrechners sorgt eine Vorverarbeitung der Messdaten im Rabbit-3000-Prozessor. Eine Analyse hinsichtlich der Leistungsfähigkeit der neuen Anbindungsmethode wird durchgeführt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Viele aktuelle Forschungsprojekte befassen sich mit Robotik, Sensorik und künstlicher Intelligenz. Durch ständige Fortschritte bei der Miniaturisierung, Steigerung der Rechenleistung und Verringerung des Energiebedarfs eröffnen sich bei der Entwicklung intelligenter Systeme immer neue Möglichkeiten.

Sollen Bewegungsabläufe von diesen Systemen elektronisch gesteuert werden, spielt die Erfassung der Arbeitsumgebung eine wichtige Rolle. Nur wenn der eigene Standort sowie die Positionen von Objekten und Hindernissen hinreichend bekannt sind, kann eine Planung der auszuführenden Bewegung erfolgen und die Kollisionsfreiheit gewährleistet werden. Informationen über die Umgebung können mit Hilfe verschiedener Sensorsysteme gewonnen werden. Bei der Bewegung eines mobilen Robotersystems bieten sich zur Orientierung Lasermesssysteme an. Diese Messsysteme bestimmen vom eigenen Standort aus unter verschiedenen Winkeln die Entfernung zum jeweils nächstgelegenen Hindernis. Auf diese Art entsteht ein Messdatensatz, der die Konturen der Umgebung beschreibt. Über einen Vergleich mit einer Datenbasis, in der die Grundrisse schon bekannter Räume gespeichert sind, kann die eigene Position bestimmt werden. Zusätzlich können die aufgenommenen Messwerte zur Kollisionsvermeidung verwendet werden.

Der Arbeitsbereich TAMS[1] des Departments Informatik an der Universität Hamburg betreibt im Rahmen der Forschung ein Serviceroboterprojekt. Hierbei geht es um Betrieb und Weiterentwicklung eines mobilen Roboters, bei dem zwei Lasermesssysteme der Firma SICK eingesetzt werden. Die bisherige Anbindung dieser Lasermesssysteme an den Serviceroboter ist jedoch mit einigen Problemen verbunden. Der Betrieb verursacht eine hohe Prozessorauslastung des verwendeten Steuerrechners, so dass ein Teil der Messdaten verloren geht.

1.1 Ziel dieser Arbeit

Ziel dieser Diplomarbeit ist die Entwicklung einer ressourcenschonenden und plattformunabhängigen Anbindung der SICK-Lasermesssysteme an den mobilen Serviceroboter. Realisiert werden soll diese Anbindung durch ein Eingebettetes System, das zwei Lasermesssysteme ansteuert und die Messdaten dann in aufbereiteter Form weiterleitet. Eine Vorverarbeitung der Messdaten soll dabei eine weitere Entlastung des Steuerrechners ermöglichen. Die Kriterien für die zu entwickelnde Anbindungsmethode sind Kosteneffizienz, Leistungsfähigkeit, Zuverlässigkeit sowie die unkomplizierte Integrierbarkeit in verschiedene Systeme. Die Lösung soll unabhängig von Hardwarearchitektur und Betriebssystem des Hostrechners einsetzbar sein. Diese Vorgaben erfordern eine gründliche Planung bei der Wahl der einzusetzenden Hardware und der Schnittstelle zur Anbindung an das Hostsystem.

[1] Technische Aspekte multimodaler Systeme

Abbildung 1.1: Der mobile Serviceroboter des Arbeitsbereichs TAMS

Die Integration der neuen Anbindung in das Serviceroboterprojekt ist ebenfalls Gegenstand dieser Arbeit. Hierbei muss die Steuersoftware des Serviceroboters in weiten Teilen angepasst werden. Diese Anpassungen sollen möglichst transparent erfolgen, das heißt, dass sich für die Benutzung der Software keine Änderungen ergeben. Die gewonnenen Erkenntnisse sowie die zum Verständnis des Sachverhalts notwendigen Grundlagen sollen in dieser Arbeit ausführlich dargestellt werden.

1.2 Der mobile Serviceroboter des Arbeitsbereichs TAMS

In diesem Abschnitt wird der mobile Serviceroboter des Arbeitsbereichs TAMS vorgestellt. Dieser Roboter wurde von der Firma Neobotix[2] entwickelt. Abbildung 1.1 zeigt ein Foto des Roboters. Es handelt sich um eine mobile Plattform auf fünf Rädern, von denen zwei unabhängig voneinander über Elektromotoren angetrieben werden. Über diesen sogenannten Differentialantrieb können durch getrennt steuerbare Drehrichtung und Drehgeschwindigkeit der Antriebsräder neben Geradeausfahrten auch Kurvenfahrten und Drehbewegungen um die eigene Achse ausgeführt werden. Die drei weiteren Räder sind in ihrer Richtung frei drehbar und dienen zur Stabilisierung. Um Informationen über die gefahrene Strecke zu gewinnen, werden die Drehbewegungen der Antriebsräder digital überwacht.

[2] www.neobotix.de

Weitere Aktuatoren sind zwei Roboterarme des Typs Mitsubishi PA-10-6C. Diese sechsgelenkigen Arme haben eine Tragkraft von je 6 kg. An jedem dieser Arme ist eine Roboterhand der Firma Barret Technology installiert, die über drei einzeln steuerbare Finger verfügt. Über entsprechende Sensoren kann die Kraft gemessen werden, die auf die Finger wirkt. Der mobile Serviceroboter ist somit in der Lage, Gegenstände zu greifen und diese an einem anderen Ort abzulegen.

Die zwei Lasermesssysteme vom Typ SICK LMS 200 des Serviceroboters dienen zur Orientierung innerhalb von Räumen. Als weitere Sensoren sind verschiedene Kamerasysteme installiert. Das Stereokamerasystem basiert auf zwei Kameras des Typs Sony DFW-VL500, die über eine IEEE-1394-Schnittstelle angebunden sind. Über eine Pan-Tilt-Unit lassen sich die Kameras in nahezu beliebige Richtungen drehen. Aus den Bildern der beiden Kameras können dreidimensionale Bildinformationen rekonstruiert werden. Das Omnivisionsystem basiert auf einer Kamera ähnlichen Typs. Über einen hyperboloiden Spiegel[3] kann ein Bild des gesamten Raumes aufgenommen werden, das mittels entsprechender Operationen in eine andere Perspektive transformiert werden kann. An jede der Barret-Roboterhände ist ebenfalls eine kleine Kamera montiert, mit der die Annäherung der Hand an Gegenstände gesteuert und überwacht werden kann. Diese Kameras geben ein analoges Videosignal aus, das zur Weiterverarbeitung digitalisiert werden muss.

Der Steuerrechner des Serviceroboters ist ein Industrie-PC mit einem Pentium-4-Prozessor und einer an die mobile Plattform angepassten Stromversorgung. Aufgabe des Rechners ist es, alle Sensordaten entgegenzunehmen, diese zu verarbeiten und entsprechend der momentanen Aufgabe die Aktuatoren zu steuern. Dazu sind im Rechner verschiedene Schnittstellenkarten eingebaut.

Beim Serviceroboter handelt es sich um ein multimodales System. Ein System wird als multimodal bezeichnet, wenn es mehrere unterschiedliche Ein- und Ausgabevarianten aufweist. Verschiedene Sensortypen liefern Informationen über die Arbeitsumgebung und ermöglichen die Orientierung innerhalb dieser Umgebung. Das Zusammenfassen der verschiedenen Sensordaten soll das System robust gegen mögliche Störungen machen.

Es ergibt sich das Problem, dass die Verarbeitung der Sensordaten eine hohe Systemauslastung erzeugt und ein gleichzeitiger Betrieb aller Sensoren momentan nicht möglich ist. Besonders kritisch verhalten sich die SICK-Lasermesssysteme. Sie werden über eine RS-422-Schnittstellenkarte der Firma Moxa angesteuert. Werden die Lasermessdaten im sogenannten Echtzeitmodus angefordert, so kommt es zu einer hohen Interruptaktivität. Dabei werden pro ausgelöstem Interrupt nur wenige Byte übertragen. Obwohl die Datenrate für moderne Systeme kein Problem ist, führt diese Anbindung der Lasermesssysteme zu folgenden Beeinträchtigungen:

- hohe Systemauslastung
- Verlust von Messwerten
- Instabilitäten des Steuerrechners

Auf die Echtzeitmessdaten kann nicht verzichtet werden, da diese zur Kollisionsvermeidung benötigt werden. Für andere Anwendungen wie die Verarbeitung der Kamerabilder steht auf dem System deshalb nicht mehr genügend Rechenzeit zur Verfügung. Daher stellt diese Anbindung der Lasermesssysteme beim Betrieb des Serviceroboters ein Problem dar. Im Rahmen dieser Diplomarbeit soll durch die Entwicklung einer effizienteren Anbindungsmethode das Problem gelöst werden.

[3] Hyperboloide Spiegel sind für 360° Rundumblicke mit einer Kamera geeignet.

Abbildung 1.2: Volkswagen-Touareg der Stanford Engineering School, Quelle: [Bon05]

1.3 Motivation: weitere Einsatzgebiete von Lasermesssystemen

Dieser Abschnitt gibt einen kurzen Einblick in weitere aktuelle Entwicklungen in der Robotik, die sich ebenfalls auf die Leistung von Lasermesssystemen stützen. Es wird hierbei der Volkswagen-Touareg der Stanford Engineering School sowie der Roboter Kurt3D der Universität Osnabrück betrachtet.

1.3.1 Beispiel 1: Stanford-Touareg

Die hier zusammengetragenen Informationen stammen aus [Bon05]. Beim modifizierten VW-Touareg handelt es sich um einen autonom fahrenden Geländewagen, der anlässlich der Grand Challange 2005 entwickelt wurde (Abbildung 1.2). Bei diesem Wettbewerb traten computergesteuerte Fahrzeuge ohne Fahrer gegeneinander an. Veranstalter war die DARPA[4] des US-Verteidigungsministeriums. Es war eine Strecke von 212 km durch die Wüste im US-Bundesstaat Nevada zurückzulegen. Die Streckenführung wurde erst kurz vor Start bekannt gegeben, so dass die einzelnen Teams nicht die Möglichkeit hatten, ihre Fahrzeuge auf diese Strecke zu optimieren. Anhand von GPS-Koordinaten wurden die zu passierenden Wegpunkte vorgegeben. Start und Ziel lagen am selben Ort. Eine besondere Herausforderung lag in der teils bergigen Landschaft mit Hindernissen wie Kakteen, Felsen und Gräben. Eine Kommunikation der Teams mit den Fahrzeugen war nicht gestattet, damit auf die Entscheidungsfindung der künstlichen Intelligenz keinen Einfluss genommen werden konnte. Bereits 2004 war diese Rallye durchgeführt worden,

[4]Defense Advanced Research Projects Agency

wobei keines der Fahrzeuge mehr als zwölf Kilometer der Strecke bewältigen konnte. Im Jahr 2005 erreichten vier der 23 gestarteten Fahrzeuge das Ziel innerhalb des erlaubten Zeitraums von zehn Stunden. Schnellstes Fahrzeug war mit knapp sieben Stunden der erwähnte VW-Touareg.

Dieses Fahrzeug ist aus einer Kooperation des Volkswagen-Konzerns mit der Universität Stanford entstanden. Der Touareg gehört als Serienmodell der Kategorie der SUVs[5] an. Im Kofferraum des Fahrzeuges sind sechs Rechner mit Pentium-M-Prozessoren installiert. Fünf dieser Rechner laufen auf der Linux-Distribution Fedora Core 4, einer auf Debian.

Als Sensoren sind neben einer Kamera und einem Radar fünf Lasermesssysteme der Firma SICK installiert. Diese dienen zur Abstandsmessung auf kurzen und mittleren Distanzen, während das Radarsystem Distanzen bis etwa 250 Meter messen kann. Ein AGPS-System[6] lässt eine Positionsbestimmung mit einer Genauigkeit von etwa 20 cm zu. Hierbei handelt es sich um eine Erweiterung des herkömmlichen GPS-Systems, bei der zur Steigerung der Genauigkeit auch terrestrische Sender ausgewertet werden. Weitere Sensoren erfassen die Neigungs- und Richtungsdaten des Fahrzeuges. Alle Sensordaten werden vom Computersystem aufgenommen und verarbeitet.

Die Multimodalität des Systems stellt besonders in diesem Anwendungsgebiet eine wichtige Voraussetzung für die Leistungsfähigkeit dar. Es ist nicht ausreichend, Hindernisse nur zu erkennen, sondern es muss mit Hilfe des Kamerasystems ebenfalls eine Klassifikation der Hindernisse stattfinden. So können beispielsweise Sträucher oder die in dieser Region oft vorkommenden Steppenläufer[7] überfahren werden, während Felsen und größeren Steinen unbedingt ausgewichen werden muss.

Der Erfolg des Fahrzeuges wurde in [Bon05] als „Meilenstein in der Fahrzeug-Robotik“ bewertet. Zu diesen Leistungen haben auch die Lasermesssysteme beigetragen. Anhand der Ausstattung des Fahrzeuges mit sechs Computern lässt sich ohne weitere Kenntnis der Implementierungsdetails abschätzen, dass die Verarbeitung der Sensordaten einen hohen Rechenaufwand verursacht. Sicherlich haben daran auch die Lasermesssysteme einen entscheidenden Anteil. Durch eine ressourcenschonende Anbindung und Datenvorverarbeitung besteht hier wahrscheinlich ein hohes Optimierungspotential, um Kosten, Gewicht, Stromverbrauch und Größe der verwendeten Steuersysteme zu reduzieren.

Die Leistung des VW-Touareg in der Grand Challange stellt vermutlich erst einen Anfang in der Fahrzeug-Robotik dar. Mit großer Wahrscheinlichkeit wäre ein menschlicher Fahrer in diesem Umfeld der Computersteuerung hoch überlegen, so dass viel Raum für Weiterentwicklungen bleibt.

1.3.2 Beispiel 2: Kurt3D der Universität Osnabrück

Der Roboter Kurt3D kam 2006 im Rahmen der European Land Robot Trial (ELROB) zum Einsatz. Es handelt sich dabei nicht um einen Wettbewerb wie bei der Grand Challenge, sondern um eine Demonstration der technischen Fähigkeiten unbemannter Fahrzeuge (siehe [ELROB]). Dementsprechend offen ist das Reglement für die teilnehmenden Fahrzeuge ausgelegt:

[5] *Sports Utility Vehicle* (Geländewagen vorwiegend für den Straßenbetrieb)
[6] *Assisted Global Positioning System*
[7] rollende Büsche

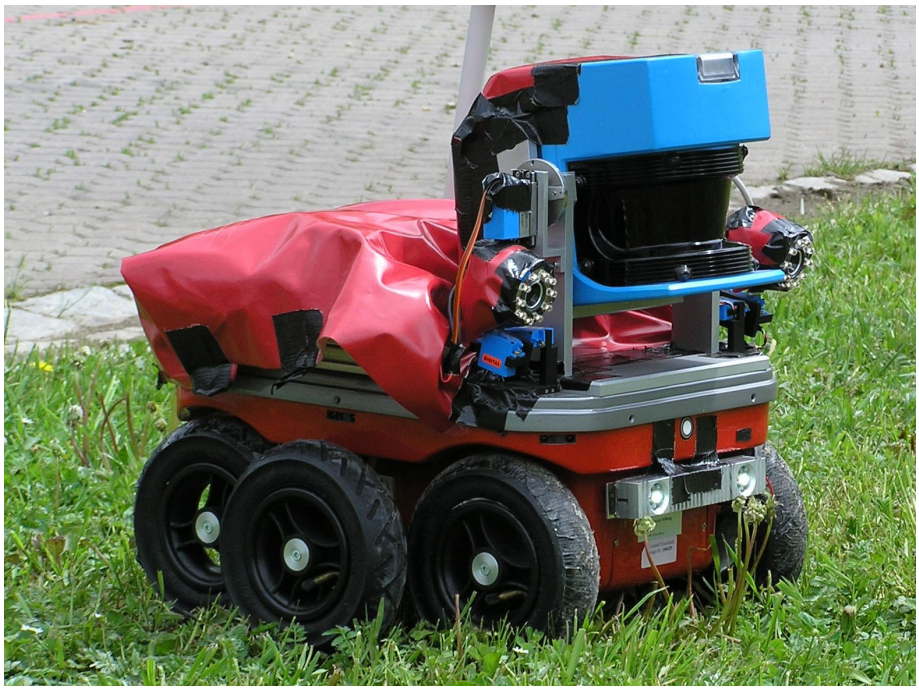


Abbildung 1.3: Der Roboter Kurt3D der Universität Osnabrück, Quelle: [UniOS]

- Das Fahrzeug ist unbemannt.
- Das Gewicht beträgt maximal drei Tonnen.
- Die Fortbewegung erfolgt durch Bodenkontakt.

Bei dieser Veranstaltung können auch ferngelenkte Fahrzeuge eingesetzt werden. Im Rahmen der ELROB werden militärische Szenarios nachgestellt, in denen diese Fahrzeuge dann eingesetzt werden (z.B. Katastropheneinsätze, Aufklärung in zerstörten Gebäuden). Bestimmte Aufgaben werden an die Teams vergeben. So können beispielsweise Aufklärungsbilder eines festgelegten Ortes angefordert werden.

Der Roboter Kurt3D ist in Abbildung 1.3 dargestellt. Dieses System wurde zunächst am Fraunhofer Institut für autonome intelligente Systeme entworfen und später an der Universität Osnabrück weiterentwickelt. Es wird auch hier ein Lasermesssystem der Firma SICK eingesetzt, dessen Neigungswinkel über ein Gelenk und einen entsprechenden Antrieb verändert werden kann. Dadurch werden dreidimensionale Konturaufnahmen der Landschaft ermöglicht. Zwei Kameras liefern zusätzlich Bilder der Umgebung. Die Steuerung erfolgt nicht autonom, sondern es werden über eine WLAN-Verbindung entsprechende Kommandos gesendet.

1.4 Aufbau der Arbeit

In diesem Abschnitt wird die Gliederungsstruktur der vorliegenden Arbeit erläutert. Neben der Einleitung umfasst die Arbeit sechs weitere Kapitel. Diese Einteilung erfolgt in Anlehnung an den tatsächlichen Ablauf der durchgeführten Arbeiten. In Kapitel 2 und 3 werden die bisherige

und die geplante Anbindung der Lasermesssysteme an den Serviceroboter vorgestellt. Kapitel 4 und 5 befassen sich mit der Entwicklung der neuen Anbindung. Eine Auswertung der Ergebnisse erfolgt in Kapitel 6 und 7. Diese Unterteilung wird im Folgenden näher ausgeführt:

Die vorbereitenden Schritte sind in Kapitel 2 und 3 beschrieben. Kapitel 2 befasst sich mit den Themenbereichen, die für die bisherige Anbindung der Lasermesssysteme maßgeblich sind. Dazu gehören Grundlagen über Sensoren, serielle Datenübertragung und Eingebettete Systeme. Des Weiteren werden die SICK-Lasermesssysteme und der Steuerrechner des Serviceroboters vorgestellt. Eine Analyse des Systems zeigt Probleme auf, die durch die ineffiziente Anbindung verursacht werden. In Kapitel 3 wird ein Konzept zur Behebung dieser Probleme entwickelt. Dazu wird auf gewünschte Funktionen des zu entwickelnden Systems und weitere Entwicklungskriterien eingegangen. Die Realisierung erfolgt mit einem Rabbit Powercore 3800, der über serielle Schnittstellen mit den Lasermesssystemen kommuniziert und die Messdaten über eine Ethernet-Verbindung zum Steuerrechner weiterleitet. Der Rabbit Powercore 3800, der Mikroprozessor (Rabbit 3000) und die Entwicklungsumgebung (Dynamic C) werden vorgestellt. Bereits bei der Beschreibung des Mikroprozessors werden später benötigte Konfigurationen erläutert, da schon bei der Auswahl der Hardware sicherzustellen ist, dass eine zum späteren Einsatzzweck passende Konfiguration möglich ist.

Kapitel 4 und 5 stellen die durchgeführten Entwicklungsarbeiten dar. Die Entwicklung eines Hardware-Prototypen wird in Kapitel 4 beschrieben. Er setzt sich aus dem Powercore 3800 und einer entwickelten Platine mit zusätzlichen elektronischen Bauteilen zusammen. Die Entwicklung umfasst einen Test der Schnittstellen, bei dem die serielle Datenübertragung mit einer Geschwindigkeit von 500.000 Baud und die Kommunikation mit dem Hostrechner über Ethernet geprüft werden. Hierzu wird ein serieller Testsender entworfen, der bekannte Daten an den Rabbit 3000 sendet. Die Daten werden dann als UDP-Pakete an den Hostrechner weitergeleitet und dort geprüft. Kapitel 5 stellt die Software für die bereits funktionierende Hardware vor. Nach einem Überblick des Programmablaufs folgt eine Beschreibung der Datenstruktur, in der die Interrupt-Service-Routine eingehende Telegramme zwischenspeichert. Durch Auswertung der Telegramme erfolgt eine Steuerung der Lasermesssysteme und eine Vorverarbeitung der Messdaten. Anschließend wird das Datenformat der zwischen Powercore 3800 und Hostrechner ausgetauschten UDP-Pakete beschrieben. Die Software im Steuerrechner wird so angepasst, dass Messdaten und Ergebnisse der Vorverarbeitung genutzt werden.

Ergebnisse dieser Arbeit sind in Kapitel 6 und 7 ausgeführt. Kapitel 6 analysiert die Leistung des entwickelten Systems. Messergebnisse bezüglich der Zuverlässigkeit der Datenübertragung, der Verzögerung von Messdaten sowie der Systemauslastung des Rabbit 3000 und des Hostrechners werden präsentiert. Im abschließenden Kapitel 7 werden weitere Entwicklungsmöglichkeiten betrachtet, die im Rahmen dieser Arbeit nicht umgesetzt wurden.

2 Grundlagen

Gegenstand dieses Kapitels ist die bisherige Anbindung der Lasermesssysteme an den Steuerrechner des Serviceroboters (Abbildung 2.1). Die vorhandenen Geräte sowie die zum Verständnis dieser Arbeit erforderlichen Grundlagen werden dargestellt. Zunächst werden allgemeine Informationen über Sensoren gegeben. Danach wird der Lasermesssystemsensor LMS 200 von SICK vorgestellt, wobei insbesondere auf die Schnittstelle und den Kommunikationsablauf eingegangen wird. Im darauf folgenden Abschnitt werden die Grundlagen der seriellen Datenübertragung dargelegt. Der RS-422-Standard zur Anbindung der SICK-Lasermesssysteme wird erklärt. Im vierten Teil des Kapitels geht es um Eingebettete Systeme im Allgemeinen, um im Hinblick auf den späteren Entwurf Kriterien für die Hardwareauswahl herauszuarbeiten. Der letzte Abschnitt stellt die Systemhardware des Roboters mit der Softwareschnittstelle im Hostrechner dar. Durch Auswertung der Logausgaben werden dabei die vorhandenen Probleme analysiert.

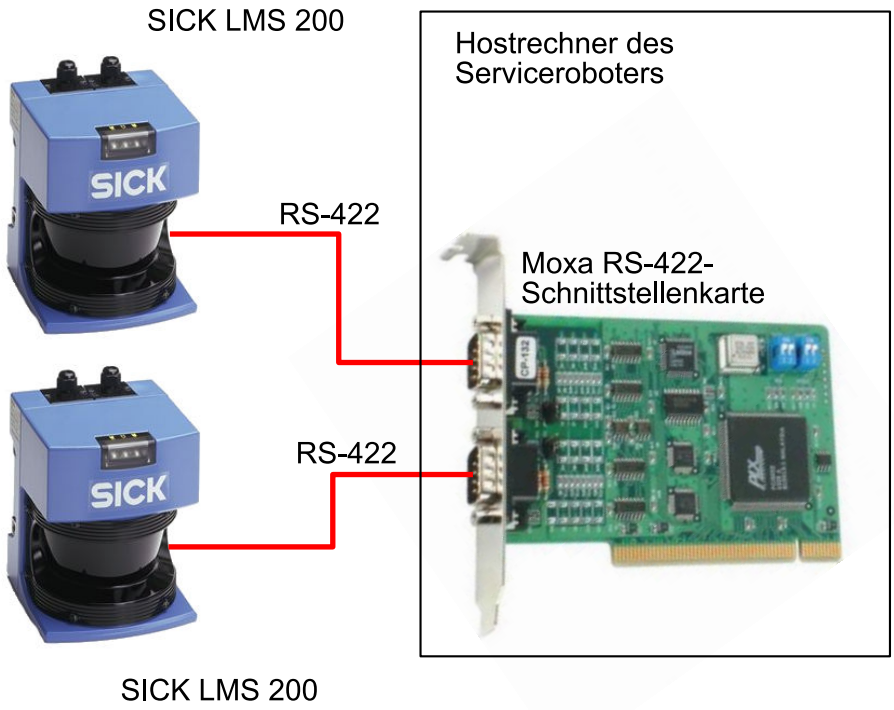


Abbildung 2.1: Bisherige Anbindung der Lasermesssysteme

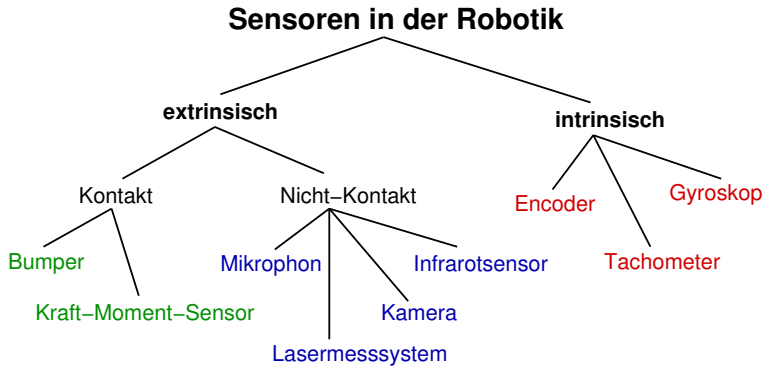


Abbildung 2.2: Beispiel einer Sensorklassifikation, Quelle: [Zha05, S. 16]

2.1 Sensoren

In [Zha05] sind die Grundlagen von Sensoren dargestellt. Ein Sensor empfängt ein Signal oder Stimulus (Reiz) und reagiert darauf. Natürliche Sensoren, wie Auge, Ohren oder Haut, reagieren mit einem elektrochemischen Signal auf ihren Nervenbahnen. Physikalische Sensoren reagieren dagegen mit einem elektrischen Ausgangssignal. Die Information, die ein elektrisches Signal trägt, kann aus der Spannung, der Stromstärke oder der Ladung gewonnen werden. Das Signal kann sich in Amplitude, Frequenz oder Phase unterscheiden.

Aufgrund der hohen Anzahl unterschiedlicher Sensoren bietet es sich an, sie in verschiedene Typen zu unterteilen. Extrinsische Sensoren können von intrinsischen Sensoren unterschieden werden. Extrinsische Sensoren reagieren auf Signale oder Stimuli aus ihrer Umgebung. Ein Mikrofon reagiert auf von außen kommende Schallwellen. Intrinsische Sensoren reagieren auf den internen Zustand des Systems, in das sie verbaut sind. Ein Encoder gibt beispielsweise den inneren Zustand des Systems, nämlich den Drehwinkel einer Achse, aus. Ein weiteres Unterscheidungskriterium ist die Einteilung in aktive und passive Sensortypen. Nach [Fra04, S. 7] variieren aktive Sensoren das elektrische Signal bei einer Änderung des Stimulus. Sie „verbrauchen“ Energie und erzeugen eine direkte elektrische Größe aus einer nicht-elektrischen Größe, dem Stimulus. Dadurch wirken sie wie eine elektrische Spannungsquelle. Passive Sensoren erzeugen ein direktes elektrisches Signal. Sie benötigen keine zusätzliche Energiequelle, sie wandeln die Erregungsenergie in ein elektrisches Signal um. Piezoelektrische Drucksensoren sind beispielsweise passiv.

Außer den unterschiedlichen Sensortypen extrinsisch/intrinsisch und aktiv/passiv lassen sich Sensoren nach unterschiedlichen Gesichtspunkten klassifizieren. Eine mögliche Klassifikation ist in Abbildung 2.2 zu sehen. Andere Klassifikationsmerkmale sind:

- die Art des Stimulus
- Eigenschaft, Spezifikation und Parameter
- Art der Stimulusdetektion
- Umwandlungsart des Stimulus ins Ausgangssignal
- Material eines Sensors

Gleiche Klassifikationsbegriffe werden von verschiedenen Autoren unterschiedlich definiert.

Sensoren zur Abstandsmessung lassen sich in Infrarot-, Ultraschall- oder Lasersensoren einteilen. Abstandssensoren sind extrinsisch, da sie die Entfernung zu Gegenständen aus der Systemumgebung messen. Sie gehören in die Klasse der aktiven Sensoren, da sie elektrische Energie benötigen um Infrarotstrahlung, Ultraschall oder Laserimpulse zu erzeugen. Physikalische Prinzipien zur Abstandsmessung sind Signallaufzeit, Triangulation, Messung der Signalintensität und Phasendifferenz von Signalen.

2.2 SICK-Lasermesssystem LMS 200

SICK-Lasermesssysteme dienen zur Entfernungsmessung unter verschiedenen Winkeln. Sie tasten berührungslos in der Ebene (zweidimensional, fächerförmig) optisch ab und stellen die gemessenen Entfernungswerte über einen digitalen Ausgang bereit. Dabei werden weder Reflektoren noch Positionsmarken zur Abstandsmessung benötigt. Durch die hohe Messgeschwindigkeit können auch sich bewegende Objekte erkannt werden. Der Messbereich beträgt bis zu 80 Meter bei einer Auflösung von einem Zentimeter oder acht Meter bei einer Auflösung im Millimeterbereich.

Diese Eigenschaften führen dazu, dass Lasermesssysteme in unterschiedlichsten Einsatzgebieten Verwendung finden [SickTB]. Die Einsatzgebiete lassen sich in zwei Klassen einteilen.

Die erste Klasse nutzt die Bereichsüberwachung der Lasermesssysteme, die autark funktioniert und damit keinerlei externe Software benötigt. Eine Auswertung findet intern im Lasermesssystem statt. Dazu gehören:

- Überstandskontrolle von Gegenständen (zum Beispiel Höhenkontrolle bei Fahrzeugen)
- Freilandüberwachung im Objektschutz

Die Lasermesssysteme werden dafür einmal konfiguriert und geben über ihre Schaltausgänge anschließend Bereichsverletzungen an. Weitere Informationen zur Bereichsüberwachung sind im Abschnitt 2.2.2 zu finden.

Die Anwendungsgebiete Objekt- und Positionsvermessung bilden die zweite Klasse. Die drei Schaltausgänge der Lasermesssysteme finden hierbei keine Verwendung. Die Entfernungsmessdaten stehen an den digitalen RS-422-Ausgängen zur Verfügung. Zur zweiten Klasse gehören:

- Volumen und Konturbestimmung von Objekten, wie beispielsweise Schüttgut auf Förderbändern
- Volumen- und/oder Positionsermittlung von Objekten wie Paketen, Paletten und Containern
- Kollisionsschutz und Andockvorgänge an Fahrzeugen und Kränen
- Objektklassifizierung, beispielsweise zur Unterscheidung von PKWs und LKWs
- Ermittlung der eigenen Position

Eine Auswertung der Entfernungsmessdaten erfolgt mit externer Software. Welche Funktionen dafür zur Verfügung stehen wird in 2.2.2 erläutert.

SICK-Lasermesssysteme können die Reflexionsstärke statt der Entfernung bzw. zusätzlich zur Entfernung ausgeben. Optimiert sind sie jedoch für Entfernungsmessungen. Von den zwei Byte, die pro Messwert zur Verfügung stehen, können ein bis drei Bit für die Reflexionsintensität verwendet werden. Spielt die Entfernung keine Rolle, so ist es möglich, alle 16 Bit pro Messwert für die Intensität zu nutzen. Die Reflexionsintensitätswerte sind nicht linear verteilt. Somit ist

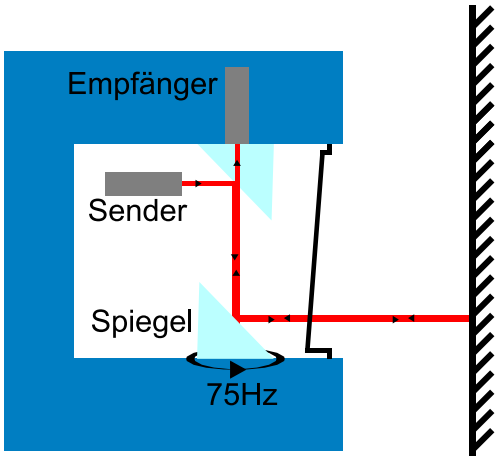


Abbildung 2.3: Funktionsprinzip eines Lasermesssystems

eine Aussage über die absolute Helligkeit von Gegenständen nicht möglich. Helligkeitsänderungen sind feststellbar. Aus Helligkeitsänderungen können keine Rückschlüsse auf Entfernungen gezogen werden, da unterschiedliche Materialien unterschiedlich starke Reflexionseingenschaften haben. Die Reflexionsstärke ist zur Erkennung von Reflektormarken geeignet.

Überlappen sich Messbereiche zweier SICK-Lasermesssysteme, so ist nicht auszuschließen, dass sie sich durch Laserimpulse gegenseitig beeinflussen. Zwei SICK-Lasermesssysteme lassen sich deshalb synchronisieren. Ein System wird als Master konfiguriert und gibt den Takt vor, das andere Lasermesssystem wird als Slave angebunden und sendet zeitlich versetzt, so dass immer nur eines zur Zeit seinen Laserimpuls aussenden kann.

2.2.1 Funktionsprinzip

Die SICK-Lasermesssysteme arbeiten nach dem *time-of-flight*-Prinzip (TOF), auch bekannt unter LIDAR (*light detection and ranging*) oder Laser Radar (Abbildung 2.3). Gemessen wird die Zeit zwischen Aussendung und Empfang eines Laserimpulses. Aus dieser „Flugzeit“ t (für Hin- und Rückweg) lässt sich die Entfernung d mit Hilfe der Lichtgeschwindigkeit c_0 wie folgt berechnen:

$$d = \frac{t \cdot c_0}{2} \tag{2.1}$$

Lasermesssysteme senden zur Zeitmessung einen gepulsten Laserstrahl aus. Dieser Laserimpuls wird von Objekten reflektiert. Es wird die Zeit bis zum Eintreffen der Reflexion am Empfangssensor des Lasermesssystems gemessen. Genau genommen misst der Empfangssensor des Lasermesssystems die Intensität und signalisiert einen Empfang bei der ersten Überschreitung des eingestellten Schwellwertes. Dazu wird über einen kurzen Zeitraum die Reflexionsstärke integriert. Hierbei handelt es sich um die Messimpulsdauer. Bei den Lasermesssystemen von SICK wird bei einer Wellenlänge von 905 nm gemessen. Diese Wellenlänge liegt im nicht sichtbaren Infrarotbereich.

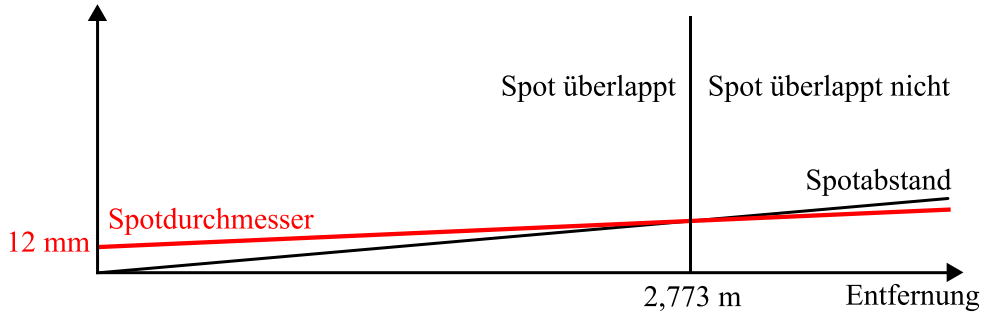


Abbildung 2.4: Spotdurchmesser und Spotabstand bei Winkelauflösung 0,5°

Das Lasermesssystem LMS 200 von SICK sendet den Laserimpuls mit einem Öffnungswinkel von 4,4 mrad (ca. 0,2521°). Der Spotdurchmesser ist beim Austritt aus dem Lasermesssystem 12 mm groß (siehe [SickTL, S. 110]). Abgebildet ist der Laserimpuls in rot mit seinem Öffnungswinkel in Abbildung 2.4. Wie der Abbildung zu entnehmen ist, überlappen die Laserimpulse bei Reichweiten unter 2,773 Meter für benachbarte Messungen. Erst bei größeren Abständen gibt es Bereiche, die in keine Entfernungsmessung mit eingehen. Bei zehn Metern Entfernung ergibt sich einen Spotdurchmesser von 5,6 cm. Wird dabei mit einer Winkelauflösung von 0,5° gemessen, so beträgt der Spotabstand ungefähr 8,73 cm. Daraus ergeben sich in einem Abstand von 10 Metern 3,13 cm breite Bereiche, die in keine Messung eingehen.

Der gepulste Laserstrahl wird bei dem LMS 200 über einen rotierenden Spiegel abgelenkt. Für eine Rotation benötigt der Spiegel 13,32 ms (75 Hz). Das LMS 200 sendet die Laserimpulse in ein Gradschritten. Um mit 0,5 Grad abzutasten, benötigt das LMS 200 zwei Spiegelrotationen (26,64 ms). Bei der ersten Rotation wird die Entfernung bei 0°, 1°, 2°, ... und bei der zweiten Rotation bei 0,5°, 1,5°, 2,5°, ... gemessen. Anschließend werden die Messdaten gemeinsam übertragen.

Die Reflexionsintensität hat eine Auswirkung auf die Entfernungsmessung. Folgende Formel ist dazu in [SickTL, S. 107] angegeben:

$$\begin{aligned} \text{ausgegebener Messwert} &= \text{Rohmesswert} \\ &+ \text{Entfernungskorrektur aus interner Entfernungstabelle} \\ &+ \text{Korrektur aus interner Empfangsenergietabelle} \end{aligned} \tag{2.2}$$

Trifft bei Messungen der Laserimpuls auf Objektkanten oder sehr dünne Objekte, so wird nicht die gesamte Energie reflektiert. Ist die Reflexion nicht ausreichend und der Abstand zwischen Objekt und Hintergrund kleiner als ein Meter (Dauer des Messimpulses zur Intensitätsbestimmung), so wird der Rohmesswert durch Anwendung von Gleichung (2.2) falsch korrigiert. Die Reflexionsintensität hat also Auswirkungen auf den ausgegebenen Entfernungsmesswert.

Das LMS 200 kann im cm- oder mm-Modus messen. Der statistische Fehler beträgt fünf Millimeter (Standardabweichung). Quantisierungsfehler bei Messungen im mm-Modus sind zu vernachlässigen, weil das Rauschen einen merklich größeren Einfluss auf die Messwerte hat. Der systematische Fehler von ±20 mm (±4 cm) im mm-Modus (cm-Modus) hat merklich größere Auswirkungen. Woher dieser kommt bzw. wie er zu beseitigen ist, steht nicht in der Dokumentation.

Material	Remission
Fotokarton, schwarz matt	10 %
Karton, grau	20 %
Holz (Tanne roh, verschmutzt)	40 %
PVC grau	50 %
Papier, weiß matt	80 %
Aluminium, schwarz eloxiert	110...150 %
Stahl, rostfrei glänzend	120...150 %
Stahl, hochglänzend	140...200 %
Reflektoren	>2000 %

Tabelle 2.1: Remission von Materialien, aus [SickTB, S. 7]

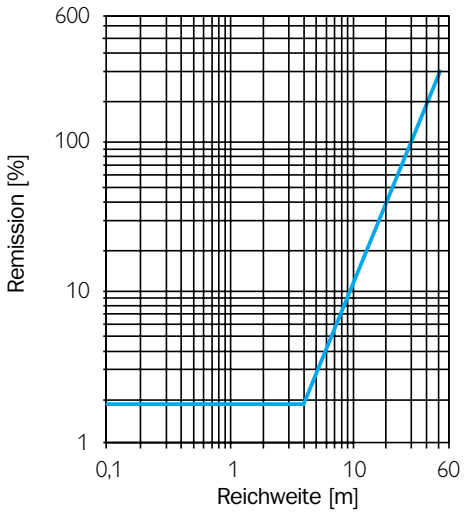


Abbildung 2.5: Benötigte Objektremission in Abhängigkeit der Reichweite, aus [SickTB, S. 7]

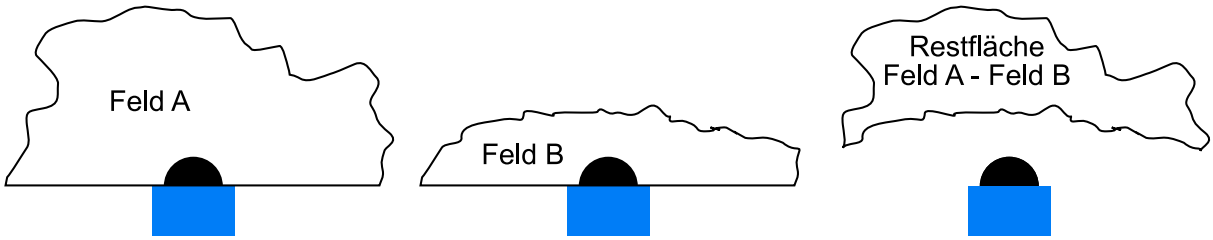


Abbildung 2.6: Darstellung einer Restfläche in der Bereichsüberwachung

Unterschiedliche Objekte haben unterschiedliche Remissionseingenschaften (Reflektivität). In Tabelle 2.1 sind einige Remissionswerte für Materialien nach dem KODAK-Standard aufgeführt. Die Remission für 100 % ist willkürlich festgelegt, so dass auch höhere Remissionen möglich sind.

In der Abbildung 2.5 ist für das LMS 200 angegeben, welche Remission für eine Reichweite benötigt wird, damit die Messung stattfinden kann. Schwarzer matter Fotokarton lässt sich vom LMS 200 also nur bis zu einer Entfernung von ungefähr zehn Metern messen, danach kann kein Objekt mit dieser Oberfläche mehr festgestellt werden.

2.2.2 Betriebsmodi

In diesem Abschnitt werden die beiden Betriebsmodi der SICK-Lasermesssysteme vorgestellt. Als erstes wird die Bereichsüberwachung beschrieben, welche die Messwerte selbstständig auswertet. Der zweite Betriebsmodus liefert die Messwerte zur externen Weiterverarbeitung und beschränkt sich auf elementare Vorverarbeitungen.

Bereichsüberwachung

Bei der Bereichsüberwachung stehen frei definierbare Felder/Bereiche zur Verfügung. Ein Bereich kann frei sein, wenn sich kein Objekt darin befindet, oder es kann eine Bereichsverletzung geben, für den Fall, in dem sich etwas in dem definierten Bereich befindet. Die Felder können als Bereichsdefinition einen Radius, ein Rechteck oder eine freie Form erhalten. Bei der freien Form wird für jeden Winkel die Entfernung gespeichert. Drei Felder A, B und C bilden zusammen einen „Feldsatz“. Zwei Feldsätze können im Lasermesssystem gespeichert werden. Ein Umschalten der Feldsätze kann über den RESTART-Anschluss[1] erfolgen.

Befindet sich ein Objekt in einem der frei definierten Felder, so wird der dazugehörige Schaltausgang geschaltet. Eine Visualisierung findet über die Leuchtdioden direkt an den Lasermesssystemen statt. Die Lasermesssysteme übernehmen die „Freimeldung“ von Bereichen ohne externe Software. Sie arbeiten autark und benötigen externe Software ausschließlich für die Konfiguration der Felder.

Es gibt für die frei definierbaren Felder zwei Erweiterungen, die sogenannte „Restfläche“ (siehe Abbildung 2.6) und die „Kontur als Referenz“. Die Restfläche ist notwendig, wenn ein zu überwachender Bereich nicht direkt vor dem Lasermesssystem beginnen soll. Soll ein Lasermesssystem

[1] Weitere Funktionen des RESTART-Anschlusses sind in [SickTB, S. 18] zu finden.

eine Fläche überwachen, die hinter einem Weg (in der Abbildung 2.6 Feld B) liegt, so müssen Bereichsverletzungen in Feld B ignoriert werden, es entsteht eine sogenannte „Restfläche“. Zum Definieren einer „Restfläche“ werden zwei Felder A und B benötigt. Es wird die Fläche von Feld A abzüglich der Fläche von Feld B überwacht.

Die Bereichsüberwachung prüft nur ob der definierte Bereich frei ist. Dieser Bereich ist durch eine äußere Linie definiert. Ist diese Linie zugleich eine Wand, so kann sie als Kontur betrachtet werden. Die Funktion „Kontur als Referenz“ überwacht nun zusätzlich zum freien Feld diese Wand[2]. Wird diese Wand unterbrochen, weil zum Beispiel eine Tür darin geöffnet wird, zeigt der dazugehörige Schaltausgang auch eine Bereichsverletzung an, da die „Kontur“ unterbrochen wurde. Mit der Kontur als Referenz kann also die Existenz von Objekten überwacht werden.

Um die Robustheit der autark arbeitenden Bereichsüberwachung zu erhöhen, gibt es zwei verschiedene Auswertungen im Lasermesssystem, von denen jedoch nur eine zur Zeit verwendet werden kann. In der „pixelorientierten“ Auswertung wird jeder Strahl einzeln ausgewertet. Dabei wird geprüft, ob in mehreren aufeinander folgenden Messungen Bereichsverletzungen bei diesem Strahl aufgetreten sind. Die pixelorientierte Auswertung ist zur Ausblendung einzelner Fehlmessungen durch Regentropfen, Schneeflocken oder sonstige Partikel vorgesehen. Bei der „scanorientierten“ Auswertung werden Bereichsverletzungen benachbarter Strahlen zusammengefasst. Eine Bereichsverletzung wird erst dann gemeldet, wenn ein zuvor eingestellter Wert in Zentimetern („Blankingfaktor“) überschritten wird. Durch „Objektblanking“ werden kleine Objekte in der Bereichsüberwachung ignoriert.

Objekt- und Positionsvermessung (externe Messwertverarbeitung)

Die externe Messwertverarbeitung benötigt zusätzlich zum Lasermesssystem Software. Für Volumenstrommessungen von Schüttgut und Vermessung von Paketvolumen stehen Standardlösungen zur Verfügung, die vom Hersteller für diese Anwendungsgebiete angeboten werden. Kann mit einer Standardlösung von SICK nicht gearbeitet werden, so muss eigene Auswertesoftware entwickelt werden.

Bei der Entwicklung eigener Auswertesoftware kann auf die interne Messdatenvorverarbeitung der Lasermesssysteme zurückgegriffen werden. Es steht eine Mittelwertbildung über bis zu 250 Messdatensätze zur Verfügung, so dass Messergebnisse mit weniger Rauschen zustande kommen. Eine Datenreduktion kann dadurch erreicht werden, dass der zu übertragende Bereich eingegrenzt wird. Es lassen sich beispielsweise nur die Messwerte zwischen 10° und 80° abfragen.

Als zusätzliche Funktion der SICK-Lasermesssysteme können für individuelle Auswerteaufgaben die Messdaten in Echtzeit abgefragt werden. Dabei können bis zu 75 Messungen in der Sekunde gemacht werden. Gerade für die Roboternavigation ist dieser Echtzeitmodus von großem Interesse.

[2] Der Hersteller nennt diese auch Hintergrundlinie

Beschreibung	Byte	Erklärung
STX (*start of text*)	1	Start-Byte (0x02)
Adresse	1	Adresse des LMS zwischen 0x00 und 0x7f, in der Rückmeldung wird Bit 7 gesetzt
Länge	2	Anzahl folgender Byte ohne Prüfsumme
Code	1	Befehls- bzw. Rückmeldungscode
Daten	n	zusätzliche Daten für den Befehl
Status	1	LMS-200-Status, nur in Rückmeldungen
Prüfsumme	2	CRC-Prüfsumme über alle Byte (von STX an)

Tabelle 2.2: Beschreibung der Telegrammstruktur

2.2.3 Schnittstelle und Telegrammformat

Die SICK-Lasermesssysteme haben eine Hardware-Schnittstelle, die nach dem RS-232- oder RS-422-Standard betrieben werden kann. Ein Jumper im Stecker des Verbindungskabels ist für die Wahl zwischen RS-232 und RS-422 zuständig. Die Daten werden dabei immer im 8N1-Modus (acht Datenbit, keine Parität, ein Stoppbit) übertragen. Nach dem Einschalten des Lasermesssystems ist normalerweise eine Baudrate von 9.600 Baud (Bd) eingestellt, die auf 19.200, 38.400 oder 500.000 Bd umgestellt werden kann. Auf ein dauerhaftes Umkonfigurieren des Lasermesssystems, so dass mit einer höheren Baudrate direkt nach dem Einschaltvorgang kommuniziert wird, sollte aus Kompatibilitätsgründen mit anderer Software verzichtet werden. Der *high-speed*-Modus 500 kBd ist nur bei RS-422 verfügbar.

Wird das Lasermesssystem nicht im *high-speed*-Modus betrieben und trotzdem in den Echtzeitmodus gewechselt, bei dem kontinuierlich Daten gesendet werden, so gehen Messwerte verloren. Die Messdaten aus dem Ausgabepuffer des Lasermesssystems werden so lange versendet, bis sie von neuen Daten überschrieben werden. Kontinuierliches Messen hat Vorrang vor dem Verschicken der Messdaten. Weitere Informationen dazu finden sich in Kapitel 10.9 und der dazugehörigen Tabelle 10-16 in [SickTL, S. 122].

Die Kommunikation mit den Lasermesssystemen findet über Telegramme statt. Ein Telegramm enthält genau ein Befehls- oder Rückmeldungscode. Befehls- und Rückmeldungscode sind genau ein Byte lang und können zusätzliche Daten enthalten. Datentypen mit mehr als acht Bit werden im Telegramm *little-endian*-codiert übertragen, das bedeutet, dass das LSB zuerst gesendet wird. Die Tabelle 2.2 gibt einen Überblick über die Telegrammstruktur.

Neben den Rückmeldungen werden zwei zusätzliche Byte vom Lasermesssystem versendet: Ein ACK („*Acknowledge*“) mit dem Hexwert 0x06 und ein NACK („*Not Acknowledge*“) mit dem Hexwert 0x15. ACK und NACK bestehen nur aus einem Byte und beziehen sich immer auf das letzte Telegramm, welches vom Lasermesssystem empfangen wurde.

Beim Senden eines Befehls muss die Pause zwischen zwei zu sendenden Byte mindestens 55 Mikrosekunden und maximal sechs Millisekunden lang sein. Beim Empfangen müssen Pausen bis zu 14 Millisekunden zwischen den einzelnen Byte akzeptiert werden. Ist eine Pause größer, so sollte von einem neuen Telegramm ausgegangen werden.

Auf einen Befehl antwortet das Lasermesssystem innerhalb von 60 Millisekunden mit einem ACK oder NACK. Kommt weder ein ACK noch ein NACK beim Hostrechner an, so ist die Adresse

(zweites Byte im Telegramm) vermutlich falsch. Wenn ein NACK als Antwort empfangen wird, so ist die Prüfsumme des letzten versendeten Telegramms falsch und es wurde verworfen. Erst nach 30 Millisekunden darf das Telegramm erneut gesendet werden. Ein empfangenes ACK bedeutet, dass das Telegramm beim Lasermesssystem mit einer korrekten Prüfsumme empfangen wurde. Nach dem ACK wird vom Lasermesssystem die Rückmeldung passend zum letzten Befehl gesendet. Wird allerdings ein gültiges Telegramm in einem logisch falschen Zusammenhang an das Lasermesssystem gesendet, so kommt als Antwort das Telegramm mit der Rückantwort 0x92. Ein solcher Fall tritt zum Beispiel auf, wenn ein Zustand verlassen werden soll, in den vorher nicht gewechselt wurde.

Die maximale Antwortzeit auf ein Telegramm zum Abfragen von Messdaten ist 60 Millisekunden (bei 0,25° Schritten). Ein Wechsel des *operation mode* kann bis zu drei Sekunden dauern. Nach einem *power-on-* oder einem *initialise-and-reset*-Befehl können bis zu 60 Sekunden vergehen bis das Lasermesssystem antwortet. Es sind entsprechend lange Wartezeiten in externer Auswertesoftware zu realisieren.

Werden Befehle zum Lasermesssystem gesendet, bevor das Lasermesssystem alle Daten aus seinem Ausgabe-FIFO gesendet hat, so wird die Übertragung des aktuellen Telegramms abgebrochen.

Jedes Rückmeldetelegramm von Lasermesssystemen hat direkt vor der Prüfsumme ein Statusbyte. Das erste Bit (*most significant bit*, Bit 7, Bitmaske 0x80) zeigt an, ob das Lasermesssystem verschmutzt ist. Das zweite Bit (Bit 6, Bitmaske 0x40) zeigt an, dass die Messwerte unplausibel sind. Bit 5 (Bitmaske 0x20) zeigt den Pegel vom RESTART-Anschluss an, eins steht für *high* und null für *low*. Die Bedeutungen der restlichen Bit ist in den Tabellen 2.3 und 2.4 zu finden.

Bit 4	**Bit 3**	
0	0	reserviert
0	1	reserviert
1	0	LMS Typ 6
1	1	Spezielles Gerät

Tabelle 2.3: Beschreibung der Datenquelle im Statusbyte

Bit 2	**Bit 1**	**Bit 0**	
0	0	0	Kein Fehler
0	0	1	Information
0	1	0	Warnung
0	1	1	Fehler
1	0	0	Schwerwiegender Fehler

Tabelle 2.4: Fehlerstatus im Statusbyte

Als 16-Bit Prüfsumme kommt das CRC-16-Polynom $x^{16}+x^{15}+x^2+1$ (0x8005) zur Anwendung. Die Prüfsumme wird in den letzten beiden Byte eines Telegramms gespeichert und umfasst nicht nur die Nutzdaten, sondern auch den Telegrammheader. Details zu einer möglichen Implementierung sind in [SickTL, S. 104f] zu finden.

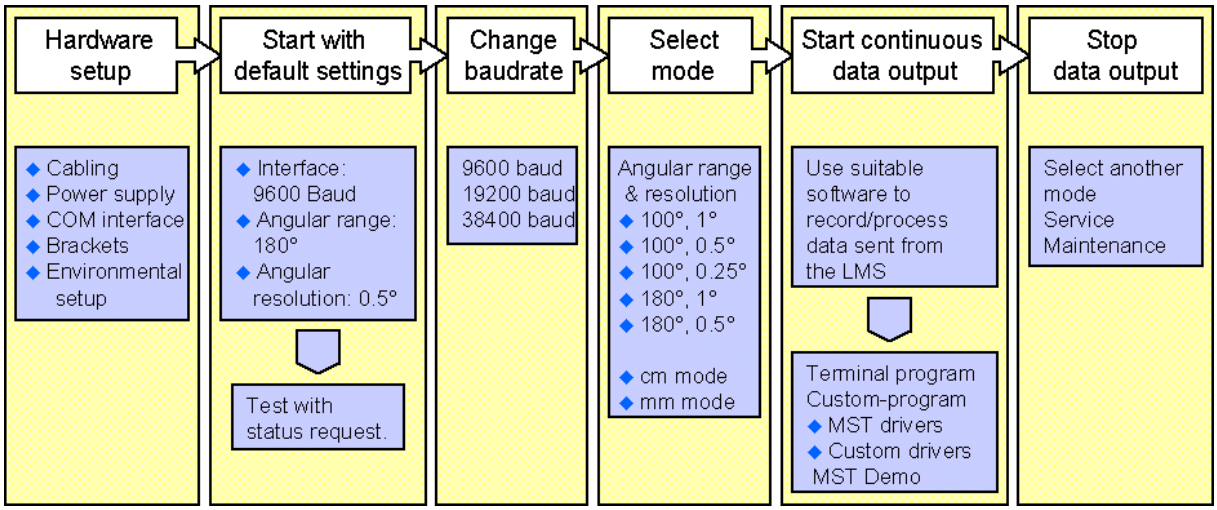


Abbildung 2.7: Schaubild vom Verbindungsaufbau zu Lasermesssystemen [SickQM, S. 9]

2.2.4 Ablauf der Kommunikation

Die Firma SICK gibt in ihrer Dokumentation einen schematischen Ablauf zum Aufsetzen einer Verbindung zum Lasermesssystem an. Die dazugehörige Grafik ist in Abbildung 2.7 zu finden.

Nach dem Einschalten des Lasermesssystems leuchten die gelbe und rote LED während der Initialisierung. Nach spätestens 60 Sekunden ist die Initialisierungsphase abgeschlossen, es leuchtet die grüne oder rote LED, je nachdem, ob eine Bereichsverletzung stattfindet oder nicht. Sollte die gelbe LED blinken, so liegt eine Verschmutzung oder ein Fehler vor.

Das erste Telegramm, das auf der seriellen Schnittstelle vom Lasermesssystem gesendet wird, ist ein *power-on*-Rückmeldetelegramm (0x90). Im Rückmeldetelegramm ist der Typ des Lasermesssystems angegeben, z.B. `LMS200;301063;V02.10`. Das Telegramm wird im *little-endian*-Format übertragen und hat eine Länge von 29 Byte, davon 23 Byte für die Angabe des Typs.

STX	Adr.	Länge	CMD	Nutzdaten							
0x02	0x80	0x17 0x00 23 Byte	0x90	0x4C L	0x4D M	0x53 S	0x32 2	0x30 0	0x30 0	0x3B ;	...

Ein Reset des Lasermesssystems kann mit dem Befehlscode 0x10 veranlasst werden. Es braucht bis zu 60 Sekunden für seine Initialisierung. Aktuelle und vergangene Fehlermeldungen werden dabei gelöscht.

SICK empfiehlt für die Suche nach angeschlossenen Lasermesssystem, die Baudrate auf 9600 zu stellen und den Status des Lasermesssystems abzufragen. Wenn keine Antwort zurückkommt sollte die Baudrate gewechselt und der Schritt so lange wiederholt werden, bis das Lasermesssystem mit einem Telegramm reagiert hat. Für Einsatzzwecke mit Echtzeitdatenübertragung ist es sinnvoller, den Verbindungsaufbau mit einer Baudrate von 500 kBd zu starten, weil ein sich bereits im Betrieb befindendes Lasermesssystem mit großer Wahrscheinlichkeit in diesem Modus ist. Wurde das Lasermesssystem gerade erst eingeschaltet, so muss sowieso bis zu 60 Sekunden

Bereich	Auflösung	Telegramm zum Lasermesssystem
0° ... 100°	1°	02 00 05 00 3B 64 00 64 00 1D 0F
0° ... 100°	0.5°	02 00 05 00 3B 64 00 32 00 B1 59
0° ... 100°	0.25°	02 00 05 00 3B 64 00 19 00 E7 72
0° ... 180°	1°	02 00 05 00 3B B4 00 64 00 97 49
0° ... 180°	0.5°	02 00 05 00 3B B4 00 32 00 3B 1F

Tabelle 2.5: Telegramme zum Setzen des Messbereichs und der Messauflösung

auf Antwort gewartet werden, da das Lasermesssystem vorher noch nicht antworten kann und ein Verbindungsaufbauversuch bei 9600 Bd zunächst auch scheitern würde.

Anstatt gleich mit einem *status request* (0x31) zu überprüfen, ob das Lasermesssystem reagiert, bietet es sich an, mögliche Echtzeitmodi zu deaktivieren. Hier kommt beispielsweise der *operation mode* 0x25 (*output measured value on request only*) in Frage, der herstellerseitig als Standardmodus nach dem Einschalten eingestellt ist. Danach kann ein *status request* (0x31) abgesetzt werden. Dieses Telegramm hat folgenden Inhalt: `02 00 01 00 31 15 12`. Das Antworttelegramm 0xB1, das darauf nach einem ACK-Byte gelesen werden sollte, ist 151 Byte lang und enthält detaillierte Informationen über das Lasermesssystem. Das Lasermesssystem befindet sich jetzt in einem synchronisierten Zustand und wartet auf Befehle. Mit synchronisiertem Zustand ist gemeint, dass das Lasermesssystem von der Anwendungssoftware gefunden wurde und auf Telegramme wartet.

Zur Konfiguration des Lasermesssystems gehört die Einstellung der Baudrate. Das Telegramm *switch operation mode* (0x20) mit dem *sub-command* (0x48) im Datenfeld, zum Wechseln in den *high-speed*-Modus, sieht wie folgt aus: `02 00 02 00 20 48 58 08`. Die dazu gehörige Antwort, die noch mit der alten Geschwindigkeit gesendet wird, sollte `02 80 03 00 A0 00 10 16 0A` sein. Danach empfiehlt es sich mit einem *status request* und dem dazugehörigen *status response* zu überprüfen, ob der Wechsel der Baudrate erfolgreich war.

Das Umkonfigurieren der Messreichweite durch Wechsel zwischen mm- und cm-Modus sollte bewusst verwendet werden, weil dazu das EEPROM[3] neu beschrieben wird und die Anzahl der möglichen Schreibvorgänge im Bereich von einigen tausend liegt. Zudem dauert der Schreibvorgang bis zu sieben Sekunden.

Wenn die Standardmesseinstellungen aus dem EEPROM nicht zutreffen, muss auf den passenden Messwinkel und die passende Schrittweite umgeschaltet werden. Hierbei bietet es sich an, eine temporäre Konfiguration zu wählen. Die Befehle für einen *switch variant* sind in Tabelle 2.5 zu sehen.

Nach der Konfiguration des Lasermesssystem muss auf kontinuierlichen Datenempfang (*all the measured values in a scan continuously*) geschaltet werden. Dazu wird in den *operation mode* 0x24 gewechselt. Das Lasermesssystem sendet jetzt Telegramme (Rückmeldungscode 0xB0) mit kompletten Messdatensätzen. Erst wenn in einen anderen *operation mode* gewechselt wird, hört das Lasermesssystem mit dem Senden auf. Ein Wechsel des *operation mode* kann bis zu drei Sekunden dauern.

[3] *Electrically Erasable Programmable Read-Only Memory*, eine Erläuterung befindet sich in Abschnitt 2.4.2 Seite 31.

Eine komplette Beschreibung aller möglichen Telegramme mit den dazugehörigen Antworttelegrammen ist in [SickTL, Kap. 7] zu finden. Ein Ablaufdiagramm der Kommunikation, für die im Rahmen dieser Arbeit entstandenen Umsetzung, befindet sich in Abbildung 5.5.

2.3 Serielle Datenübertragung (RS-422)

Die serielle Datenübertragung wird in diesem Abschnitt behandelt. Zunächst erfolgt eine Definition der Begriffe und ein Überblick über die verschiedenen Formen der seriellen Datenübertragung. Dabei wird auf die asynchrone serielle Datenübertragung ausführlich eingegangen, da dieser Standard von den SICK-Lasermesssystemen eingesetzt wird. Die Informationen zu den Grundlagen der seriellen Datenübertragung sind [Sta04] entnommen.

Bei der seriellen Datenübertragung werden Daten über nur einen Datenkanal pro Richtung übertragen. Im Gegensatz dazu werden bei der parallelen Datenübertragung mehrere Datenleitungen verwendet. Die Signalelemente werden über die Datenleitung nacheinander übertragen. Im Folgenden werden nur Formen der seriellen Datenübertragung betrachtet, bei dem ein Signalelement genau ein Bit repräsentiert.

2.3.1 Synchronisation des Empfängers

Auf der Empfängerseite muss das Signal zu den korrekten Zeitpunkten abgetastet werden, um die gesendeten Daten zu empfangen. Die Bestimmung dieser Zeitpunkte erfolgt abhängig von der Art der Übertragung, wobei zwei Formen der seriellen Datenübertragung zu unterscheiden sind.

Synchrone serielle Datenübertragung

Bei der synchronen seriellen Datenübertragung wird der Sendetakt über einen zusätzlichen Kanal übertragen. Dies erfolgt in der Regel in Form von Impulsen, zu denen auf Empfängerseite das Signal abzutasten ist. Die Impulse können auch auf Empfängerseite generiert werden, so dass der Sender seine Übertragung zu diesem Takt synchronisieren kann. Bei der synchronen Datenübertragung ist auf der Seite des Empfängers der Zeitpunkt der Abtastung explizit bekannt.

Es gibt auch Spezialformen der synchronen seriellen Datenübertragung, bei der der Sendetakt aus dem Datensignal rekonstruierbar ist (z.B. Manchesterkodierung, siehe [Sta04]). Es entfällt der zusätzliche Kanal zur Taktübertragung. Über einen synchronen seriellen Übertragungskanal können beliebig lange Datenströme übertragen werden. Die Art der Steuerung und Unterteilung des Datenstroms ist von der Anwendung abhängig.

Asynchrone serielle Datenübertragung

Die asynchrone serielle Datenübertragung sieht keine durchgehende Synchronisation von Sender- und Empfängertakt vor und erlaubt nicht das ununterbrochene Senden längerer Datenströme.

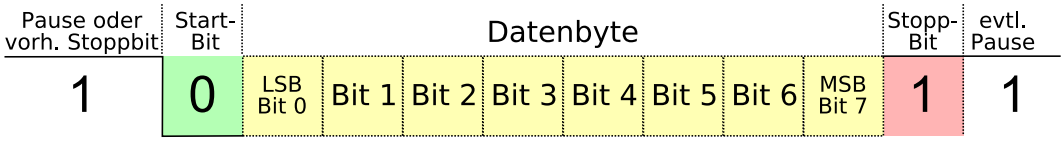


Abbildung 2.8: Ablauf der asynchronen seriellen Datenübertragung

Stattdessen werden nur Blöcke mit einer Länge von je nach Anwendung fünf bis acht Bit übertragen, wobei die Übertragung von acht Bit am weitesten verbreitet ist. Die Synchronisation muss nur für diesen Zeitbereich aufrechterhalten werden und wird bei jeder dieser Sequenzen neu hergestellt. Auf beiden Seiten muss vorab die Übertragungsgeschwindigkeit bekannt sein, die als Baudrate bezeichnet wird. Die Baudrate spezifiziert die Anzahl an Zeichen, die pro Sekunde gesendet werden. Aus diesem Wert ergibt sich die Zeitspanne, in der ein Zeichen vom Sender am Ausgang angelegt werden muss. Auch wenn prinzipiell beliebige Baudraten möglich sind, haben sich einige Werte als Standard durchgesetzt. Diese sind: 300 Bd, 600 Bd, 1,2 kBd, 2,4 kBd, 4,8 kBd, 9,6 kBd, 19,2 kBd, 38,4 kBd, 115,2 kBd, 230,4 kBd, 460,8 kBd und 921,6 kBd. In der Regel werden diese Baudraten von Geräten mit seriellen Schnittstellen unterstützt, wobei die maximal unterstützte Geschwindigkeit variiert, so dass die höheren dieser Baudraten seltener benutzt werden.

Aufgrund der Neusynchronisation nach einer Sequenz von fünf bis acht Bit ist eine gewisse Abweichung der Baudrate zwischen Sender und Empfänger zulässig. Diese beträgt laut [Sta04] theoretisch maximal 5 %. In [Kai93], wo es um die praktische Anwendung der seriellen Schnittstelle geht, wird eine Einhaltung des Taktes mit einer Genauigkeit von 3 % empfohlen.

2.3.2 Ablauf der Übertragung einer Sequenz

Im Folgenden wird der Ablauf der Übertragung einer Sequenz beschrieben. Dieser ist in Abbildung 2.8 grafisch dargestellt. Hier wird der übliche Fall betrachtet, dass pro Sequenz acht Bit übertragen werden.

Erfolgt keine Übertragung, befindet sich die Übertragungsleitung ständig auf dem Pegel, der einer binären Eins entspricht. Eine Übertragung wird mit dem Senden eines Startbit eingeleitet, das einer binären Null entspricht. Die Übertragung kann zu einem beliebigen Zeitpunkt beginnen. Nach dem Startbit werden die acht Datenbit nacheinander gesendet. Das *least-significant*-Bit[4] wird hierbei zuerst übertragen. Nach der Übertragung des letzten Bit folgt eine Zeitspanne, in der sich die Übertragungsleitung im Leerlauf befinden muss, also eine binäre Eins gesendet wird. Die Übertragung nach dem achten Bit wird als Stoppbit bezeichnet und ist typischerweise auf einen Zeitraum von 1, 1,5 oder 2 Zeichen festgelegt. Frühestens nach dem Senden des Stoppbit kann die nächste Übertragungssequenz beginnen. Bei einigen Anwendungen wird zwischen dem achten Bit und dem Stoppbit noch ein Paritätsbit übertragen. Dieses dient zur Fehlererkennung von Einbitfehlern. Für den Fall, dass acht Datenbit, kein Paritätsbit und ein Stoppbit übertragen werden, müssen pro Sequenz zehn Zeichen gesendet werden. Die resultierende Datenrate ist dann 20 % geringer als die Baudrate.

[4] in Abbildung 2.8 mit LSB abgekürzt

Die verschiedenen Parameter (Anzahl der Datenbit, Paritätsbit, Länge des Stoppbit) müssen sowohl auf Sender- als auch auf Empfängerseite bekannt sein, um die korrekte Übertragung zu gewährleisten. Auf der Empfängerseite muss sichergestellt sein, dass jedes Datenbit zum korrekten Zeitpunkt abgetastet wird. Dieser Zeitpunkt sollte am besten genau in der Mitte des übertragenen Zeichens liegen. Von dem Zeitpunkt an, wo auf Empfängerseite ein Startbit erkannt wird, erfolgt die Abtastung der acht Datenbit gesteuert durch den Taktgeber des Empfängers.

UART

Der Begriff UART steht für *Universal Asynchronous Receiver Transmitter*. Hierbei handelt es sich um Komponenten, die folgende Aufgaben erfüllen:

- Wandeln der parallelen Eingangsdaten in einen seriellen Datenstrom
- Versenden des Datenstroms mit festgelegter Baudrate
- Synchronisation einer empfangenen Sequenz
- Wandeln des seriellen Datenstroms in parallele Datenwörter
- Bereitstellen von Statusinformationen über Empfang und Versenden

UARTs können entweder als eigenständiger Baustein zur Anbindung an weitere Hardware ausgeführt sein oder Teil eines Mikrocontrollers sein. Die Taktung der UARTs erfolgt mit einem Vielfachen der Baudrate, üblicherweise dem 8-, 16- oder 32-fachen. Der Zustand der Empfangsleitung kann nur synchron zum Takt überprüft werden. Diese Überabtastung ist notwendig, damit beim Empfangen der Beginn des Startbit genauer erfasst werden kann. Vom Erkennen des Startbit muss bis zum Abtasten des ersten Bit idealerweise 1,5 Schritte gewartet werden, damit der Zeitpunkt der Abtastung genau in der Mitte des Zeichens liegt. Je größer der Faktor der Überabtastung ist, desto genauer liegen die Abtastzeitpunkte in der Mitte der übertragenen Zeichen.

2.3.3 Signalrepräsentation

Die zu übertragenden Zeichen können auf verschiedene Art codiert werden. Es soll hier die Codierung beim RS-232- und RS-422-Standard betrachtet werden. Die Beschreibung der Standards basiert auf [Cat05].

RS-232

Die asynchrone serielle Datenübertragung nach dem RS-232-Standard erlaubt eine Leitungslänge bis 25 Meter bei einer Baudrate von 38,4 kBd. Im Heimcomputerbereich wurde die RS-232-Schnittstelle früher zum Anschluss von Modems, Eingabegeräten, Digitalkameras und weiteren Geräten verwendet. Heutzutage ist die Schnittstelle dort überwiegend durch USB und Netzwerkverbindungen ersetzt. Jedoch ist auch aktuell bei einer großen Anzahl von Geräten diese Schnittstelle für Wartungsaufgaben implementiert:

- Satellitenreceiver (Aktualisierung der Programmliste)
- Telefonanlagen (Konfiguration)

- Server und verschiedene industrielle Geräte (Wartung)

Der Pegel des Signals wird bei RS-232 gegen eine gemeinsame Masse gemessen. Bei der Verkabelung ist eine Masseleitung vorzusehen. Eine negative Spannung zwischen -5 und -15 Volt repräsentiert eine logische Eins, eine Spannung zwischen +5 und +15 Volt eine logische Null. Nach [Kai93] muss beim RS-232-Standard auf der Empfängerseite mindestens noch ein Pegel von ±3 Volt anliegen, so dass erhebliche Spannungsverluste auf der Leitung erlaubt sind.

Auf der Seite des Empfängers hat sich eine Schaltschwelle von etwa +1 Volt als sinnvoll erwiesen, da so auch eine direkte Kommunikation mit TTL- (0 V und 5 V) oder CMOS-Komponenten (0 V und 3,3 V) möglich ist. Es ergibt sich zudem der Vorteil, dass auf Senderseite die Erzeugung einer negativen Spannung, die einen vor allem bei mobilen Geräten unerwünschten Mehraufwand und Mehrverbrauch bedeuten würde, nicht zwingend erforderlich ist. In diesem Punkt wird oft vom Standard abgewichen. Ebenfalls sind mittlerweile Baudraten bis 115,2 kBd üblich, was bei Kabellängen im Bereich von wenigen Metern unkritisch ist.

RS-422

Die serielle Datenübertragung nach dem RS-422-Standard erfolgt durch eine differentielle Übertragung des Signals. Pro Übertragungsrichtung werden zwei Leitungen verwendet, zwischen denen eine Spannungsdifferenz erzeugt wird. Auf der Empfängerseite wird nur die Spannungsdifferenz und nicht der Spannungspegel gegenüber einer Masse ausgewertet. Der Vorteil der differentiellen Übertragung besteht darin, dass sich Störungen auf beide Leitungen gleichermaßen auswirken. Die Spannungsdifferenz bleibt trotz dieser Störungen jedoch nahezu konstant. Bei RS-422 sind Kabellängen bis zu 1,2 Kilometern zulässig. Die Baudrate kann erheblich höher ausgelegt werden.

Auf der Senderseite beträgt die erzeugte Spannungsdifferenz zwischen 2 und 6 Volt. Die Empfänger sind hingegen darauf ausgelegt, Spannungsdifferenzen ab 200 mV sicher auszuwerten, um eine möglichst robuste Übertragung zu gewährleisten. In Abbildung 2.9 sind die zulässigen Spannungsdifferenzen auf Sender- und Empfängerseite gezeigt.

Sowohl die Übertragung nach RS-232 als auch nach RS-422 erfordern auf der Senderseite einen Transmitter-Chip und auf der Empfängerseite einen Receiver-Chip. Es sind ebenfalls sogenannte Transceiver erhältlich, die beide Funktionen vereinen. Diese Komponenten sorgen für eine Anpassung des Spannungspegel eines UART auf das Niveau des gewählten Übertragungsstandards.

2.3.4 Datensicherheit der asynchronen seriellen Datenübertragung

Bei der Realisierung einer Übertragungsstrecke, insbesondere bei hoher Baudrate, sind noch weitere Faktoren zu berücksichtigen. Wie bereits erwähnt, muss sichergestellt werden, dass jedes Datenbit zu einem passenden Zeitpunkt abgetastet wird. Dieser liegt im besten Fall in der Mitte des gesendeten Zeichens. Stimmt der Taktgeber des Empfängers mit dem des Senders nicht überein, verschieben sich die Abtastzeitpunkte. Ab einer Abweichung von 5 % ist nicht mehr gewährleistet, dass die Abtastung des achten Datenbit zu einem Zeitpunkt erfolgt, wo dieses auch tatsächlich gesendet wird.

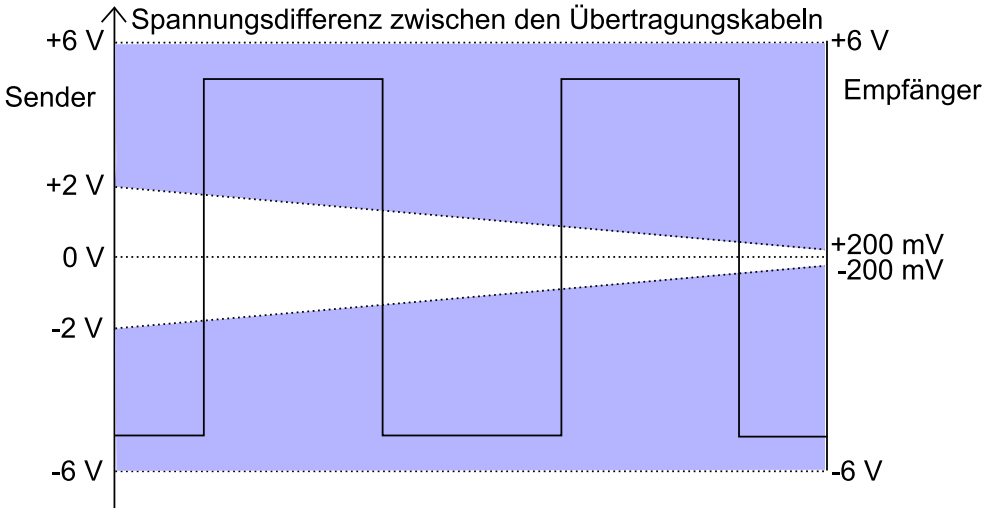


Abbildung 2.9: Spannungsdifferenzen des RS-422-Signals

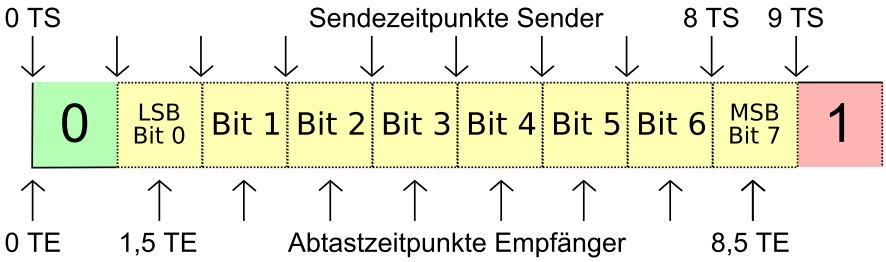


Abbildung 2.10: Ideale Abtastung ohne Abweichung

Es wird im Folgenden hergeleitet, wie sich die theoretisch erlaubte Abweichung bis etwa 5 % ergibt und welche zusätzlichen Gegebenheiten dazu beitragen, dass die tatsächliche Abweichung deutlich niedriger ausfallen sollte. Der Empfänger startet nach dem Erkennen des Startbit zum Zeitpunkt 1,5·TE (Taktdauer Empfänger) die Abtastung des ersten Bit. Zu einem Zeitpunkt von 8,5·TE wird das letzte Nutzdatenbit abgetastet. Auf der Seite des Senders wird das letzte Nutzdatenbit vom Zeitraum 8·TS (Taktdauer Sender) bis 9·TS gesendet. Es ist daher notwendig, dass die Abtastung des letzten Datenbit in diesen Zeitraum fällt, was im Einzelnen bedeutet:

$$8,5 \cdot TE > 8 \cdot TS \quad \rightarrow \quad \frac{TE}{TS} > 0,941$$

$$8,5 \cdot TE < 9 \cdot TS \quad \rightarrow \quad \frac{TE}{TS} < 1,059$$

Hieraus ist ersichtlich, dass eine Abweichung von 6 % zu einer Fehlabtastung führt. Aus Abbildung 2.10 ergibt sich die Bedeutung von TS und TE anhand einer dargestellten idealen Abtastung ohne Abweichung der Taktrate zwischen Sender und Empfänger.

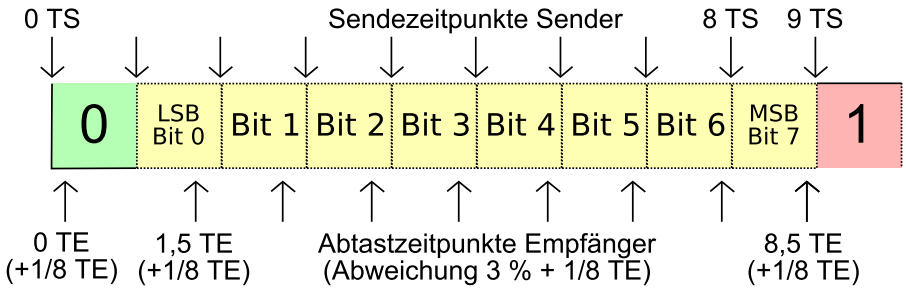


Abbildung 2.11: Erfolgreiche Abtastung mit 3 % Abweichung und initialer Verschiebung

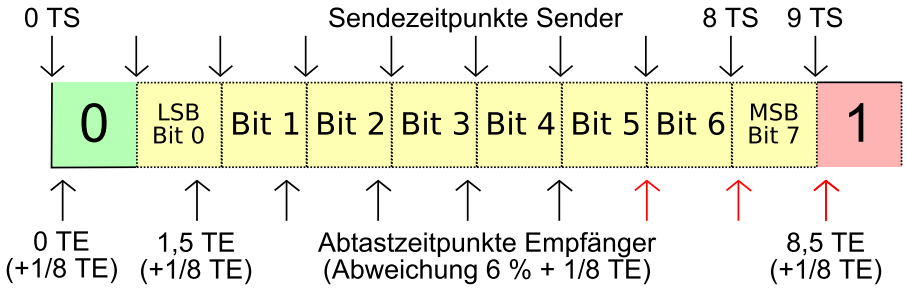


Abbildung 2.12: Fehlerhafte Abtastung mit 6 % Abweichung und initialer Verschiebung

Überabtastung

Ein Faktor, der das Zeitkriterium in der Praxis verschärft, stellt die erläuterte Funktionsweise der UARTs dar, das Signal nur zu diskreten Zeitpunkten abzutasten. Dies führt dazu, dass der Start einer Übertragungssequenz eventuell zu spät erkannt wird und dementsprechend die Abtastzeitpunkte verschoben werden. Es ergibt sich eine verschärfende Auswirkungen auf das Kriterium, dass die Abtastung vor dem Zeitpunkt 9·TS zu erfolgen hat. Für die 8-, 16- und 32-fache Überabtastung sind die Auswirkungen bei der jeweils maximal möglichen Verzögerung errechnet:

8-fache Überabtastung: $\frac{1}{8} \cdot TE + 8,5 \cdot TE < 9 \cdot TS \quad \rightarrow \quad \frac{TE}{TS} < 1,043$

16-fache Überabtastung: $\frac{1}{16} \cdot TE + 8,5 \cdot TE < 9 \cdot TS \quad \rightarrow \quad \frac{TE}{TS} < 1,051$

32-fache Überabtastung: $\frac{1}{32} \cdot TE + 8,5 \cdot TE < 9 \cdot TS \quad \rightarrow \quad \frac{TE}{TS} < 1,055$

Bei 8-facher Überabtastung ergibt sich eine zulässige Abweichung der Taktraten von 4 %. Eine erfolgreiche Abtastung bei der maximal möglichen Verzögerung zur Erkennung des Startbit und einer zusätzlichen Abweichung der Taktrate von 3 % ist in Abbildung 2.11 gezeigt. Abbildung 2.12 zeigt eine fehlgeschlagene Abtastung aufgrund einer Abweichung von 6 % und der gleichen Verzögerung der Startbiterkennung. Die letzten drei Bit werden zum falschen Zeitpunkt abgetastet.

Augenöffnung

Aufgrund der begrenzten Bandbreite des Übertragungskanals und anderer Effekte, die hier nicht näher erläutert werden, kommt es dazu, dass das Signal auf der Seite des Empfängers nicht in der Impulsform ankommt, in der es gesendet wurde. Insbesondere ergeben sich störende Effekte, wenn von einer logischen Eins auf eine logische Null und umgekehrt gewechselt wird. Dieser Vorgang kann nicht in infinitesimal kurzer Zeit erfolgen. Besonders bei sehr hoher Baudrate fällt diese Zeit deutlich ins Gewicht. Die Mitte eines Zeichens weist meist ein eindeutiges Signal auf, während an den Rändern die Störungen besonders stark sind.

In [Kam92] wird das sogenannte Augendiagramm allgemein für Datenübertragungen eingeführt. Ein solches Diagramm ergibt sich aus dem mehrfachen Übereinanderlegen eines stochastischen Datensignals, so dass die idealen Abtastzeitpunkte zusammenfallen. Es ergibt sich je nach Signal ein augenförmiges Diagramm. Anhand der relativen horizontalen Augenöffnung kann der Zeitraum bestimmt werden, in dem eine eindeutige Abtastung des Signals möglich ist. Bei einer relativen Augenöffnung von 1 ist das Signal über den kompletten Zeitraum eindeutig abtastbar.

Bei der Einbeziehung der Augenöffnung in die Berechnungen der maximal zulässigen Taktabweichung müsste ebenfalls noch berücksichtigt werden, dass das Signal nicht nur theoretisch eindeutig abtastbar sein muss, sondern auch der Signalpegel auf einem Niveau liegen muss, bei dem der Empfänger das Signal eindeutig zuordnen kann. Für eine quantitative Bestimmung der Augenöffnung müssten Auswertungen des Signals mit einem Speicheroszilloskop erfolgen. Die Berechnung wird daher nur exemplarisch für eine Augenöffnung von 0,8 bei einer 8-fachen Überabtastung durchgeführt. Der Abtastzeitpunkt muss dann zwischen 8,1·TS und 8,9·TS liegen. Der erkannte Beginn der Übertragung kann sich aufgrund des unbestimmten Signalpegels zwischen zwei Zeichen um $\pm 0,1 \cdot \mathrm{TS}$ verschieben. Der kritische Fall, bei dem die Verschiebung in die ungünstige Richtung erfolgt und sich zusätzlich die Überabtastung auswirkt, ergibt sich zu:

$$0,1 \cdot TS + \frac{1}{8} \cdot TE + 8,5 \cdot TE < 8,9 \cdot TS \quad \rightarrow \quad \frac{TE}{TS} < 1,020$$

In diesem Fall ist eine Abweichung von 2 % schon als kritisch zu betrachten. Wird die Übertragungsgeschwindigkeit weiter gesteigert, verkleinert sich die relative Augenöffnung und die asynchrone serielle Übertragung wird ab einem gewissen Bereich technisch nicht mehr realisierbar.

2.4 Eingebettete Systeme

Der folgende Abschnitt befasst sich mit den Grundlagen der Eingebetteten Systeme (*Embedded Systems*). Hierzu finden sich Informationen in [Moe03] und [Cat05]. Es wird ein Überblick über die Einsatzgebiete der Eingebetteten Systeme, deren Struktur und den Entwurfsprozess gegeben. Ziel dieser Einführung ist es, Realisierungsmöglichkeiten für das zu entwerfende System abzuwägen.

2.4.1 Allgemeines zu Eingebetteten Systemen

Bei Eingebetteten Systemen handelt es sich um informationsverarbeitende Rechnerstrukturen, die in größere Umgebungen integriert sind. Innerhalb dieser Umgebungen ist diesen Rechnerstrukturen eine von vornherein festgelegte Funktionalität zugeordnet. In diesem Punkt liegt der entscheidende Unterschied zwischen Eingebetteten Systemen und Desktop-PC-Systemen, bei denen sich die Funktion ständig entsprechend den Wünschen des Anwenders verändert.

Die Umgebungen, in die Eingebettete Systeme integriert sind, zeichnen sich durch eine sehr starke Heterogenität aus. Es können Aufgaben unterschiedlichster Komplexität vorliegen. Sowohl digitale Daten als auch analoge Signale können zu verarbeiten sein. Viele der Geräte, die im Alltagsleben Anwendung finden, sind mit Eingebetteten Systemen ausgestattet. Einige Beispiele aus den verschiedensten Bereichen sind Mobiltelefone, Digitalkameras, Fernseher, Telefonanlagen, Mikrowellen, Waschmaschinen, Automobile, Flugzeuge, medizinische Geräte und Fahrkartenautomaten. Ebenso können Rechnerstrukturen zur Steuerung von Industrieanlagen als Eingebettete Systeme betrachtet werden (z.B. Steuerung eines Kernkraftwerks).

Bei den einfacheren Eingebetteten Systemen liegt der Aufgabenbereich hauptsächlich in der Ablaufsteuerung und Überwachung. So werden beispielsweise bei Waschmaschinen Heizung, Motor, Frischwasserzufuhr, Abwasserpumpe und Türschloss gesteuert. Zu überwachen sind Wasserstand, Temperatur, bei aufwendigeren Geräten eventuell auch Wäschegewicht, Wasserverschmutzung und Unwucht. Ebenfalls kann die Ansteuerung eines Displays zum Leistungsumfang des Eingebetteten Systems gehören.

Strukturen Eingebetteter Systeme

So heterogen wie die Einsatzgebiete der Systeme kann auch die Struktur der Systeme sein. Es muss jeweils für den speziellen Anwendungszweck abgewogen werden zwischen Leistung, Energieverbrauch, Größe, Entwicklungs- und Produktionskosten bei der prognostizierten Absatzzahl. Bei dem Entwurf kann auf Standardkomponenten (Prozessoren, Controller, DSPs[5]) zurückgegriffen werden oder Hardware eigens für diesen Zweck entwickelt werden. Auch vorgefertigte Systeme, die aus den genannten Standardkomponenten bestehen, können Bestandteile Eingebetteter Systeme sein und bei Bedarf um weitere Komponenten ergänzt werden. Die Systeme können auch aus mehreren Teilsystemen bestehen, denen jeweils ein Teil der Aufgabe zugeordnet ist. Über festgelegte Kanäle tauschen diese Teilsysteme Informationen aus. Als Beispiel seien hier zwei Teilsysteme in einem modernen Fahrzeug genannt: Motorsteuerung und ESP[6]. Sollte vom ESP eine kritische Fahrsituation festgestellt werden, so kann es notwendig sein, die Motorleistung zu verringern. Dieses kann dann dem Steuergerät des Motors mitgeteilt werden.

Eine einheitliche Strukturbeschreibung der Systeme gestaltet sich schwierig, da der Begriff „Eingebettete Systeme“ eine Vielzahl von Systemklassen umfasst. Generell kann jedoch anhand der Entwurfsstrategie eine Unterteilung vorgenommen werden:

- Verwendung eines Prozessors bzw. Mikrocontrollers
- Entwicklung von Spezialhardware (z.B. FPLD[7])

[5] *digital signal processor*, deutsch: digitale Signalprozessoren
[6] Elektronisches Stabilitäts Programm
[7] Field Programmable Logic Device

- Hardware-Software-Partitionierung

2.4.2 Entwurf basierend auf einem Prozessor oder Controller

Der folgende Abschnitt befasst sich mit der Struktur Eingebetteter Systeme, die auf einem Prozessor oder Mikrocontroller basieren. Als „eingebettete PCs“ werden solche Rechnerstrukturen bezeichnet, die einen oder eventuell auch mehrere Prozessoren enthalten und im Gesamtsystem eine spezifische Aufgabe übernehmen. Weitere Bezeichnungen für diese Systeme sind in der Fachliteratur „Industrie-PC“ oder auch „Einplatinen-Computer“. Typisch für solche Systeme ist eine Steckverbindung, über die die Systeme auf einer *embedded*-PC-Trägerkarte (Master) installiert werden. Die Schnittstellenleitungen der eingebetteten PCs werden über die Trägerkarte nach außen geführt oder anderweitig verarbeitet, wobei die Trägerkarte auch anwendungsspezifische Funktionseinheiten enthalten kann.

Eingebettete PCs sind in vielen Ausführungen und Leistungsklassen erhältlich. Diese unterscheiden sich hauptsächlich durch folgende Eigenschaften:

- Prozessor-Architektur
- Einsatzbedingungen (Temperaturbereich, Robustheit, Bauform, Stromverbrauch, Störsicherheit, CE-Zertifizierung)
- Ausstattung und Erweiterbarkeit (Speicherausstattung, Schnittstellen, PCI-Bus, ISA-Bus, PC-Card)
- Verfügbarkeit und langfristige Liefergarantie
- Entwicklung (BIOS-Entwurf, Integrationsfreundlichkeit, Kompatibilität, Zukunftssicherheit)

Eingebettete Systeme weisen häufig eine Standard-PC-Architektur auf. Die Struktur solcher Systeme soll im Rahmen dieser Arbeit nicht beschrieben werden, deshalb wird an dieser Stelle auf [TG01, S. 67ff] oder auf die kompakte Darstellung in [Cat05, Kapitel 1] verwiesen. Als Beispiel für einen eingebetteten PC kann hier der Steuerrechner des Serviceroboters des Arbeitsbereichs TAMS herangezogen werden. Bei diesem System sind ebenfalls die Hauptkomponenten (Prozessor, Speicher, Chipsatz, Grafikcontroller, einige Schnittstellen) auf einer Platine vorhanden. Über die Trägerkarte werden die PCI-Schnittstellen herausgeführt und sind als Steckplätze auf dieser Trägerkarte vorhanden. Somit ergibt sich eine gute Erweiterbarkeit des Systems um Komponenten, die beispielsweise zum Ansteuern der Motoren verwendet werden.

Als Einsatzgebiet für eingebettete PCs kommen vor allem Bereiche in Frage, bei denen es auf hohe Rechenleistung ankommt. Durch die Integration aller notwendigen Komponenten auf einer Platine werden geringe Abmessungen erreicht, so dass die Integration in Geräte erleichtert wird. Der Energieverbrauch kann durch den Einsatz der Stromsparmechanismen moderner Prozessoren eingegrenzt werden, ist aber oft immer noch relativ hoch für mobile Geräte. Durch die freie Wahl des Betriebssystems stehen bei der Anwendungsentwicklung viele Optionen offen.

Bei dem Entwurf weniger aufwendiger Eingebetteter Systeme werden oft Mikrocontroller eingesetzt. Diese vereinen CPU, Speicher und weitere Funktionseinheiten zur Datenein- und -ausgabe auf einem Chip. Beim CPU-Kern wird oft auf das Design vorhandener Prozessoren zurückgegriffen und dieses um Speicher und anwendungsspezifische Peripherie ergänzt. Ein Beispiel ist der Motorola-68000-Prozessor, der in den späteren 80er Jahren Verwendung im Commodore Amiga und Atari ST fand und nun unter der Bezeichnung M68K als Mikrocontroller erhältlich ist. Es

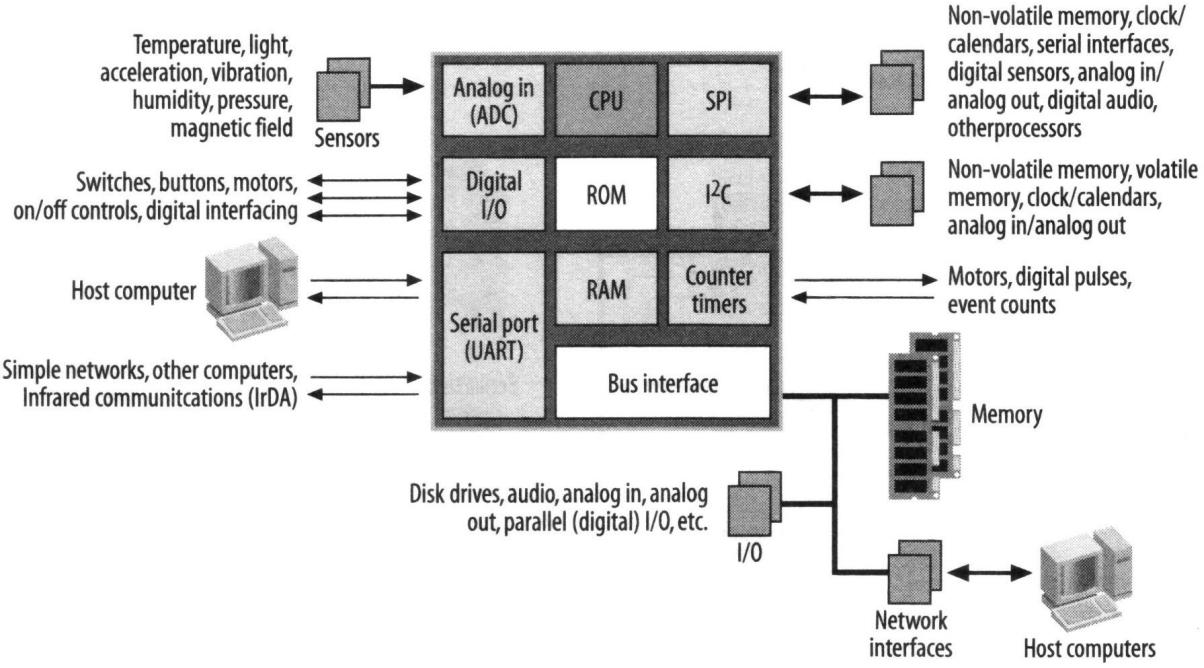


Abbildung 2.13: Blockdiagramm eines Mikrocontrollers, Quelle: [Cat05]

existieren aber auch komplett neu entwickelte Mikrocontroller, die speziell auf hohe Ein- und Ausgabeleistung, niedrigen Stromverbrauch und geringe Produktionskosten optimiert sind. Das Blockdiagramm eines Mikrocontrollers ist in Abbildung 2.13 zu sehen.

Speicher

Die Hauptfunktionseinheiten sind neben dem CPU-Kern RAM (*random access memory*) und ROM (*read only memory*). Beim ROM handelt es sich um nichtflüchtigen Speicher, der zur persistenten Speicherung des Codes verwendet wird. Der Speicher kann in verschiedenen Technologien gefertigt werden, die sich vor allem in der Art der Programmierung und der Integrationsdichte unterscheiden. Einige Exemplare sind nur einmal programmierbar und nicht wieder löschbar. Maskenprogrammierbares ROM wird bei der Chipfertigung über eine spezielle Maske beschrieben. Eine nachträgliche Änderung ist nicht möglich. Für Entwicklungszwecke ist diese Technologie demnach ungeeignet. Soll hingegen ein bereits entwickeltes Gerät in Serienfertigung gehen, so ist ROM-Speicher eine kostengünstige und leistungsfähige Variante.

Beim EPROM (*ereasable programmable ROM*) erfolgt die Programmierung durch das Laden des sogenannten *floating gate*, das einen Teil eines speziellen Transistors darstellt. Dieser Ladevorgang des elektrisch isolierten *floating gate* erfolgt unter Ausnutzung des Tunneleffekts durch Anlegen von höheren Spannungen, so dass einzelne Elektronen die Isolierung durchtunneln. Die Ladungen bleiben dort über mehrere Jahre erhalten. Je nachdem, ob eine Ladung aufgebracht wurde, kann der Transistor schalten oder nicht. Der Programmiervorgang ist durch Bestrahlen des Chips mit UV-Licht reversibel, da den Elektronen durch Licht dieser Wellenlänge ausreichend Energie zugeführt wird, um die Isolierung erneut zu überwinden. Es wird dabei immer

der gesamte Chip gelöscht. Diese Technik ist für Entwicklungszwecke grundsätzlich geeignet. Der Löschvorgang erfordert jedoch eine spezielle UV-Lampe und ist umständlich und zeitaufwändig. Eine spätere Pflege des Codes in ausgelieferten Geräten wäre aufgrund der Programmiertechnik nicht sinnvoll realisierbar.

Bei der EEPROM-Technologie (*electrically ereasable programmable ROM*) erfolgt der Löschvorgang durch Anlegen einer Spannung. Die Durchtunnelung findet nun in entgegengesetzter Richtung statt. Somit kann der Löschvorgang auch selektiv erfolgen, so dass nicht der gesamte Chip gelöscht werden muss. Zum Beschreiben muss der Chip nicht aus der Schaltung ausgebaut werden. Das Einspielen eines aktuellen Codes gestaltet sich relativ einfach. Ein Beispiel für die Anwendung dieser Technik außerhalb von Mikrocontrollern ist das BIOS der Standard-PCs, das vom Anwender über entsprechende Software stets auf dem neusten Stand gehalten werden kann.

Auf einem Mikrocontroller steht in der Regel zusätzlich ein Arbeitsspeicher (RAM) zur Verfügung. In diesem flüchtigen Speicher werden Daten der aktuellen Programmausführung abgelegt. Üblicherweise wird ein Mikrocontroller verwendet, um analoge oder digitale Signale aufzunehmen, zu verarbeiten und entsprechend weiterzuleiten oder Entscheidungen aufgrund der aufgenommenen Signale zu treffen. Für diese Schritte ist die Zwischenspeicherung der Daten im Arbeitsspeicher unumgänglich.

Schnittstellen

Ein Mikrocontroller ist üblicherweise mit verschiedenen Schnittstellen ausgestattet. Je nach Modell können die im Folgenden kurz vorgestellten Schnittstellen vorhanden sein.

Sehr weit verbreitet sind bei Mikrocontrollern die sogenannten *general-purpose-input/output*-Schnittstellen (GPIO). Es handelt sich um einzeln konfigurierbare Ein- und Ausgänge. Als Eingang konfiguriert, kann der Zustand von Schaltern und Tastern überwacht werden. Wird ein Pin im Ausgangsmodus betrieben, so können über entsprechende Elektronik beliebige Verbraucher gesteuert werden. Des Weiteren kann der Pin zur Ausgabe eines Zustandes benutzt werden, der dann einem anderen Gerät mitgeteilt wird.

Oft kommen bei Mikrocontrollern auch Analog-digital-Wandler zum Einsatz. So können die Signale beliebiger Sensoren ausgewertet werden (z.B. Lichtstärke, Temperatur, Beschleunigung). Digital-analog-Wandler erzeugen aus einem digitalen Wert eine Spannung, die zum Steuern elektronischer Bauteile genutzt werden kann.

Serielle Anschlüsse ermöglichen die Verbindung zu Hostrechnern, Modems oder beliebigen anderen Geräten mit serieller Schnittstelle. Diese Schnittstelle ist in Kapitel 2.3 ausführlich beschrieben. Auf einigen Mikrocontrollern sind auch die spezialisierten Formen der seriellen Schnittstelle SPI und I^2C vorhanden.

Über das *Serial Peripheral Interface* (SPI) kann Peripherie an den Mikrocontroller angebunden werden. Es handelt sich um eine synchrone serielle Übertragung. Der Takt wird von dem Gerät bereitgestellt, das als Master konfiguriert ist, also typischerweise vom Mikrocontroller. Auf diese Art werden Funktionseinheiten wie Echtzeituhren mit Timer- und Kalenderfunktion angebunden. Ebenfalls können serielle Flash-Speicher zur persistenten Speicherung angebunden werden.

Der I^2C-Bus dient zur Anbindung mehrerer adressierbarer Geräte an ein Bussystem. Von diesen Geräten können mehrere die Funktion des Masters übernehmen. Der Einsatz dieses Bussystems kann beispielsweise zur Anbindung von Displays dienen.

Viele neuere Mikrocontroller implementieren auch USB-Schnittstellen. Dadurch wird der Datenaustausch mit neueren Hostrechnern ohne serielle Schnittstelle ermöglicht. Des Weiteren bietet USB gegenüber der seriellen Schnittstelle weniger Fehlerquellen in der Verkabelung und weist eine hohe Kompatibilität zwischen den einzelnen Geräten auf. Sogenanntes „Plug and Play“, das Verbinden von Geräten während des Betriebes, ist bei USB-Verbindungen problemlos möglich.

Größere Mikrocontroller führen manchmal auch weitere Bus-Leitungen nach außen. Über diese können dann je nach Kompatibilität beliebige Komponenten angebunden werden. Dabei kann es sich beispielsweise um Ethernet-Controller, zusätzliche RAM-Speicherchips oder auch Anbindungen für Festplatten und Flash-Speicher handeln.

Vor- und Nachteile des Entwurfs basierend auf Prozessor oder Controller

Ein großer Vorteil des Entwurfs basierend auf Prozessoren oder Mikrocontrollern sind die geringen Entwicklungskosten. Es sind viele vorgefertigte Lösungen für gängige Aufgabenstellungen vorhanden, die mit geringem Aufwand an die Anwendung angepasst werden können. Die Programmierung kann meist in einer Hochsprache wie C erfolgen, wodurch die Entwicklungszeit und somit die *time to market* erheblich verkürzt wird. Üblicherweise werden umfangreiche Bibliotheken für Standardfunktionen mitgeliefert. Auf größeren Mikrocontrollern oder Prozessoren können Betriebssysteme eingesetzt werden, so dass die Verwaltung der verschiedenen Prozesse ebenfalls nicht vom Entwickler gesteuert werden muss.

Die Rechenleistung und die Echtzeitfähigkeit solcher Systeme sind für einige Anwendungsbereiche nicht ausreichend. Die Prozessoren und Controller sind für eine breite Menge von Anwendungen ausgelegt. Zwar können alle Aufgaben prinzipiell erfüllt werden, jedoch bedingt diese Vielseitigkeit, dass der Optimierung auf spezielle Algorithmen Grenzen gesetzt sind. Des Weiteren existiert nicht die Möglichkeit der Parallelverarbeitung. Pro Prozessor kann nur ein Befehlsstrom zur Zeit abgearbeitet werden. Wenn in einem System beispielsweise verschiedene Datenströme nach einem vorgegebenen Muster verarbeitet werden (z.B. Videodatenströme), müssen diese nacheinander vom Prozessor bedient werden. Bei Mikrocontrollern kommt noch der entscheidende Nachteil hinzu, dass Datentransferoperationen stets über den CPU-Kern erfolgen müssen. Bei aufwendigeren Systemen werden diese oft durch einen eigenen Controller gesteuert, so dass der Hauptcontroller, besonders beim Kopieren größerer Blöcke, entlastet wird.

2.4.3 Entwurf von Spezialhardware

Bei dem Entwurf von Spezialhardware wird für einen Aufgabenbereich spezielle Hardware entwickelt. Diese Hardware ist dann nur für eine bestimmte Anwendung nutzbar, dafür ist sie aber hoch optimiert. Anforderungen an die Leistungsfähigkeit und die Echtzeitfähigkeit können so erfüllt werden. Bei der Entwicklung stehen hierbei verschiedene Möglichkeiten der Realisierung zur Wahl. Diese unterscheiden sich vor allem durch die Ebene, auf der entwickelt wird, und durch die Technologie, die im späteren Gerät zum Einsatz kommen soll.

Typische Entwurfsstile sind nach [RC89] folgende:

- Voll-kundenspezifischer Entwurf (*full custom design*)
- Zellenentwürfe
- Entwurf mit Gate-Arrays
- Entwurf mit programmierbaren Schaltungen

Die verschiedenen Entwurfsstile weisen bezüglich Integrationsdichte, Leistungsfähigkeit, Entwicklungsaufwand, Fertigungszeit und Stückkosten erhebliche Unterschiede auf.

Voll-kundenspezifischer Entwurf

Beim voll-kundenspezifischen Entwurf werden die Schaltungen komplett nach den Spezifikationen des Kunden gefertigt. Die Platzierung jedes einzelnen Transistors auf dem Chip kann vom Entwerfer frei bestimmt werden. Auf diese Art kann die höchstmögliche Integrationsdichte und Leistungsfähigkeit erreicht werden. Allerdings entstehen durch das Anfertigen des Konstruktionsplans und entsprechender Masken zur Chipherstellung erhebliche Entwicklungskosten. Nur bei entsprechend hohen Stückzahlen sinkt der Stückpreis auf ein akzeptables Niveau. Bei Spezialanwendung werden jedoch oft nur geringe Stückzahlen hergestellt. Ein weiterer Nachteil ist, dass durch den hohen Entwicklungsaufwand die *time to market* ungünstig ausfällt und somit das Produkt erst Marktreife erlangt, wenn sich möglicherweise am Markt schon Veränderungen ergeben haben.

Zellenentwürfe

Zellenentwürfe ähneln den voll-kundenspezifischen Entwürfen. Beim Chiphersteller erfolgt ebenfalls die Fertigung eines Chips, der komplett nach Kundenwunsch aufgebaut ist. Die dadurch entstehenden Kosten und Produktionszeiten sind mit denen eines voll-kundenspezifischen Entwurfs vergleichbar. Der Entwurfsprozess wird jedoch durch den Einsatz von Zellen stark vereinfacht. Bei den Zellen handelt es sich um bereits entwickelte Funktionsblöcke, die frei auf dem Chip platziert und mit weiteren Zellen verbunden werden. Es ist zwischen Makrozellen und Standardzellen zu unterscheiden:

- Makrozellen haben unterschiedliche Größe, können wiederum aus einfacheren Zellen aufgebaut sein, die Entwicklung erfolgt oft spezifisch für einen Entwurf.
- Standardzellen haben gleiche Höhe und ähnliche Größe, Aneinanderreihung von Zellen üblich, Zellen basieren auf vordefinierter Zellenbibliothek

Entwurf mit Gate-Arrays

Beim Entwurf mit Gate-Arrays werden Chips verwendet, die eine regelmäßige Anordnung von Transistoren aufweisen. Es handelt sich um Standardbausteine, die in großen Stückzahlen vorgefertigt werden können. Sie enthalten je nach Typ neben den Transistoren verschiedene Ein- und Ausgabe-Zellen. Die Verschaltung dieser Funktionseinheiten erfolgt nach Vorgabe des Kunden. Realisiert wird diese Anpassung der vorgefertigten Standardbausteine beim Hersteller durch einen Belichtungsprozess mit einer Maske. Die Fertigungsdauer wird dadurch gegenüber dem voll-kundenspezifischen Entwurf verkürzt.

Die Integrationsdichte und Leistungsfähigkeit erreicht nicht die eines voll-kundenspezifischen Entwurfs, ist aber deutlich höher als die von programmierbaren Schaltungen. Der Entwicklungsprozess ist immer noch relativ aufwändig, da für Entwurf und Test der zu entwickelnden Chips in der Regel teure Spezialhard- und Software erforderlich ist. Die Fertigung einer Maske zum Belichten ist ebenfalls kostenaufwändig. Der Stückpreis liegt bei mittleren Absatzzahlen auf einem günstigen Niveau.

Als problematisch stellt sich bei diesem Vorgehen die Tatsache dar, dass der komplette Entwicklungsplan, der das Ergebnis eines kostenaufwändigen Entwicklungsprozesses darstellt, dem Chiphersteller offengelegt werden muss. Dieses ist in vielen Bereichen nicht gewünscht.

Entwurf mit programmierbaren Schaltungen

Beim Entwurf mit programmierbaren Schaltungen werden die Chips komplett beim Kunden an ihre späteren Aufgabe angepasst. Es wird zwischen einmal programmierbaren und mehrfach programmierbaren Exemplaren unterschieden.

Die einmal programmierbaren haben eine höhere Integrationsdichte als die mehrfach programmierbaren, jedoch liegt diese unter der von maskenprogrammierten Gate-Arrays. Technisch realisiert ist diese Programmierung durch die Fusable-Link- oder die Antifuse-Technologie. Bei der Fusable-Link-Technologie sind in den vorgefertigten Chips Verbindungen vorhanden, die gezielt durch Anlegen einer Spannung beim Programmiervorgang selektiv geschmolzen werden. Auf diese Weise wird eine anwendungsspezifische Verschaltung erreicht. Bei der Antifuse-Technologie werden hingegen leitende Verbindungen durch Anlegen der Programmierspannung irreversibel hergestellt.

Mehrfach programmierbare Logikbausteine können in EPROM-, EEPROM- oder SRAM-Technologie realisiert werden. Die (E)EPROM-Technologie ist am Anfang dieses Abschnitts bereits erläutert. Bei der SRAM-Technologie ist in dem Chip ein flüchtiger Speicher eingebracht, der den Zustand der Verbindungen steuert, so dass sich dieser Chip entsprechend den Vorgaben verhält. Wenn diese Technologie in Geräte für einen Endanwender integriert werden soll, so ist die Schaltungsbeschreibung persistent in einem zusätzlichen nichtflüchtigen Speicher bereitzustellen, da der Chip nach jedem Abschalten der Stromversorgung die Funktion verliert. Für Entwicklungszwecke und Kleinserien ist diese Technologie sehr gut geeignet, bei größeren Auflagen ist der Stückpreis mit den vorher beschriebenen Technologien auf einem günstigeren Niveau. Der Fertigungszeit der Chips fällt kurz aus, da sich der Prozess auf das Programmieren schon vorgefertigter Chips beschränkt.

Aufgrund ständig wachsender Integrationsdichte steigt die Leistungsfähigkeit entsprechender Bausteine. Es können immer mehr Funktionseinheiten mit größerer Geschwindigkeit integriert werden. Bei vielen dieser Bausteine können sogar synthetische CPUs und IP-Komponenten realisiert werden. Einige FPGAs[8] besitzen auch „fertige“ eingebettete CPU-Kerne.[9] Zusätzlich können anwendungsspezifische Elemente auf dem Chip implementiert werden, so dass Flexibilität und Leistungsfähigkeit verbunden werden mit der Universalität eines Prozessors und den Funktionen eines Betriebssystems.

[8] Field Programmable Gate Array
[9] Altera Excalibur, ARM-Prozessor mit 200 MHz, www.altera.com

2.4.4 Entwurf mittels Hardware-Software-Partitionierung

Beim Entwurf mittels Hardware-Software-Partitionierung wird versucht, die Vorteile der zuvor beschriebenen Entwurfstechniken zu vereinen. Diese sind beim Entwurf von Spezialhardware die Performance und die Echtzeitfähigkeit. Beim Entwurf mittels Prozessor oder Mikrocontroller ist der Vorteil die Kosteneffizienz, bei der Entwicklung von Spezialhardware die Leistungsfähigkeit und die Echtzeitfähigkeit.

Beim Entwurf wird nicht von vornherein festgelegt, welche Systemteile als Software auf einem Prozessor ausgeführt werden und welche als eigene Hardware zu realisieren sind. Diese Entscheidungen werden vielmehr während des Entwurfs getroffen, wobei das Abwägen von Alternativen ebenfalls Bestandteil der Entwurfstechnik ist.

Beim Entwurf wird die Funktionalität des Systems partitioniert, so dass verschiedene Softwareteile auf unterschiedlichen Prozessoren oder durch spezielle Funktionseinheiten ausgeführt werden. Die einzelnen Partitionen müssen bezüglich ihrer Anforderungen klassifiziert werden, so dass für sie ein Prozessor vorgesehen wird, der die notwendige Leistungsfähigkeit aufweist und dabei nicht mehr Kosten verursacht als notwendig.

Im weiteren Verlauf des Entwurfs ist ein genauer Ablaufplan hinsichtlich der Funktionen innerhalb des Gesamtsystems aufzustellen. Auf diese Art kann die Korrektheit des Entwurfs bezüglich Timinganforderungen überprüft werden. Letztendlich erfolgt die Bindung, die jeder Variablen eine Speicherzelle und jeder Funktion eine Funktionseinheit zuweist.

2.4.5 Die Hardwarebeschreibungssprache VHDL

VHDL steht für VHSIC[10] *Hardware Description Language* und wird oft bei der Entwicklung von anwendungsspezifischer Spezialhardware eingesetzt. Diese Sprache ermöglicht eine eindeutige Dokumentation eines Hardwareentwurfs und zeichnet sich durch Leistungsfähigkeit und leichte Verständlichkeit aus. Bestehende Entwürfe können erweitert oder angepasst werden, so dass vorhandene Entwicklungsergebnisse weiterverwendet werden können. Der Entwurfsprozess von Spezialhardware führt mit Hilfe von VHDL und Synthesewerkzeugen schneller zu einem fertigen Produkt als beim klassischen Full-Custom-Entwurf. Die integrierten Schaltungen können vor der Fertigung vollständig simuliert werden. Eventuelle Fehler sind somit frühzeitig korrigierbar. Je nach eingesetzter Technologie ist das Ergebnis des Entwurfsprozesses eine Datei zur Programmierung des Chips oder ein entsprechender Plan zur Fertigung der Chips beim Hersteller.

Eine Ursache für die Notwendigkeit des Einsatzes einer Hardwarebeschreibungssprache ist die immer größere Komplexität der Chips. Diese liegt begründet in der ständig steigenden Integrationsdichte durch technologischen Fortschritt sowie in der wachsenden Nachfrage des Marktes nach entsprechenden Produkten.

Der Entwurf in VHDL erfolgt in der Regel auf der Register-Transfer-Ebene. Ein System wird durch seine Register und durch die Art, wie sich der Zustand der Register ändert, beschrieben.

[10] *Very High Speed Integrated Circuit*

Die Zustandsänderung wird durch eine Funktion angegeben, nach der sich der neue Registerinhalt ergibt, und durch eine Bedingung (z.B. eine Taktflanke), bei deren Eintreten die Änderung durchgeführt wird. Das Entwicklungssystem generiert aus diesen Angaben weitgehend selbstständig die darunterliegenden Ebenen, so dass der Entwicklungsaufwand in diesen Ebenen maßgeblich erleichtert wird.

Hauptmerkmal eines VHDL-Entwurfs ist die Unterteilung in die Einheiten „Entity“ und „Architecture“, die zum Beschreiben einer Komponente dienen. In der „Entity“ sind die Schnittstellen einer Komponente beschrieben. Das Verhalten der Komponente ist in der „Architecture“ beschrieben. Es besteht die Möglichkeit, eine hierarchische Beschreibung des Hardwareentwurfs vorzunehmen, also in die Beschreibung einer Komponente eine weitere Komponente einzubetten. Auf diese Art ist ein Top-Down-Entwurf eines Systems möglich: Auf der obersten Schicht ist eine in der Regel überschaubare Anzahl Funktionsblöcke vernetzt, deren Implementierung dann in der entsprechenden Beschreibung der jeweiligen Komponente zu finden ist. Diese stellen dann eventuell wiederum eine Netzliste von Komponenten dar.

Es ergibt sich so bei der Entwicklung die Möglichkeit, häufig benutzte Komponenten (Speicher, ALUs[11], etc.) ohne weiteren Entwicklungsaufwand einzubinden. Speziell für den Fall, dass Komponenten in mehreren Designs zum Einsatz kommen, bietet sich die Verwendung der Einheiten „Configuration“ und „Package“ an. Über die „Configuration“ lassen sich verwendete Komponenten parametrisieren. Es ist also nicht notwendig, für jedes mögliche Derivat der Komponente eine eigene Beschreibung vorzusehen. In einem „Package“ sind Deklarationen verschiedener Typen und Komponenten zusammengefasst, so dass beim Benutzen eines Packages die enthaltenen Komponenten in das Design eingebunden werden können.

2.5 Integration in den Serviceroboter

Nachdem in den vorherigen Abschnitten dieses Kapitels das SICK-Lasermesssystem und die serielle RS-422-Schnittstelle vorgestellt wurden, folgt in diesem Abschnitt zunächst eine Beschreibung der Systemhardware. Es werden die Hardwarekomponenten des mobilen Serviceroboters näher erläutert, die für diese Arbeit von Bedeutung sind. Nach der Hardwarebeschreibung folgt ein Abschnitt über die Software, die auf dem mobilen Serviceroboter läuft. Software aus höheren Schichten, die auf der Softwareschnittstelle aufsetzt, wird nur am Rande erwähnt. Als letztes folgt eine Auswertung der Software-Logausgaben. Die Auswertung zeigt detailliert die Probleme, die bei der Anbindung der Lasermesssysteme auftreten.

2.5.1 Systemhardware

Der mobile Serviceroboter hat zwei Antriebsräder mit einem Abstand von ungefähr 60 Zentimetern. Sie lassen sich einzeln steuern, so kann der Roboter beliebige Kurven fahren und sich auf der Stelle drehen. Die beiden Motoren sind über einen CAN-Bus[12] an den Hostrechner im Inneren des Roboters angebunden. Der Computer verfügt über einen Intel-Pentium-4-Prozessor mit 2,4 GHz und 512 MB Arbeitsspeicher. Als Betriebssystem läuft ein aktueller Linux-2.4-Kernel

[11] *Arithmetic Logical Unit*, Rechenwerk für arithmetische und logische Funktionen
[12] *Controller Area Network* – ein asynchrones, serielles Bussystem

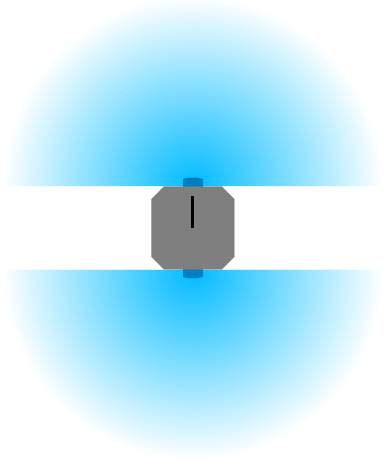


Abbildung 2.14: Toter Winkel der Lasermesssysteme am Serviceroboter

unter SuSE Linux 9.0. Zur Stromversorgung von Motoren und Computer befinden sich Bleiakkus im Roboter, die eine Nennspannung von 48 Volt liefern.

Wie bereits erwähnt verfügt der Roboter, damit er nicht gegen Hindernisse (insbesondere Menschen) fährt, zur Abstandsmessung vorne und hinten jeweils über ein Lasermesssystem vom Typ LMS 200. Die Lasermesssysteme messen die Entfernung in der horizontalen Ebene und sind 33,6 Zentimeter vor und hinter dem Robotermittelpunkt in einer Höhe von ungefähr 21 Zentimetern montiert. Die Lasermesssysteme decken jeweils einen Bereich von 180 Grad ab. Dadurch, dass sie einen Abstand von 67,2 Zentimeter voneinander haben, entstehen auf den beiden Seiten des Roboters tote Winkel, in denen keine Abstandsmessungen durchgeführt werden können. Grafisch dargestellt ist dies in Abbildung 2.14. Flache Hindernisse, die unter der Messebene liegen, also weniger als 20 Zentimeter hoch sind, können von den Lasermesssystemen nicht erkannt werden. In den Raum ragende Objekte oder Tische können nur erkannt werden, wenn sie zum Beispiel „Beine“ haben.

Angebunden sind die SICK-Lasermesssysteme über eine serielle Schnittstellenkarte der Firma Moxa. Diese Karte verfügt über zwei RS-422-Schnittstellen. Der Roboterhersteller Neobotix hat diese Karte in einer angepassten Version mitgeliefert, damit die Schnittstelle mit einer Baudrate von 500 kBd betrieben werden kann. Die Lasermesssysteme benötigen genau diese Geschwindigkeit, um in Echtzeit kontinuierlich Messdaten zu senden. Ursprünglich unterstützt die Karte nur eine Baudrate von 921.600, 460.800 und niedriger. Durch den Austausch des Quarzes kann die Karte jedoch mit einer Baudrate von 500.000 betrieben werden.

Auf dem Roboter befinden sich noch weitere Sensoren und Aktuatoren (siehe Abbildung 1.1). Da es in dieser Arbeit jedoch um die Anbindung von Lasermesssystemen geht, werden diese Sensoren und Aktuatoren nicht weiter behandelt.

2.5.2 Software-Schnittstelle

Die Software zur Steuerung des Roboters besteht aus mehreren Komponenten, die aufeinander aufbauen. Die unterste Ebene bildet der `mobiled`[13]. Dabei handelt es sich um ein in C++ geschriebenes monolithisches Programm zur Steuerung der Motoren und der Lasermesssysteme. Zusätzlich zur Hardwareansteuerung ist eine Lokalisation auf Basis eines Kalman-Filters[14] integriert. Die Lokalisation verwendet die Messwerte der Lasermesssysteme und die durch Odometrie[15] gewonnenen Information der beiden angetriebenen Räder.

Die serielle Schnittstellenkarte für die beiden Lasermesssysteme benötigt einen eigenen Treiber in Form eines Linux-Kernelmoduls. Dieses Kernelmodul `mxser` wird vom Hersteller Moxa im Quelltext zur Verfügung gestellt. Es muss für den aktuellen Kernel eigenhändig kompiliert werden. Die Daten der RS-422-Schnittstelle werden dann über die *character devices* `/dev/ttyM0` und `/dev/ttyM1` zur Verfügung gestellt. Zum Vergleich, die Standard-Schnittstellen sind bei Linux unter `/dev/ttyS0`, `/dev/ttyS1`, ... zu finden.

Außerdem bietet der `mobiled` die Möglichkeit, Graphen für die Batteriespannung und für die Motortemperatur mit Hilfe von `gnuplot` anzuzeigen. Die vorgefertigten Startskripte deaktivieren jedoch die Anzeige der Graphen.

Der `mobiled` lässt sich über den TCP/IP-Port 9002 steuern und kontrollieren. Es können Steuerkommandos gesendet, Parameter verändert oder Sensordaten (z.B. Lasermesswerte, Motortemperatur, ...) abgefragt werden. Der Roblet-Server[16] `uhh.fbi.tams.mobilerobot` greift auf den zu Grunde liegenden `mobiled` über eine solche TCP-Verbindung zu.

Umgesetzt sind die einzelnen Teilaufgaben im `mobiled` in Threads. In Abbildung 2.15 ist der Zusammenhang der Threads und deren Aufgaben dargestellt. Als erstes wird der TCP-Port 9002 gebunden. Danach wird ein CGENBASE-Objekt erzeugt, das einen eigenen Thread benutzt. Der Hauptthread wartet anschließend nur noch auf eingehende TCP-Verbindungen und lässt dann gegebenenfalls vom CGENBASE-Thread einen neuen CCLIENT-Thread erstellen und reicht die ankommende Verbindung weiter. Diese CCLIENT-Verbindung kommuniziert innerhalb des `mobiled` ausschließlich mit dem CGENBASE-Thread.

Im CGENBASE-Thread wird ein CMOTORFEEDER- und ein CLASERFEEDER-Objekt erzeugt. Der CVOLTAGEDISPLAYTHREAD wird normalerweise nicht erzeugt. Alle drei sind eigenständige Threads. Auf den CVOLTAGEDISPLAYTHREAD und den CMOTORFEEDER-Thread wird in dieser Arbeit nicht näher eingegangen, sie spielen für die Anbindung der Lasermesssysteme praktisch keine Rolle. Die gesamte Kommunikation zwischen den Threads geht dabei immer über den CGENBASE-Thread.

Der CLASERFEEDER-Thread erzeugt für jedes Lasermesssystem einen weiteren Thread (CLASER). Nach Außen (zum CGENBASE) werden folgende Methoden bereitgestellt:

- `int GetClosestObstacleDistance(const int idx, double &distance)`
- `int GetLaserScanScanner(const int idx, CRADIALSCAN &scan)`
- `int GetLaserScanPlatform(const int idx, CRADIALSCAN &scan)`

[13] Der Name leitet sich von *mobile daemon* ab.
[14] Näheres zur Lokalisation mit Kalman-Filtern ist in [SW02] zu finden.
[15] Durch Beobachtung der Räder kann auf die gefahrene Strecke geschlossen werden.
[16] Ein in Java geschriebenes Programm zur Nutzung des Roboters in einem verteilten System.

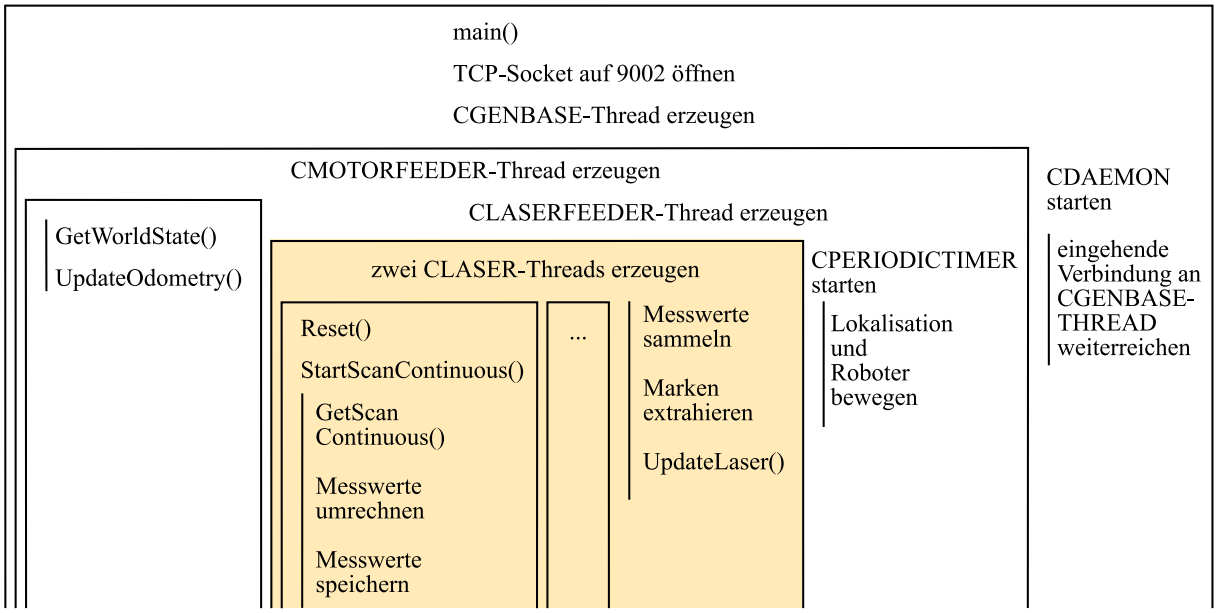


Abbildung 2.15: Threadstruktur der Moxa-Anbindung

- `int GetLaserScanPlatformMatched(const int idx, CRADIALSCANMATCHED &scan)`
- `int GetNumScanners(void) const`
- `int GetScannerPosition(meter_t &x, meter_t &y, radian_t &a, const int idx) const`

Hierbei ist `idx` immer der Index, der angibt, von welchem Lasermesssystem die Messdaten abgefragt werden. Das Objekt CRADIALSCAN kapselt einen Messdatensatz von einem Lasermesssystem. Ein `float`-Array `*_scanAngle` gibt den Winkel im Bogenmaß für die einzelnen Messwerte an. Im `*_scanDist`-Array sind die dazugehörigen Entfernungen ebenfalls als `float` in Metern angegeben. Die Anzahl der Messwerte ist in `_numScans` als Integer gespeichert. Die Lasermesssysteme liefern die Messwerte in festen Gradschritten, so dass theoretisch auf die Angabe eines Arrays mit Winkeln verzichtet werden kann. Da das CRADIALSCAN-Objekt auch als Datentyp für umgerechnete Datensätze verwendet wird, bei denen der Winkel variiert, müssen die Winkel mit gespeichert werden. Außerdem kann die Anzahl der Messwerte pro Messung variieren, da ab und zu vom Lasermesssystem keine Entfernung für einzelne Winkel ermittelt werden kann. Als Ursache dafür ist ein zu kleiner Signal-Rausch-Abstand[17] oder eine Blendung des Lasermesssystem wahrscheinlich.

Zu einem Messdatensatz gehören neben den Entfernungen auch gefundene Reflektormarken. Die dazugehörigen Werte sind ebenfalls in `float`-Arrays (`*_markAngle` und `*_markDist`) gespeichert. Die Anzahl der gefundenen Reflektormarken steht in `_numMarks`. Das Objekt CRADIALSCANMATCHED erbt von CRADIALSCAN. Es speichert zusätzlich zu den gefundenen Reflektormarken eine Marken-ID in einem Integerarray `*_markIdx`. Diese Marken-ID ist nur gesetzt, wenn die gemessene Marke mit einer Marke aus der Lokalisationskarte übereinstimmt. Für die Anbindung von Lasermesssystemen spielt sie keine Rolle.

[17] Das Verhältnis zwischen Signal und Rauschen ist zu klein.

Die Methode `GetClosestObstacleDistance()` ist für die Kollisionsvermeidung gedacht, da sie zurückgibt, wie weit der Robotermittelpunkt vom nächstgelegenen Hindernis entfernt ist. `GetLaserScanScanner()` liefert für ein Lasermesssystem die Messwerte des letzten Messvorganges zurück. Die Methoden `GetLaserScanPlatform()` und `GetLaserScanPlatformMatched()` geben die Daten in Roboterkoordinaten umgerechnet in radialer Form zurück, so als sei vom Robotermittelpunkt aus gemessen worden. Der berechnete Datensatz gibt nicht in jedem Fall das dem Robotermittelpunkt am nächsten liegenden Hindernis zurück. Zwischen Robotermittelpunkt und dem erkannten Hindernis befinden sich möglicherweise ein vom Lasermesssystem nicht erkannte Hindernisse.

Die oben aufgeführten Funktionen beziehen ihre Daten aus den einzelnen CLASER-Objekten und reichen sie an CGENBASE durch. Daten werden dabei nur bei Bedarf verschickt, das heißt, nicht alle Messdaten werden zwangsweise abgefragt. Für eine Lokalisation ist es notwendig, dass möglichst alle Messdaten zeitnah verarbeitet werden, ohne dass sie regelmäßig abgerufen werden müssen. Der CLASERFEEDER sammelt deshalb von den CLASER-Threads die einzelnen Messdaten ein, um sie an die Lokalisationseinheit weiterzureichen. Hierbei achtet der CLASERFEEDER darauf, dass einzelne Messdaten nicht zu alt sind. Gehen beispielsweise bei einem Lasermesssystem zwei Messdatensätze nacheinander verloren, so sind die Messdaten des anderen Lasermesssystems bereits veraltet. Sobald von beiden Lasermesssystemen aktuelle Messwerte vorliegen, ruft der CLASERFEEDER beim CGENBASE-Objekt die Methode `void UpdateLaser(CVEC &marks, int *matched)` mit den gefundenen Reflektormarken in `marks` und einem leeren Integer-Array `*matched` auf. Der Abstand muss dabei in Millimetern übergeben werden. Die Lokalisation versucht, mit Hilfe der aktuellen Position in der Karte und dem Wissen, wo sich überall Reflektormarken befinden, die neue Position zu ermitteln. Zur Positionsbestimmung werden nicht nur die gefundenen Reflektormarken verwendet, sondern auch die Auswertung der Odometrie. Weitere Informationen zur Lokalisation von mobilen Robotern ist in [SW02] zu finden.

Die CLASER-Objekte kapseln CSICKLASER und CSICKSIMLASER unter der gemeinsamen Klasse _RAWLASER. Im Konstruktor wird der Typ des zu erzeugenden Objektes übergeben. Zum Initialisieren ruft CLASER die Methoden `Reset()` und `StartScanContinuous()` vom _RAWLASER-Objekt auf. Im Main-Loop des Threads werden mit `GetScanContinuous()` die nächsten Messwerte abgefragt. Die Funktion blockiert bis die nächsten Daten vorhanden sind. Für die empfangenen Daten wird hinterher grundsätzlich der Minimalabstand zum nächsten Hindernis berechnet und eine Transformation in Roboterkoordinaten durchgeführt. Es spielt dabei keine Rolle, ob die Messwerte in diesen Formaten benötigt werden, sie werden „auf Vorrat“ berechnet. Ihre Gültigkeit geht mit dem nächsten Messdatensatz bereits wieder verloren.

2.5.3 Voruntersuchung an dem bestehenden System

Normalerweise erzeugt der `mobiled` keine Debuginformationen. Wird er jedoch mit dem Parameter `-d` gestartet, so entstehen unter anderem Logeinträge der folgenden Art:

```
#    0x00000004@133755.066630    (0x080af128->CSICKLASER::GetScanContinuousSick(
CSICKSCAN&)) : scanner=0, msg=131/669, skipped=2190
```

Jedes mal, wenn es aufgrund zu spät abgeholter Byte zu Telegrammverlusten kommt, wird beim nächsten korrekt empfangenen Telegramm ein Logeintrag geschrieben. Für eine Sekunde ergibt sich für ein Lasermesssystem zum Beispiel folgender Auszug:

```
# 0x00000004@133755.066630 ... : scanner=0, msg=131/669, skipped=2190
# 0x00000004@133755.284342 ... : scanner=0, msg=132/675, skipped=1458
# 0x00000004@133755.440026 ... : scanner=0, msg=133/679, skipped=1459
# 0x00000004@133755.603883 ... : scanner=0, msg=134/684, skipped=722
# 0x00000004@133755.761185 ... : scanner=0, msg=135/688, skipped=1459
# 0x00000004@133755.814508 ... : scanner=0, msg=136/689, skipped=728
# 0x00000004@133755.920168 ... : scanner=0, msg=137/692, skipped=729
```

Während zwei Sekunden sollen im Schnitt 75 Telegramme ankommen. Dem Logfileausschnitt ist zu entnehmen, dass $692 - 669 = 23$ Telegramme in der Sekunde angekommen sind. Siebenmal musste nach einem neuen Telegrammanfang gesucht werden, dabei wurden zwischen 722 und 2190 Byte übersprungen. Unter Berücksichtigung einer Telegrammlänge von 732 Byte ist festzuhalten, dass teilweise nicht nur ein Telegramm verworfen wurde, sondern gleich drei Telegramme in Folge. In der ersten Zeile sind nur 2190 von $3 \cdot 732 = 2196$ Byte angekommen. Das Verhältnis von 131/669 (KO/OK-Messages), wie es das Logfile suggeriert, stimmt nicht. Die sieben oben aufgeführten KO-Messages haben einen Verlust von zwölf Telegrammen nach sich gezogen. Um genauere Werte zu erhalten, sind längere Messungen durchgeführt worden.

In Tabelle 2.6 ist die Häufigkeit angegeben, wie oft eine bestimmte Anzahl an Byte in einem Messdurchgang verworfen wurde. Die Summe über die verloren gegangenen Telegramme ergibt 4109. Während dieser Zeit von ungefähr fünf Minuten sind 7257 Telegramme richtig empfangen worden, so dass von den $7257 + 4109 = 11366$ Telegrammen ca. 36,2 % verloren gingen. Bei dem zweiten Lasermesssystem ist die Verlustrate etwas höher, 4629 Telegramme sind von 11357 verloren gegangen. Damit liegt die Verlustrate bei ca. 40,8 %. Die Erklärung für den höheren Verlust des zweiten Lasermesssystems ist im Kernelmodul `mxser` zu finden, da die seriellen Ports in aufsteigender Reihenfolge nacheinander abgearbeitet werden.

Die Anzahl verloren gegangener Byte liegt bei 11259. Die 11366 ausgesandten Telegramme bestehen aus $11366 \cdot 732 = 8319912$ Byte, folglich sind ca. 0,135 % der empfangene Byte im FIFO der Moxa-Karte verloren gegangen. Bereits ein fehlendes Byte macht das Telegramm unbrauchbar. Wären die fehlenden Byte stochastisch unabhängig verteilt, so kämen nur ca. 37 % der Telegramme an.

Der FIFO-Puffer der Moxa-Karte ist 16 Byte groß. Es besteht die Möglichkeit, einzustellen, bei welchem Füllstand des FIFO ein Interrupt ausgelöst werden soll. Je früher ein Interrupt ausgelöst wird, desto stärker ist die Belastung des Hauptprozessors.

Der Hostrechner des Roboters verbringt nach einem Bootvorgang ohne das Starten zusätzlicher Software wie dem `mobiled` 0,4 % seiner Zeit im *user mode* und 0,03 % im *system mode*. Es handelt sich bei den Werten um Durchschnittswerte für 60 Sekunden. Wird der `mobiled` gestartet, so steigt die Zeit im *user mode* auf 52,3 % und im *system mode* auf 23,4 %. Die Werte (insbesondere für den *system mode*) sind sehr hoch.

Das Kernelmodul für die Moxa-Schnittstellenkarte liegt als Quellcode vor. Die Interrupt-Handler-Funktion `mxser_interrupt()` wird im Kernel für den IRQ der Moxa-Karte registriert. Nach

Byte verworfen	Häufigkeit verworfener Telegramme			Byte verloren		
721	2			$11 \cdot 2 = 22$		
722	4			$10 \cdot 4 = 40$		
723	7			$9 \cdot 7 = 63$		
724	11			$8 \cdot 11 = 88$		
725	12			$7 \cdot 12 = 84$		
726	28	$1110 \cdot 1 = 1110$		$6 \cdot 28 = 168$	2905	
727	70			$5 \cdot 70 = 350$		
728	192			$4 \cdot 192 = 768$		
729	169			$3 \cdot 169 = 507$		
730	200			$2 \cdot 200 = 400$		
731	415			$1 \cdot 415 = 415$		
1449	1			$15 \cdot 1 = 15$		
1450	4			$14 \cdot 4 = 56$		
1451	9			$13 \cdot 9 = 117$		
1452	4			$12 \cdot 4 = 48$		
1453	14			$11 \cdot 14 = 154$		
1454	24			$10 \cdot 24 = 240$		
1455	34	$646 \cdot 2 = 1292$		$9 \cdot 34 = 306$	3770	
1456	60			$8 \cdot 60 = 480$		
1457	73			$7 \cdot 73 = 511$		
1458	97			$6 \cdot 97 = 582$		
1459	131			$5 \cdot 131 = 655$		
1460	75			$4 \cdot 75 = 300$		
1461	66			$3 \cdot 66 = 198$		
1462	54			$2 \cdot 54 = 108$		
2177	1			$19 \cdot 1 = 19$		
2178	1			$18 \cdot 1 = 18$		
2179	1			$17 \cdot 1 = 17$		
2180	4			$16 \cdot 4 = 64$		
2181	2			$15 \cdot 2 = 30$		
2182	4			$14 \cdot 4 = 56$		
2183	11			$13 \cdot 11 = 143$		
2184	20		4109	$12 \cdot 20 = 240$		11259
2185	19	$319 \cdot 3 = 957$		$11 \cdot 19 = 209$	2589	
2186	26			$10 \cdot 26 = 260$		
2187	43			$9 \cdot 43 = 387$		
2188	44			$8 \cdot 44 = 352$		
2189	34			$7 \cdot 34 = 238$		
2190	51			$6 \cdot 51 = 306$		
2191	30			$5 \cdot 30 = 150$		
2192	16			$4 \cdot 16 = 64$		
2193	12			$3 \cdot 12 = 36$		
2908	1			$20 \cdot 1 = 20$		
2909	1			$19 \cdot 1 = 19$		
2910	4			$18 \cdot 4 = 72$		
2911	3			$17 \cdot 3 = 51$		
2912	3			$16 \cdot 3 = 48$		
2913	8			$15 \cdot 8 = 120$		
2914	8			$14 \cdot 8 = 112$		
2915	16			$13 \cdot 16 = 208$		
2916	22	$180 \cdot 4 = 720$		$12 \cdot 22 = 264$	1905	
2917	17			$11 \cdot 17 = 187$		
2918	30			$10 \cdot 30 = 300$		
2919	23			$9 \cdot 23 = 207$		
2920	15			$8 \cdot 15 = 120$		
2921	12			$7 \cdot 12 = 84$		
2922	10			$6 \cdot 10 = 60$		
2923	5			$5 \cdot 5 = 25$		
2924	2			$4 \cdot 2 = 8$		
3641	1			$19 \cdot 1 = 19$		
3642	1			$18 \cdot 1 = 18$		
3645	1	$6 \cdot 5 = 30$		$15 \cdot 1 = 15$	90	
3646	2			$14 \cdot 2 = 28$		
3650	1			$10 \cdot 1 = 10$		

Tabelle 2.6: Auswertung verloren gegangener Telegramme

einem Aufruf der Funktion überprüft sie selbst, ob die Moxa-Karte gemeint ist, da sich mehrere Karten einen IRQ teilen können. Ist dies der Fall, so wird in einer `while`-Schleife die IRQ-Bitmaske der Karte überprüft und anschließend mit `mxser_receive_chars()` das nächste empfangene Byte abgeholt.

Im Rahmen der Voruntersuchungen zu dieser Arbeit wurde das Kernelmodul soweit angepasst, dass Zeitmessungen durchgeführt werden können, um zu ermitteln, wo die Rechenzeit verloren geht. Die Funktion `mxser_receive_chars()` wurde dabei so erweitert, dass am Anfang die Zeitmessung gestartet und am Ende gestoppt wurde. Nach einer Sekunde wurde die insgesamt verstrichene Zeit sowie die Anzahl der empfangenen Byte ins Kernel-Logpuffer ausgegeben. Es ergaben sich folgende Zeilen:

```
took 143799 us for 54902 chars in 1000259 us world time
took 143600 us for 54736 chars in 1000437 us world time
took 143718 us for 54893 chars in 1000146 us world time
took 143383 us for 54724 chars in 1000556 us world time
took 143911 us for 54854 chars in 1000034 us world time
took 143379 us for 54737 chars in 1000635 us world time
took 143819 us for 54899 chars in 1000197 us world time
took 143355 us for 54731 chars in 1000468 us world time
took 143733 us for 54866 chars in 1000178 us world time
```

Pro Sekunde kommen pro Lasermesssystem $37,5 \cdot 732 = 27450$ Byte an, was für zwei Lasermesssysteme 54900 Byte pro Sekunde sind. Der Durchschnittswert dieses Kernel-Logfile-Auszugs liegt bei ca. 54816, was mit der oben ermittelten Byteverlustrate in etwa übereinstimmt. Die Zeit, die der Hauptprozessor zum Empfangen der Daten benötigt, liegt damit schon bei ca. 14,3 %.

Der Treiber der Moxa-Karte ist also ineffizient, da die Daten nicht durch *memory mapping* in den Speicher kommen, sondern alle einzeln mit programmiertem I/O (*pio*) abgeholt werden müssen.

2.6 Zusammenfassung

In diesem Kapitel wurden die theoretischen Grundlagen, auf denen die folgenden Kapitel basieren, sowie die am bisherigen System durchgeführten Vorarbeiten dargestellt. Eine Analyse der bisherigen Anbindung der Lasermesssysteme deckte Probleme bei der Zuverlässigkeit und der Systemauslastung auf. Etwa 40 % der Messdatentelegramme gehen verloren und die Systemauslastung steigt beim Betrieb der Lasermesssysteme auf 75,6 %. Als Ursache stellte sich die serielle Schnittstellenkarte mit einem zu kleinen FIFO-Speicher und einem ineffizienten Treiber heraus.

3 Auswahl und Beschreibung der Hardware

Nachdem im vorangegangenen Kapitel die Probleme der bisherigen Anbindung dargelegt sind, geht es in diesem Kapitel darum, eine Anbindungsmöglichkeit der Lasermesssysteme an den Hostrechner zu entwickeln, die bezüglich der Systemauslastung und der Empfangssicherheit eine deutliche Verbesserung darstellt. Dazu wird zunächst ein Konzept für eine Lösung erstellt und es werden Anforderungen an diese Lösung formuliert. Entscheidungen bezüglich der Schnittstellenwahl sowie der Wahl der Realisierungsmethode werden abgewogen. Schließlich erfolgt eine ausführliche Darstellung der ausgewählten Hardware. Hierbei wird vor allem auf die Aspekte eingegangen, die für die Anbindung der Lasermesssysteme von entscheidender Rolle sind.

3.1 Konzept eines Realisierungsweges

Im folgenden Abschnitt werden verschiedene Realisierungsmöglichkeiten diskutiert und schließlich die Entscheidung für einen Lösungsweg getroffen. Die Lösung soll die Struktur haben, dass zwischen die Lasermesssysteme und den Hostrechner ein weiteres Gerät installiert wird. Dieses Gerät soll den Datenstrom der Lasermesssysteme sammeln und an den Hostrechner in größeren Datenblöcken weiterleiten. Beim Empfangen und Übertragen dürfen keine Daten verlorengehen. Die zu entwickelnde Lösung soll sich möglichst unabhängig von der Art des vorhandenen Hostsystems einsetzen lassen. Für den Fall, dass im Arbeitsbereich TAMS am Serviceroboter der Steuerrechner getauscht wird, könnte das System dann trotzdem weiterverwendet werden. Ebenfalls wäre eine Integration des Systems in andere Projekte möglich.

Das Hauptziel bei der Entwicklung des Systems ist die Reduzierung der Rechnerauslastung des Hostsystems. Diese Reduzierung soll vor allem durch Übertragung des Datenstroms in größeren Datenblöcken erreicht werden, da so vermieden werden kann, dass der Hostrechner zum Empfangen weniger Byte eine Interrupt-Service-Routine ausführen muss.

3.1.1 Anforderungen an die Funktionalität des System

An die Funktionalität des zu entwickelnden Systems können verschiedene Anforderungen gestellt werden. Diese können nach ihrer Relevanz gruppiert werden. Je nach gewähltem Lösungsweg und dem Entwicklungserfolg des Systems sollen in der späteren Anwendung möglichst viele dieser Anforderungen erfüllt werden. Die Anforderungen sind in der folgenden Liste zusammengefasst und werden anschließend erläutert.

- Grundanforderungen
  - mindestens eine RS-422-Schnittstelle mit 9,6 kBd, 19,2 kBd, 38,4 kBd und 500 kBd
  - Schnittstelle zum Hostsystem

    - Durchreichen der Daten zwischen Hostsystem und Lasermesssystem(en)
- Erweiterte Anforderungen
    - Synchronisation auf Telegrammebene
    - Steuerung der Lasermesssysteme
    - Flusskontrolle
- Einfache Vorverarbeitung
    - CRC-Prüfung
    - Berechnung des Minimalabstands
    - Markenbestimmung
- Komplexe Vorverarbeitung
    - Umrechnung aller Messwerte in Roboterkoordinaten
    - Implementierung beliebiger Filter (z.B. Median, Reduktion, Winkelreduktion, Linien, Ecken)

Grundanforderungen

Die Grundanforderungen müssen in jedem Fall erfüllt werden, damit dieses System später zum Einsatz kommen kann. Es sollte zumindest ein Lasermesssystem angebunden werden können. Das System soll in der Lage sein, die Daten des Lasermesssystems über eine RS-422-Schnittstelle bei einer Geschwindigkeit von 500 kBd entgegenzunehmen und über einen geeigneten Übertragungsweg an den Hostrechner zu übermitteln. Hierbei dürfen keine Informationen verloren gehen. Ebenfalls muss die serielle Kommunikation mit den Baudraten 9,6 kBd, 19,2 kBd und 38,4 kBd unterstützt werden. Vom Hostrechner sollen Konfigurationsdaten an die Lasermesssysteme gesendet werden können.

Mit solchen Merkmalen ausgestattet, hat das zu entwerfende System die Funktion eines Schnittstellenumsetzers. Auf der Seite des Hostrechners erfolgt weiterhin die komplette Ansteuerung der Lasermesssysteme sowie die Einteilung des Datenstroms in Telegramme. Von dieser Lösung kann bereits eine Reduzierung der Auslastung des Hostsystems erwartet werden, sofern eine Schnittstelle verwendet wird, die über eine effiziente Implementierung im Betriebssystem verfügt. Das Übertragen größerer Blöcke ermöglicht eine weitere Entlastung.

Erweiterte Anforderungen

Zu den erweiterten Anforderungen an das zu entwerfende Systems zählt vor allem das Synchronisieren des eintreffenden Datenstroms auf Telegrammebene. Hierdurch soll das System in die Lage versetzt werden, den Datentransfer zum Hostrechner immer genau dann zu starten, wenn ein komplettes Telegramm empfangen wurde. Das Einsynchronisieren in einen laufenden Datenstrom stellt dabei eine besondere Herausforderung dar. Des Weiteren soll die Synchronisation und die Steuerung der Lasermesssysteme direkt von dem zu entwickelnden System erfolgen und keinen Eingriff des Hostrechners mehr erforderlich machen. Hierzu müssen die empfangenen Telegramme teilweise ausgewertet werden, um Informationen über den Zustand der Messsysteme zu erhalten. Abhängig von diesen Zuständen müssen selbstständig Konfigurationstelegramme generiert und an die Lasermesssysteme verschickt werden. Im Betrieb soll eine Flusskontrolle

den kontinuierlichen Datenstrom überwachen. Bricht dieser ab, so soll versucht werden, einen erneuten Synchronisationsvorgang zu starten.

Bei Erfüllung dieser erweiterten Anforderungen wird eine weitere Entlastung des Hostrechners erreicht. Die Suche nach Telegrammanfängen entfällt, da immer komplette Telegramme übertragen werden. Ebenfalls entfällt auf der Seite des Hostrechners die Steuerung der Lasermesssysteme und die Überwachung des Datenstroms. Neben der Senkung der Systemauslastung verringert sich die Komplexität der Software auf dem Hostrechner deutlich. In der Anwendung muss lediglich auf eintreffende Telegramme gewartet werden. Die Steuerung der Lasermesssysteme muss nicht mehr implementiert werden. Dies stellt einen großen Vorteil dar, wenn Änderungen an der Software vorgenommen werden oder ähnliche Anwendungen auf anderen Systemen entwickelt werden sollen.

Einfache Vorverarbeitung

Für das zu entwerfende System wäre es von Vorteil, wenn ein möglichst großer Teil der Datenverarbeitung, die bei der aktuellen Implementierung im Hostsystem durchgeführt wird, in das zu entwickelnde System ausgelagert wird. Im Telegramm der SICK-Lasermesssysteme ist eine CRC-Prüfsumme enthalten, über die sich der korrekte Empfang der Messdaten sicherstellen lässt. Die Auswertung der Prüfsumme könnte im zu entwickelnden System stattfinden. Die Synchronisation auf Telegrammebene könnte durch die Auswertung ebenfalls verbessert werden, da eine fehlerhafte Synchronisation mit allergrößter Wahrscheinlichkeit an einer falschen Prüfsumme zu erkennen ist.

Für jeden empfangenen Datensatz wird zur Kollisionsvermeidung im Hostrechnersystem der Abstand zum nächstgelegenen Hindernis errechnet. Dieser Abstand wird vom Robotermittelpunkt ausgehend bestimmt. Dazu sind einige Rechenoperationen pro gemessenem Wert erforderlich. Eine Implementierung dieser Operationen innerhalb des zu entwickelnden Systems wird daher angestrebt.

Da die Lokalisation des Robotersystems auf der Erkennung von Reflektormarken basiert, findet im Steuersystem die Generierung einer Liste von Marken statt. Eine Marke kann in mehreren benachbarten Messwerten erscheinen. Daher muss für diese Marke dann die tatsächliche Position abgeschätzt werden. Diese Operationen könnten ebenfalls zur Reduzierung der Rechnerauslastung in das zu entwickelnde System ausgelagert werden.

Komplexe Vorverarbeitung

In den Softwareklassen der Anwendung auf dem Steuerrechner steht die Funktion bereit, die Lasermessdaten in Roboterkoordinaten anzeigen zu lassen. Hierbei werden Winkel und Länge imaginärer Strahlen berechnet, die ihren Ursprung im Robotermittelpunkt haben. Diese Funktion wird im System nicht zwingend zur Lokalisierung benötigt, jedoch erfolgt der Aufruf dieser Funktion, wenn ein Programm zur Visualisierung der Lasermessdaten gestartet wird.

Darüber hinaus besteht die Möglichkeit, weitere Vorverarbeitungsalgorithmen auf die Lasermessdaten anzuwenden. In [Zha05] sind einige Algorithmen beschrieben, die zur Vorverarbeitung von

Lasermessdaten sinnvoll sind. Es handelt sich um Medianfilter, Reduktionsfilter, Winkelreduktionsfilter, Linienfilter und Eckenfilter. Letztere dienen zum Auffinden von Linien und Ecken in den Messdaten und eignen sich zur Vorverarbeitung für eine Lokalisation ohne Reflektormarken. Der Medianfilter dient zum Glätten eines Messdatensatzes. Einzelne Fehlmessungen können so eliminiert werden. Der Reduktionsfilter verringert die Datenmenge, ohne relevante Bildelemente zu verlieren. Anhäufungen von Punkten werden zu einem Punkt reduziert. Der Winkelreduktionsfilter dient zur Verringerung der Auflösung, indem mehrere Punkte nach festem Muster zu einem Punkt zusammengefasst werden. Die Implementierung dieser Filter kann in zukünftigen Versionen der Anwendungssoftware auf dem Hostrechner verwendet werden.

Bei der Implementierung der Vorverarbeitungen muss stets sichergestellt werden, dass die Grundanforderungen unter allen Umständen erfüllt bleiben. Einschränkungen, wie der Verlust von Messdaten, dürfen nicht auftreten.

3.1.2 Weitere Entwurfskriterien

Zu den im letzten Teil beschriebenen Anforderungen an den Funktionsumfang kommen weitere Kriterien, die zur Realisierung des Systems zu erfüllen sind:

Kosten Die Kosten für die Anschaffung der Hardware sollten im Rahmen von ungefähr 300 Euro bleiben.

Abmessungen Weil das Innenleben des Serviceroboters relativ dicht mit Komponenten besetzt ist, bietet sich die Montage an der Außenwand an. Dieser Platz ermöglicht auch einen einfachen Zugang für die Wartung des Systems. Zusätzlich lassen sich Betriebsanzeigen am System jederzeit ablesen. Für die Abmessungen ergibt sich somit keine generelle Beschränkung, jedoch sollte das System nicht über den Grundriss des Roboters hinausragen, um keinerlei zusätzliche Kollisionsgefahr darzustellen.

Stromversorgung Die Stromversorgung ist unproblematisch, da der Serviceroboter für seinen Antrieb über eine leistungsfähige Stromversorgung durch Bleiakkumulatoren mit einer Spannung von 48 Volt verfügt. Über Spannungswandler könnten die 48 Volt auf beliebige Werte angepasst werden. Da neben den Motoren auch das Hostrechnersystem auf Pentium-4-Basis versorgt wird, ist die im Verhältnis dazu geringe Leistungsaufnahme des zu entwickelnden Systems relativ unkritisch.

Persistente Codespeicherung Für den Fall, dass ein Entwicklungsboard eingebaut wird, muss sichergestellt werden, dass dieses mit einem nichtflüchtigen Speicher ausgestattet ist und zum Systemstart automatisch das auszuführende Programm lädt. Eine wiederkehrende Programmierung nach Einschalten der Stromversorgung wäre nicht praktikabel, auch wenn dieses durch Startskripte vereinfacht werden könnte.

3.1.3 Wahl der Schnittstelle

Das zu entwickelnde System muss in der Lage sein, die Telegramme an den Hostrechner weiterzuleiten. Hierzu muss eine geeignete Schnittstelle ausgewählt werden. Diese muss sowohl am Hostrechner als auch am System selbst vorhanden sein. Um die Integration auch in beliebige andere Hostsysteme zu ermöglichen, soll eine Schnittstelle gewählt werden, die eine größtmögliche Verbreitung aufweist.

Hier bieten sich vor allem USB[1] und Ethernet an. Diese sollen nun näher vorgestellt werden, insbesondere sollen die Vor- und Nachteile beim Einsatz dieser Schnittstellen dargelegt werden.

USB

Nahezu jeder aktuelle PC ist mit mehreren USB-Anschlüssen ausgestattet. Über diese Anschlüsse kann beliebige Peripherie angebunden werden. Zu diesen Geräten gehören unter anderem Computermäuse, Tastaturen, Scanner, Drucker, MP3-Player, Digitalkameras, Speichersticks und externe Festplatten. Die angebundenen Geräte werden als Clients bezeichnet, bei den USB-Ports am PC handelt es sich um Host-Anschlüsse. Es wird zwischen den Standards USB 1.1 und USB 2.0 unterschieden, wobei der USB 1.1 Standard 12 MBit/s erreicht und USB 2.0 eine Datenrate von 480 MBit/s bewältigt. Beide Standards wären hinsichtlich der Datenrate für den Einsatzzweck mehr als ausreichend.

Auf vielen Entwicklungsplatinen sind USB-Schnittstellen vorhanden. Dieses bezieht sich sowohl auf mikrocontrollerbasierte Platinen als auch auf FPGA-Boards. Die USB-Schnittstelle wird zum einen zur Programmierung verwendet, zum anderen findet der Datentransfer im Betrieb über die USB-Schnittstelle statt. Auf der Seite des zu entwickelnden Systems ist die Implementierung ohne großen Aufwand möglich. Auf der Webseite www.easyfpga.com werden FPGA-Platinen mit USB-Schnittstellen angeboten. Bei diesen Boards sind die USB-Schnittstellen über eine FIFO-artige Struktur ansprechbar. Die Daten werden byteweise in diesen FIFO geschrieben und erreichen den Hostrechner in der gleichen Reihenfolge, in der sie zuvor geschrieben wurden. Beim Datentransfer in entgegengesetzter Richtung wird über ein Signal angezeigt, dass neue Daten abrufbar sind. Diese können dann byteweise gelesen werden. Auf der Seite des Hostrechners erfolgt die Kommunikation über mitgelieferte lizenzfreie USB-Treiber. Diese werden jedoch nur für die Betriebssysteme von Microsoft beigefügt, so dass zu den Entwicklungsaufgaben die Implementierung eines speziellen USB-Treibers unter Linux gehören würde. Selbst wenn ein Linuxtreiber mitgeliefert wäre, könnte keinerlei Aussage hinsichtlich der Systemauslastung und der Datensicherheit gemacht werden. Eine Portierung der Anwendung auf andere Betriebssysteme würde eine aufwändige Treiberentwicklung erforderlich machen.

Ethernet

Eine Netzwerkschnittstelle nach dem 10/100Base-T (oder eventuell auch 10/100/1000Base-T) Standard ist heutzutage Grundausstattung jedes PCs. Diese dient zum Austausch von Informationen mit entfernten Rechnern. Typische Anwendungen sind das Abrufen von Internetseiten

[1] *universal serial bus*

oder der Zugriff auf entfernte Dateisysteme. In jüngerer Zeit werden über die Netzwerkschnittstelle vermehrt auch Echtzeitdaten übertragen, also Daten, die mit einer geringen Verzögerungszeit ihr Ziel erreichen müssen. Dabei handelt es sich um Audio- und Videoübertragung sowohl in Kommunikationsanwendungen als auch in der Ausstrahlung eines Radio- oder Fernsehprogramms.

Obwohl immer mehr Endgeräte über eine Funknetzwerkschnittstelle verfügen, wird die kabelgebundene Variante voraussichtlich noch viele Jahre erhalten bleiben, da die kabellose Übertragung bezüglich Datenrate und Zuverlässigkeit deutliche Nachteile gegenüber der kabelgebundenen Netzwerkschnittstelle hat.

Die Kommunikation erfolgt in der Regel über das TCP/IP oder das UDP/IP-Protokoll. Alle Endgeräte, die dieses Protokoll unterstützen, können miteinander Daten austauschen, wobei die Architekturen der Rechnersysteme und die verwendeten Betriebssysteme keine Einschränkungen bezüglich der Interoperabilität darstellen. Die Datenrate von 10 MBit/s bei der langsamsten Form 10Base-T ist für die Anbindung von Lasermesssystemen ausreichend. Auf der Seite des Hostrechners wäre keinerlei Treiberentwicklung notwendig. Es ist prinzipiell auf jedem System mit Netzwerkschnittstelle ein Treiber vorhanden, der für hohe Datenraten bei niedriger Systemauslastung optimiert ist. Die TCP/IP und UDP-Implementierung ist Bestandteil des Betriebssystems, so dass ebenfalls kein zusätzlicher Entwicklungsaufwand anfallen würde.

Auf der Seite des zu entwickelnden Systems müsste hingegen die Implementierung des TCP/IP- und des UDP-Protokolls bedacht werden. Aufgrund der relativ hohen Komplexität der Protokolle werden diese meist von einem Mikrocontroller oder Prozessor abgearbeitet. Es sind auch FPGA-Lösungen mit Netzwerkschnittstellen erhältlich, auf denen dann eine sogenannte synthetische CPU aufgesetzt wird, um die nötigen Protokollabläufe steuern zu können. Preislich liegen solche Systeme jedoch außerhalb des gesetzten Rahmens. Bei vielen Lösungen wird eine Softwareimplementierung der Protokolle bereits mitgeliefert, so dass der Entwicklungsaufwand überschaubar bleibt und im Rahmen dieser Arbeit durchführbar ist.

Nachteile der Verwendung einer Netzwerkschnittstelle stellen sich vor allem dann ein, wenn weitere Kommunikation über diese Netzwerkschnittstelle erfolgt. Die Lasermessdaten erzeugen eine geringe, aber ständige Auslastung des Netzwerkes. Bei weiterer Netzwerkaktivität kann es durch Kollisionen zu Paketverlusten kommen, so dass Lasermessdaten verloren gehen können.

3.1.4 Entscheidung für einen Realisierungsweg

Aufgrund der einfachen Implementierbarkeit einer Netzwerkverbindung auf Hostrechnerseite erfolgt die Entwicklung des Eingebetteten Systems auf einem Modul mit Netzwerkschnittstelle.

Die Nachteile der niedrigeren Datensicherheit und der Netzwerkauslastung lassen sich umgehen, indem am Hostrechner eine Netzwerkschnittstelle dediziert für die Anbindung des zu entwickelnden Systems betrieben wird. Da es sich bei dem Serviceroboter um eine mobile Plattform handelt, ist der Netzwerkanschluss unbelegt und kann verwendet werden. Selbst für den Fall eines belegten Anschlusses könnte am System über einen USB-Ethernet-Adapter kostengünstig eine Netzwerkschnittstelle nachgerüstet werden, die dann zur Verfügung steht.

In der gesetzten Preisklasse bieten sich mikrocontrollerbasierte Entwicklungsplatinen an, die über serielle Schnittstellen und eine Ethernet-Schnittstelle verfügen. Als problematisch stellt

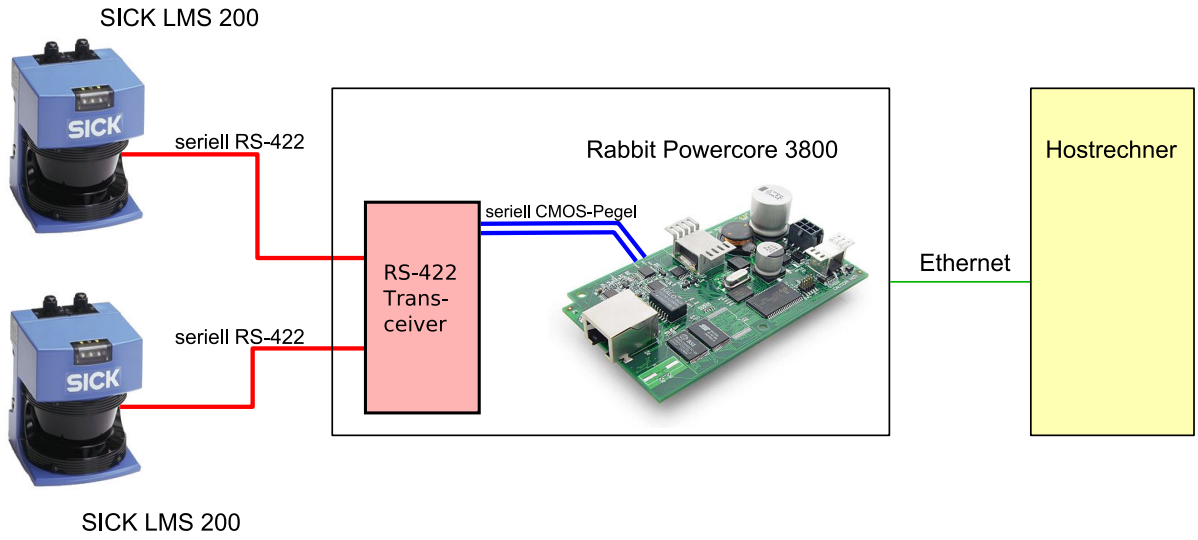


Abbildung 3.1: Aufbau neuer Anbindung

sich die benötigte Baudrate von 500 kBd heraus. Entweder ist die maximal unterstützte Geschwindigkeit niedriger oder es werden nur Vielfache der Standardgeschwindigkeit unterstützt. In diesem Fall müssten externe UARTs mit eigenen Taktquellen genutzt werden, die dann von einem Mikrocontroller als externes Gerät angesprochen werden würden.

Die Firma Rabbit Semiconductor[2] bietet zu ihren Produkten sämtliche Datenblätter und Anleitungen im Internet zum freien Download an. Anhand dieser Quellen konnte herausgearbeitet werden, dass der kompakte Einplatinen-Computer „Powercore 3800“ die geforderte Baudrate ohne zusätzliche Hardware unterstützt.

Im Folgenden sind die Hauptmerkmale aufgezählt, die für die Wahl des Powercore 3800 bei der Entwicklung des Systems zur Anbindung zweier Lasermesssysteme der Firma SICK sprechen:

- sechs serielle Schnittstellen (es wird eine Baudrate von 496 kBd unterstützt, d.h. Abweichung unter 1 %)
- 10Base-T Ethernet
- Development-Kit für 169 Euro erhältlich
- TCP/IP-Bibliothek im Lieferumfang

Die geplante Anbindung ist in Abbildung 3.1 grafisch dargestellt.

Eine grobe Abschätzung der Anwendungsperformance kann anhand gegebener Beispiele in den Dokumentationen vorgenommen werden. Für das Empfangen serieller Daten wird in [R3000UM, S. 192] eine Interrupt-Service-Routine vorgestellt, die auf einem Prozessor dieses Typs 117 Takte zum Empfangen eines Byte benötigt. Für den Fall, dass eine volle Auslastung von zwei seriellen Ports vorliegt, kann die Systemauslastung durch das Empfangen von Byte berechnet werden. Pro Sekunde können $2 \cdot \frac{500.000}{10} = 100.000$ Byte eintreffen. Der Prozessor des Powercore 3800 hat eine

[2] wurde inzwischen von Digi International übernommen

Taktrate von 51,6 MHz. Von den 51,6 Millionen Takten pro Sekunde werden dann 11,7 Millionen zum Datenempfang benötigt. Die resultierende Systemauslastung durch den Datenempfang beträgt somit etwa 23 %.

Es ist davon auszugehen, dass das System generell in der Lage ist, die Ethernet-Schnittstelle mit einer für diese Verbindung üblichen Datenrate von etwa einem MByte pro Sekunde anzusprechen. Somit wird höchstwahrscheinlich auch bei dieser geringen Systemauslastung die Datenrate nicht unter 100.000 Byte pro Sekunde sinken. Die Grundfunktionalität kann also mit diesem System hergestellt werden. Es bleibt wahrscheinlich noch ausreichend Leistung für die Implementierung zusätzlicher Funktionen. Die Lasermesssysteme lasten ihre seriellen Schnittstellen zudem nur zum Teil aus, was die tatsächliche Systemauslastung im späteren Betrieb noch geringer ausfallen lassen wird.

Aus den dargestellten Gründen erfolgt die Entwicklung des Eingebetteten Systems zur Ansteuerung von zwei SICK-Lasermesssystemen mit einem Rabbit Powercore 3800.

Der Powercore 3800 wird als Development-Kit für 169 Euro angeboten. Bestandteile dieses Kits sind neben dem Rabbit Powercore 3800 die Dynamic-C-Entwicklungsumgebung, ein serielles Programmierkabel, ein Netzteil, verschiedene Dokumentationen und ein Prototyping-Board.

Der Rabbit Powercore 3800 ist ein kompakter Einplatinen-Computer bestehend aus einem Rabbit-3000-Mikroprozessor, Flash-Speicher, RAM, Ethernet-Controller und Spannungswandler. Über eine 50-polige Pin-Steckerleiste wird der Powercore 3800 in eine anwendungsspezifische Platine eingesetzt. Diese Platine wird vom Entwickler selbst entworfen und enthält die für die spezifische Anwendung erforderlichen zusätzlichen elektrischen Bauteile. Das Prototyping-Board dient als Grundlage für eine solche Platine und enthält neben dem Konnektor zu dem Powercore 3800 schon einige Baugruppen wie LEDs, Taster und RS-232-Treiber. Zusätzlich können noch weitere Bauteile auf freiem Platinenplatz verlötet werden.

Der verwendete Rabbit-3000-Prozessor ist eine 8-Bit-CPU für den Einsatz in kleinen und mittleren Steuergeräten. Auf dem Powercore 3800 ist die CPU mit 51,6 MHz getaktet. Die Hauptmerkmale des Prozessors sind im Folgenden zusammengefasst:

- sechs serielle Schnittstellen mit einer Baudrate bis zu einem Achtel des Prozessortakts oder ganzzahligen Teilern davon
- 56 parallele Ein- und Ausgabepins für bis zu sieben parallele 8-Bit-Schnittstellen
- vier PWM (pulsbreitenmodulierte) Ausgänge zur Ansteuerung von Treibern für Leistungselektronik oder zur Steuerung von Digital-analog-Wandlern
- zwei Encoder-Eingänge zum Anschluss von optischen Encodern[3] mit Drehrichtungserkennung
- zwei *Input-Capture-Unit*s zur Zeitmessung zwischen zwei spezifizierbaren Ereignissen
- zahlreiche Timer zur Ereignissteuerung und zur Taktung der seriellen Schnittstellen
- Interrupts auf verschiedenen Prioritätsstufen
- diverse Stromsparmechanismen und Maßnahmen zur elektromagnetischen Verträglichkeit

Die Softwareentwicklung erfolgt über die mitgelieferte Dynamic-C-Entwicklungsumgebung. Lauffähig ist die Software unter allen Windows-Versionen ab Windows 95. Über das mitgelieferte Programmierkabel wird der Powercore 3800 mit der seriellen Schnittstelle des PC verbunden.

[3] Optische Encoder messen die Drehbewegung einer Achse

Programmiert wird der Rabbit Powercore 3800 in einer ISO/ANSI C ähnlichen Programmiersprache oder direkt in Assembler. Die Entwicklungsumgebung vereint folgende Funktionen:

- Editieren des Quellcodes
- Kompilieren und Linken
- Programmieren des Rabbit Powercore 3800
- Debuggen zur Laufzeit

Bestandteil der Entwicklungsumgebung ist auch eine umfangreiche Bibliothek mit Funktionen. Es werden verschiedene mathematische Funktionen und Funktionen zur Datenein- und -ausgabe bereitgestellt. Zusätzlich ist eine lizenzfreie Implementierung des TCP/IP-Protokolls vorhanden, so dass der Anwender mit geringem Aufwand in der Lage ist, Netzwerkanwendungen zu entwickeln.

In den folgenden Teilen dieses Kapitels werden die drei Hauptkomponenten des Development-Kits detailliert vorgestellt. Bei den Vorstellungen werden ebenfalls ausführliche Überlegungen zur späteren Realisierung des Systems mit einbezogen. Zunächst wird der Rabbit-3000-Mikroprozessor ausführlich betrachtet, da dieser das Herzstück des gesamten Systems darstellt. Im darauf folgenden Abschnitt geht es um die Eigenschaften des Rabbit Powercore 3800. Die Dynamic-C-Entwicklungsumgebung ist Thema eines weiteren Abschnitts, wobei insbesondere auf die Dynamic-C-Programmiersprache eingegangen wird. Die Informationen über die Komponenten der Firma Rabbit stammen aus den Dokumentationen [R3000UM], [R3000DH], [RabInst], [PCFlex], [PCUM], [DynCUM] und [DynCFRM].

3.2 Rabbit-3000-CPU

Der Rabbit-3000-Mikroprozessor wurde von der Firma Rabbit Semiconductors in Zusammenarbeit mit der Firma Z-World entwickelt. Die Architektur des Rabbit 3000 ähnelt wie die des Vorgängers Rabbit 2000 dem Zilog Z80. Der Zilog Z80 ist ein 8-Bit-Prozessor, der 1976 vorgestellt wurde und in vielen der ersten erhältlichen Heimcomputer und Spielkonsolen Anwendung fand.

Der Rabbit 3000 stellt eine Weiterentwicklung des Zilog Z80 mit vielen Verbesserungen dar. Folgende Entwicklungsziele sind beim Rabbit 3000 realisiert worden:

- einfache Benutzung, sowohl der Hardware als auch der Software; Debuggen ist im Betrieb über das Programmier-Kabel möglich, es werden daher keine Emulatoren benötigt
- hohe Performance im Vergleich zu anderen 8-Bit-Mikroprozessoren, die durch hohe Taktraten des Prozessors (bis zu 54 MHz und mehr) und durch den kompakten Instruktionssatz erreicht wird
- reichhaltige Ausstattung mit Schnittstellen
- niedrige Kosten für die Development-Kits und die Prozessoren

3.2.1 Spezifikationen

Der Rabbit-3000-Mikroprozessor wird in einem 128-Pin LQFP[4]-Gehäuse geliefert. Hierbei handelt es sich um ein flaches Gehäuse, das in SMD-Technik verlötet wird. Die Abmessungen betragen 16 mm x 16 mm x 1,5 mm. Es werden auch Exemplare im noch kompakteren TFBGA[5]-Gehäuse mit den Abmessungen 10 mm x 10 mm x 1,2 mm angeboten. Bei letzteren sind die elektrischen Anschlüsse auf der Unterseite des Chips in feldartiger Struktur (*Array*) angebracht. Auf jedem dieser Anschlüsse ist schon ein Punkt Lötzinn aufgebracht (*Ball*). Der Lötprozess findet in einem speziellen Lötofen statt.

Die Versorgungsspannung ist mit 1,8 V bis 3,6 V angegeben, wobei die Standardspannung bei 3,3 V liegt. Die Stromaufnahme ist nahezu linear abhängig von der Taktrate und der Betriebsspannung. Bei 3,3 V liegt die Aufnahme bei 2 mA/MHz. Somit ergibt sich bei einer Taktrate von 51,6 MHz eine Stromaufnahme von 103,2 mA und nach $P = U \cdot I$ eine Leistungsaufnahme von 0,34 Watt. Dieser Wert kann bei einer Vielzahl der Anwendungsgebiete vernachlässigt werden. Ein Kühlkörper oder eine aktive Kühlung, wie sie bei aktuellen Prozessoren in Standard-PCs fast unumgänglich ist, wird nicht benötigt. Zusätzlich kann für eine noch geringere Leistungsaufnahme die Taktrate herabgesetzt werden.

Die Pins des Rabbit 3000 haben CMOS-Pegel, also 0 V für einen *low*-Pegel und 3,3 V als *high*-Pegel. Trotzdem vertragen die Pins eine Eingangsspannung von bis zu 5 V. Es kann also direkt TTL-Logik angebunden werden, wobei zu beachten ist, dass TTL-Logik meist eine Schaltschwelle von etwa 2,5 V hat. In der Nähe dieser Schaltschwelle kann es zu erhöhter Stromaufnahme kommen.

3.2.2 Aufbau des Prozessors

Im Folgenden soll der Aufbau der Rabbit-3000-CPU erläutert werden. Die CPU und die Funktionseinheiten sind in Abbildung 3.2 dargestellt. Ebenfalls sind die nach außen führenden Datenleitungen dargestellt. Detaillierte Erläuterungen werden in folgenden Abschnitten für die Komponenten gegeben, die für den Einsatzzweck als System zum Betrieb zweier SICK LMS 200 von besonderer Bedeutung sind. Die Rabbit-3000-CPU besitzt keinen eigenen Speicher und ist damit der Familie der Prozessoren zuzuordnen. Typisch für einen Mikrocontroller wäre jedoch die reichhaltige Ausstattung mit Schnittstellen und Funktionseinheiten.

Innerhalb des Rabbit-3000-Mikroprozessors erfolgt die Kommunikation über einen 8-Bit breiten Datenbus. Der Adressbus hat ebenfalls eine Datenbreite von acht Bit. Dieser Bus dient zum Beschreiben und Auslesen der Register der verschiedenen Komponenten des Prozessors. Sie sind nicht zu verwechseln mit den Registern des CPU-Kerns, auf den die eigentlichen Rechenoperationen durchgeführt werden. Die Register des CPU-Kerns werden in Abschnitt 3.2.3 beschrieben. Schreib- und Leseoperationen auf den Registern der Komponenten können mit verschiedenen Zielen erfolgen:

- Konfiguration der Komponenten (Baudrate der seriellen Schnittstellen)
- Datentransfer (Lesen eines empfangenen Bytes der seriellen Schnittstelle)

[4] *Low Profile Quad Flat Pack*
[5] *Thin Fine Pitch Ball Grid Array*

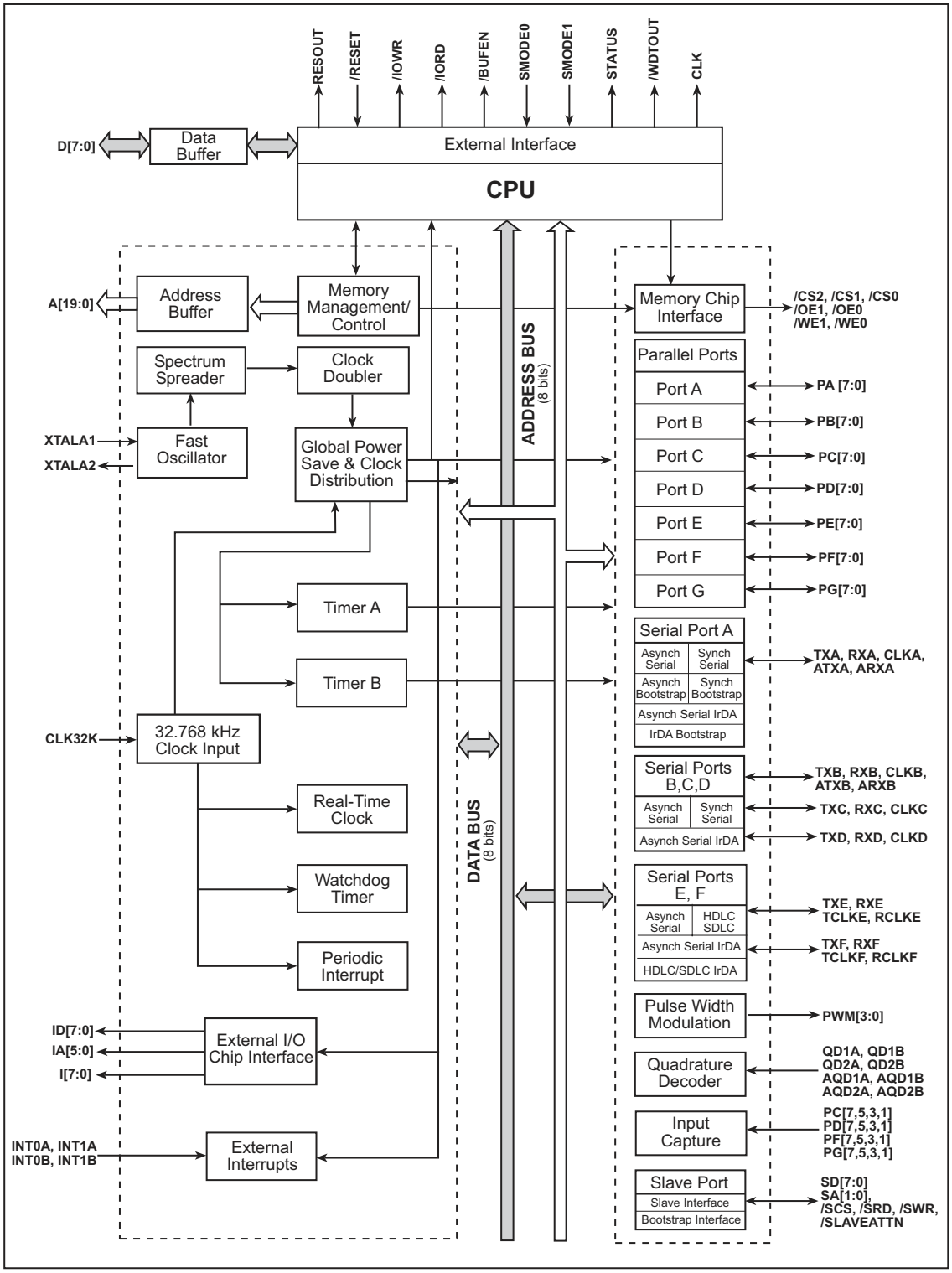


Abbildung 3.2: Aufbau der Rabbit-3000-CPU, Quelle: [R3000UM, S. 5]

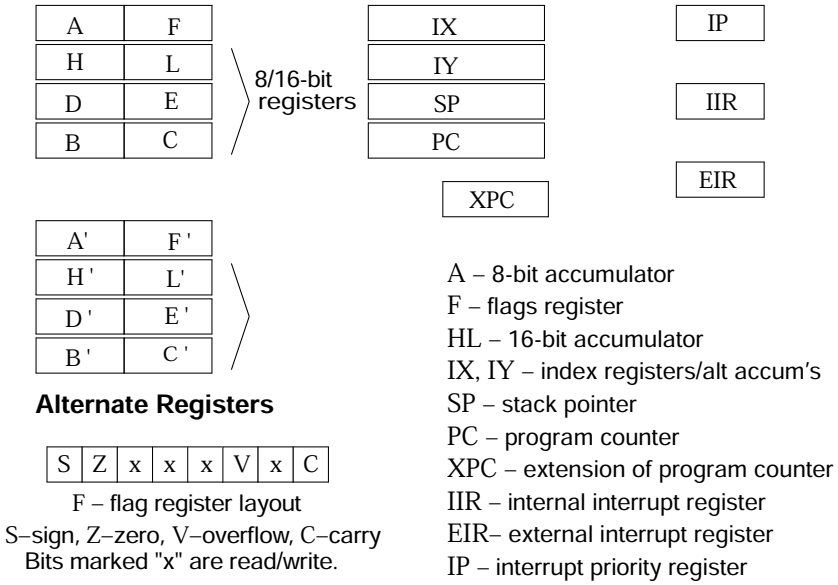


Abbildung 3.3: Registerlayout der Rabbit-3000-CPU, Quelle: [R3000UM, S. 21]

- Prüfen des Zustandes (Prüfen, ob ein Byte empfangen wurde und gelesen werden kann)

Die genannten Beispiele in Klammern beziehen sich auf die seriellen Schnittstellen und werden in Abschnitt 3.2.8 noch näher erläutert. Bei der Beschreibung der anderen Komponenten wird ebenfalls auf die dazugehörigen Register eingegangen.

Aufgrund des 8-Bit breiten Adressbusses könnten theoretisch bis zu 256 verschiedene Register angesprochen werden. Jedoch existieren weniger Register, so dass nicht jede Adresse belegt ist. Es werden sämtliche Konfigurationen der Komponenten in die entsprechenden Register geschrieben.

3.2.3 Register und Operationen

In diesem Abschnitt geht es um das Layout der Prozessorregister und um die darauf arbeitenden Operationen. Diese Informationen sind relevant, wenn die Programmierung in Assembler erfolgen soll. Bei Programmierung in der Programmiersprache Dynamic C braucht sich der Programmierer mit dem Aufbau des Registerlayouts und den Assemblerbefehlen nicht zu befassen.

Der Registersatz der Rabbit-3000-CPU

Bei den Prozessorregistern handelt es sich um schnelle Speicher, die in der Regel für Daten und Adressen verwendet werden. Zusätzlich existieren Register, die spezielle Funktionen haben. In Abbildung 3.3 ist das Registerlayout der Rabbit-3000-CPU dargestellt. Bei dem Register `A` handelt es sich um den 8-Bit Akkumulator. Dieses Register ist das Standardregister für viele Operationen. So erfolgen zum Beispiel über den Akkumulator Zugriffe auf den Speicher. Das Register *F* ist das sogenannte *Flags Register*. Dieses kann nicht direkt beschrieben oder gelesen werden, sondern bei vielen Operationen werden einzelne Bit dieses Registers gesetzt. Die Bedeutung dieser Bits sind folgende:

- Sign-Bit: wird bei Subtraktionen gesetzt, wenn das Ergebnis < 0 ist
- Zero-Bit: wird bei Subtraktionen gesetzt, wenn das Ergebnis $= 0$ ist
- Overflow-Bit: wird gesetzt, wenn eine Zahl den darstellbaren Bereich überschreitet
- Carry-Flag: enthält den Überlauf einer Addition oder Subtraktion

Diese Bit sind von Bedeutung bei einigen Befehlen. Ein bestimmter Wert im *Flags Register* kann als Bedingung für Sprünge verwendet werden. Bei Additionen und Subtraktionen kann zusätzlich des Carry-Bit aufaddiert oder subtrahiert werden.

Für 16-Bit Operationen sind die Register `HL`, `DE`, `BC` vorgesehen. Hierbei dient das Register `HL` als 16-Bit Akkumulator. Die Ergebnisse vieler Operationen werden in diesem Register gespeichert. Arithmetische Operationen mit 16-Bit Operanden und Speicherzugriffe mit 16-Bit Adressen sind somit möglich.

Der komplette Registersatz mit den Registern `AF`, `HL`, `DE` und `BC` ist doppelt ausgelegt. Diese weiteren Register werden `AF′`, `HL′`, `DE′` und `BC′` genannt. Durch den Befehl `EXX` kann der Inhalt jedes dieser Register mit dem Inhalt des entsprechenden alternativen Registers getauscht werden. Hierdurch wird die Anzahl der vorhandenen Register erhöht. Über weitere Operationen kann ein Datenaustausch zwischen den beiden Registersätzen stattfinden. Allerdings können die alternativen Register nicht direkt als Operanden verwendet werden.

Bei den Registern `IX` und `IY` handelt es sich ebenfalls um 16-Bit Register, die in ihrer Funktion dem `HL`-Register ähneln. Diese Register sind nicht doppelt ausgeführt. Mit diesen Registern können Speicheradressierung und arithmetische Operationen vorgenommen werden.

Das `SP`-Register (*Stack Pointer Register*) ist ein 16-Bit Zeiger auf das oberste Element des Stackspeichers. Bei einem Stackspeicher wird immer auf das oberste Element zugegriffen. Somit wird bei einer Leseoperation das zuletzt auf dem Stack abgelegte Element zuerst zurückgegeben. Beim Speichern von Daten (`PUSH`) auf dem Stack wird der Zeiger inkrementiert, beim Lesen (`POP`) dekrementiert.

Das `PC`-Register (*Program Counter Register*) hat eine Breite von 16 Bit und zeigt immer auf den auszuführenden Befehl. Nach der Ausführung eines Befehls wird das `PC`-Register entsprechend der Länge des Befehls erhöht. Bei Sprüngen wird dem `PC` die Sprungadresse zugewiesen. Für den Fall, dass die Adresse des *Program Counters* in das *Extended Code Segment* fällt, wird zur Berechnung der tatsächlichen Speicheradresse der Wert des 8-Bit Registers `XPC` (*extension of program counter*) herangezogen. Die Speicheradressierung ist in Abschnitt 3.2.4 genauer erklärt.

Die Register `IP`, `IIR` und `EIR` dienen zur Steuerung der Interruptabarbeitung. Ausführlich erläutert wird diese in Abschnitt 3.2.5.

Befehlssatz der Rabbit-3000-CPU

Im Folgenden soll ein Überblick über einige wichtige Operationen gegeben werden. Die Instruktionen können verschiedene Längen haben. Auch in der Ausführungszeit gibt es entscheidende Unterschiede. In Tabelle 3.1 sind einige Befehle, die Länge ihrer Opcodes, die Ausführungszeiten in Takten und kurze Funktionsbeschreibungen aufgelistet.

Operation	**Länge des Opcodes in Byte**	**Ausführungszeit in Takten**	**Funktion**
`INC A`	1	2	Inkrementiert den Akkumulator um 1
`ADD A`,L	1	2	8-Bit Addition der *lower*-Bits des `HL`-Registers zum Akkumulator
`ADC HL,DE`	2	4	16-Bit Addition mit Berücksichtigung des Carry-Bit
`CP n`	2	4	Vergleicht den Akkumulator mit dem im zweiten Byte übergebenen Wert
`JP NZ,mn`	3	7	wenn ein vorheriger Vergleich die Gleichheit der Operanden ergab, wird an die übergebene 16-Bit Adresse gesprungen
`PUSH AF`	1	10	Sichert die Register `A` und `F` auf dem Akkumulator und erhöht den Stackpointer
`POP AF`	1	7	Stellt den Inhalt der Register `A` und `F` wieder her, dekrementiert den Stackpointer
`LD A,(DE)`	1	6	Lädt den 8-Bit Wert in den Akkumulator, auf den das Register `DE` zeigt
`LD A,(mn)`	3	9	Lädt den 8-Bit Wert in den Akkumulator, dessen 16-Bit Adresse mit übergeben wurde
`LD HL,(HL+d)`	3	11	16-Bit Speicherzugriff, 16-Bit Wert der Adresse, auf die `HL` + übergebener 8-Bit Wert zeigt, wird geladen
`RET`	1	8	Springt aus einer Funktion oder Routine zurück, in den `Program Counter` wird der Wert vom *Stack* geladen
`IPRES`	2	4	Setzt die Prioritätsstufe auf den zuvor gültigen Wert

Tabelle 3.1: Beispiele von Assemblerbefehlen

Wie der Tabelle zu entnehmen ist, benötigen sämtliche Befehle zur Ausführung länger als einen Takt. Der Prozessor besitzt damit eine CISC-Architektur. Es gibt viele von der Funktion identische Befehle, die auf verschiedenen Registern ausgeführt werden können und dabei unterschiedliche Ausführungszeiten haben. Eine komplette Übersicht der Instruktionen findet sich in [RabInst].

Zu beachten ist, dass die Rabbit-3000-CPU keine Assemblerbefehle für Fließkommaoperationen unterstützt. Für den Fall, dass trotzdem Fließkommazahlen verarbeitet werden müssen, muss auf entsprechende Bibliotheken zurückgegriffen werden, in denen diese Operationen aus mehreren der vorhandenen Operationen nachgebildet werden. Die Performance der Ausführung ist um ein Vielfaches langsamer als bei Prozessoren mit einer *floating point unit*. In [R3000UM, S. 4] ist bei einer Taktrate von 50 MHz eine Ausführungszeit von 7 μs für Additionen und Multiplikationen angegeben, eine Quadratwurzel benötigt 20 μs. Dies entspricht 350 bzw. 1000 Takten für eine Operation. Demzufolge eignet sich der Prozessor nur bedingt für Fließkommaoperationen.

Zugriff auf Register der internen und externen Komponenten

Wie bereits erwähnt, verfügt die Rabbit-3000-CPU über zahlreiche Komponenten, die über Register angesprochen werden. Auf Assembler-Ebene kann auf diese Register wie folgt zugegriffen werden: Jeden Befehl, der einen Speicherzugriff enthält, kann der Präfix `IOI` vorangestellt werden. Dieser Präfix hat eine Länge von einem Byte und bewirkt, dass der folgende Befehl nicht auf den normalen Speicher zugreift, sondern auf eines der Register. Anhand der Adresse in dem folgenden Befehl wird das Register ausgewählt, wobei nur das niederwertige Byte beachtet wird, da es sich um einen 8-Bit Adressraum handelt. Zu der regulären Ausführungszeit des folgenden Befehls addieren sich zwei Takte. Bei Schreibzugriffen kommt lediglich ein Takt hinzu, da der Schreibzugriff auf das Register schneller erfolgt als auf den Speicher.

Auf eine ähnliche Art kann auf die externen Komponenten zugegriffen werden. Hier wird der Präfix `IOE` vorangestellt. Je nach Konfiguration der Komponente verlängert sich die Ausführungszeit um bis zu 15 Takte. Weiteres zur Anbindung externer Komponenten ist in Abschnitt 3.2.9 zu finden.

3.2.4 Speicheranbindung

Im diesem Abschnitt geht es um die Speicheranbindung der Rabbit-3000-CPU. Der Anwender braucht die Details der Ansteuerung des Speichers nicht zu kennen, da innerhalb der Entwicklungsumgebung entsprechende Anpassungen des Codes vorgenommen werden. Es ist jedoch sinnvoll, die grundlegende Architektur der Speicheranbindung zu kennen. Dieses gilt insbesondere, wenn passagenweise in Assembler programmiert wird.

Die Rabbit-3000-CPU kann bis zu sechs Speicherchips mit je bis zu einem MByte ansprechen. Diese werden über die Datenleitungen D0 bis D7, die Adressleitungen A0 bis A19, sowie die *Chip-Select*-Leitungen (CS0, CS1, CS2), die *Output-Enable*-Leitungen (OE0, OE1) und die *Write-Enable*-Leitungen (WE0, WE1) angebunden. Jeweils zwei Speicherchips können sich eine *Chip-Select*-Leitung teilen, sie müssen sich dann aber in ihrer *Output-Enable* und *Write-Enable*-Leitung unterscheiden. Auf diese Weise erfolgt die Ansteuerung von bis zu sechs Speicherchips, da ein Chip nur aktiv wird, wenn die *Chip-Select*-Leitung und je nach gewünschter Operation

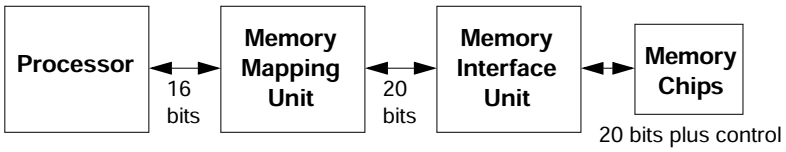


Abbildung 3.4: Ablauf der Speicheradressierung, Quelle: [R3000UM, S. 23]

die OE oder die WE aktiv sind. Es kann jedoch nur ein Bereich von einem MByte zur Zeit verwaltet werden. Dieser Bereich ist in vier Blöcke von jeweils 256 kByte unterteilt, denen dann jeweils ein Bereich des physikalischen Speichers zugewiesen wird.

Innerhalb eines Programms erfolgt eine Speicheradressierung mit 16 Bit. Dies hat zur Folge, dass direkt nur ein Bereich von 64 kByte ansprechbar ist. In Abbildung 3.4 ist der Ablauf der Speicheradressierung dargestellt. Die *Memory Mapping Unit* errechnet aus den 16-Bit Adressen die 20-Bit Zieladressen. Die Regeln der Berechnung sind im Folgenden erläutert:

Der 64 kByte Adressraum ist in vier Bereiche unterteilt:

- Root Segment (enthält den Startcode des Programms sowie die Interrupt-Service-Routinen)
- Data Segment (enthält Variablen)
- Stack Segment
- XPC Segment (enthält den größten Teil des Programmcodes)

Die Grenze zwischen dem Root und dem Data Segment sowie die zwischen dem Data und dem Stack Segment ist jeweils in vier kByte Schritten verschiebbar. Dieses erfolgt über das *SEGSIZE Register*, wobei jeweils vier Bit dieses Registers die Lage einer Grenze bestimmen. Das XPC Segment hat immer eine Größe von acht kByte. Eine typische Aufteilung der Bereiche ist folgende:

- Root Segment 12 kByte
- Data Segment 40 kByte
- Stack Segment 4 kByte
- XPC Segment 8 kByte

Von dem Bereich, der adressiert wird, hängt ab, wie die *Memory Mapping Unit* die 20-Bit Adresse erzeugt. Adressen, die in das Root Segment fallen, werden unverändert durchgeleitet, lediglich die vier höherwertigen Bit werden mit Nullen gefüllt. In den anderen Bereichen wird der 16-Bit Adresse der 8-Bit Wert des zugehörigen Registers (*Data Register*, *Stack Register*, *XPC Register*) aufaddiert. Die Addition erfolgt in der Weise, dass der 8-Bit Wert des zugehörigen Registers um zwölf Bit nach links verschoben wird und dann die 16-Bit Adresse aufaddiert wird. Grafisch dargestellt ist so eine Zuweisung der Speicherbereiche in Abbildung 3.5. Die so errechnete 20-Bit Adresse wird an die *Memory Interface Unit* weitergeleitet. Diese wird über die vier Register *MB0CR* bis *MB3CR* (*Memory Bank X Control Register*) konfiguriert, wobei jedes dieser Register die Zugriffe innerhalb eines Quadranten des Adressraumes steuert. Es kann eingestellt werden, welcher der Speicherchips angesteuert wird, und zusätzlich noch der Bereich gewählt werden, der beim Zugriff auf den Quadranten benutzt wird. Dieses erfolgt durch eine Manipulation der beiden höherwertigen Bit der Adresse. Notwendig wird dieses zum Beispiel,

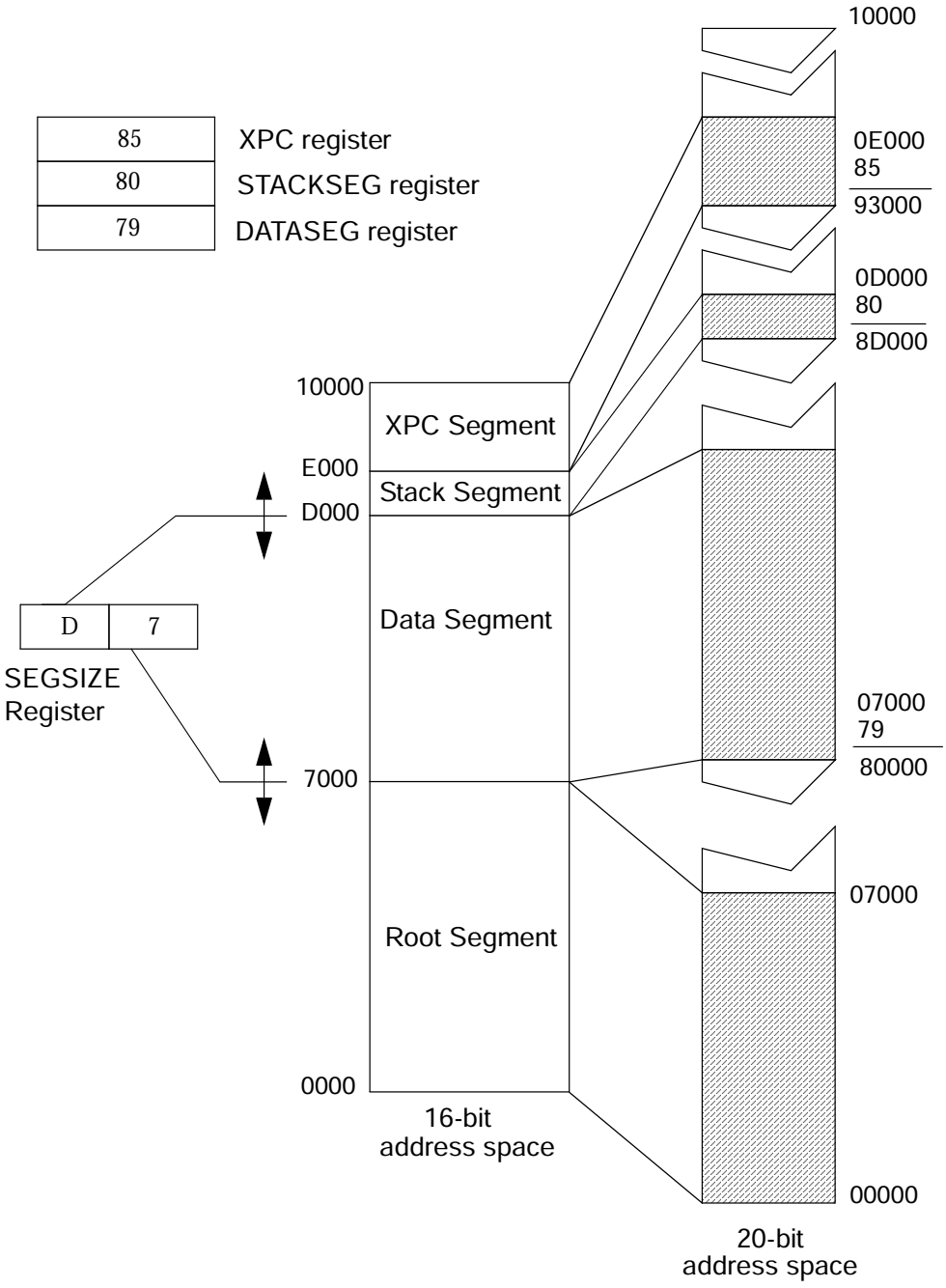


Abbildung 3.5: Beispiel für eine Speicheradressierung, Quelle: [R3000UM, S. 24]

wenn der dritte oder vierte Quadrant auf einen Speicherchip mit 512 kByte gelegt werden soll. Des Weiteren kann noch die Zahl der Wartezyklen an die verwendeten Module angepasst werden.

Durch die Adressierung mit 16-Bit Adressen ist der Instruktionssatz sehr kompakt. Dieses ermöglicht eine hohe Ausführungsgeschwindigkeit der Befehle. Jedoch ist der korrekte Speicherzugriff abhängig von dem Inhalt der entsprechenden Register. Soll zum Beispiel auf mehr als 40 kByte Daten zugegriffen werden, so ist bei jedem Zugriff sicherzustellen, dass das *Data Register* passend belegt ist. Dieses erfolgt in der Regel für den Anwender transparent durch entsprechende Anpassungen des Codes durch Dynamic C. Beim Zugriff auf die Daten über einen zuvor berechneten Zeiger kann der Compiler die Korrektheit des Zugriffs jedoch nicht prüfen und es besteht eine erhöhte Fehlergefahr. Besonders innerhalb einer Interrupt-Service-Routine (ISR) kann nicht vorhergesagt werden, ob entsprechende Register den richtigen Inhalt haben. Um die fehlerfreie Ausführung der ISR zu gewährleisten, müsste zum Start das *Data Register* mit dem gewünschten Wert beschrieben werden und gegen Ende der ursprüngliche Wert wiederhergestellt werden. Dieses erhöht jedoch die Ausführungszeit der ISR deutlich.

Eine Methode, die bei der Rabbit-3000-CPU angewendet werden kann, um den direkt adressierbaren Speicher für Code und Daten zu erhöhen, besteht darin, den Daten- und Befehlsraum zu trennen. Hierzu kann die Rabbit-3000-CPU wahlweise das Bit A19 und/oder das Bit A16 immer dann invertieren, wenn es sich um einen Datenzugriff handelt. Das Root Segment und das Data Segment werden dann über die gleichen 16-Bit Adressen angesprochen, es erfolgt aber ein Zugriff auf verschiedene Adressen. Dieser Modus wird über die Dynamic-C-Entwicklungsumgebung aktiviert. Die nötigen Anpassungen des Codes werden dann automatisch durchgeführt.

3.2.5 Interruptverarbeitung

Der folgende Abschnitt befasst sich mit der Interruptverarbeitung der Rabbit-3000-CPU. Die Grundlagen über Interrupts und die Interruptverarbeitung werden dabei ebenfalls kurz erläutert. Ausführliche Beschreibung finden sich zum Beispiel in [TG01] oder auch in [Cat05].

Interrupts sind Steuersignale, die dem Prozessor anzeigen, dass bei einer Komponente ein Ereignis aufgetreten ist und abgearbeitet werden muss. Der Ablauf der Programmausführung wird durch das Auftreten des Interrupts verändert. Einige Ursachen für das Auslösen eines Interrupts sind folgende:

- Es liegen Benutzereingaben vor, die verarbeitet werden sollen (zum Beispiel eine Tastatureingabe beim Standard-PC).
- Von einem Gerät oder einer Komponente wurden Daten empfangen (zum Beispiel serielle Schnittstellen oder Netzwerkschnittstelle).
- Ein Gerät oder eine Komponente wartet auf Daten (tritt ebenfalls bei seriellen Schnittstellen auf).
- Es handelt sich um periodische Interrupts zum regelmäßigen Ausführen von Code.

Das Auftreten eines Interrupts veranlasst den Prozessor dazu, die Ausführung des aktuellen Codeabschnitts zu stoppen und die ISR auszuführen. Innerhalb der ISR wird dann das aufgetretene Ereignis behandelt. Meist werden also entsprechende Datentransferoperationen durchgeführt. Für jede Komponente existiert eine eigene ISR, die dann ausgeführt wird, wenn diese Komponente einen Interrupt auslöst. Dazu wird der Interrupt Vektor in den *Program Counter*

geladen. Nach vollendeter Ausführung der ISR wird an der Stelle weitergearbeitet, an der die Ausführung zuvor unterbrochen wurde.

Der Vorteil der Benutzung von Interrupts ist vor allem, dass das System schnell auf Ereignisse reagieren kann und die nötigen zeitkritischen Operationen schnellstmöglich ausführen kann. Es muss zudem nicht ständig jede einzelne Komponente daraufhin überprüft werden, inwiefern dort Ereignisse aufgetreten sind, sondern dieses wird von der Komponente selbstständig über die Interruptleitung mitgeteilt.

Eine Besonderheit ergibt sich bei der Rabbit-3000-CPU durch die Verfügbarkeit der *Fast Interrupts*. Bei diesen Interrupts werden die Register des Prozessors nicht automatisch auf dem Stack gesichert und wiederhergestellt (vgl. [Cat05, S.12]). Dieses muss daher in der ISR erfolgen. Der Vorteil der *Fast Interrupts* liegt darin, dass die Ausführung der ISR schneller beginnen kann, zumal das zeitaufwändige Sichern des gesamten Registersatzes entfällt. Es müssen nur die Register gesichert werden, die auch tatsächlich benötigt werden. Somit lässt sich die Ausführungszeit der Interruptverarbeitung reduzieren.

Ein wichtiges Kriterium bei der Interruptverarbeitung eines Systems stellt die Interrupt-Latenz dar. Dieser Wert gibt die Zeit an, die zwischen Auftreten eines Interrupts und dem Starten der ISR vergeht. Die maximale Interrupt-Latenz ist davon abhängig, wie viele Komponenten Interrupts auslösen können und wie lange die ISR zur Ausführung benötigen. Die tatsächliche Zeit hängt von der momentanen Interruptaktivität der Komponenten ab und kann stark schwanken. Es wird somit meist für das *worst-case*-Szenario die Interrupt-Latenz errechnet. Generell ist eine möglichst niedrige Interrupt-Latenz anzustreben, damit es bei zeitkritischen Operationen nicht zu Problemen kommt.

Interrupts auf verschiedenen Prioritätsstufen

Beim Rabbit-3000-Prozessor können den verschiedenen Interrupts drei Prioritätsstufen zugeordnet werden. Der Bereich erstreckt sich von eins bis drei, wobei eins die niedrigste Priorität bedeutet und drei die höchste. Eine ISR niedriger Priorität kann von einer Interruptanforderung höherer Priorität unterbrochen werden, umgekehrt jedoch nicht. Dazu speichert der Prozessor in seinem `IP`-Register die aktuelle Prioritätsstufe, die von null bis drei reicht und somit zwei Bit Speicherplatz benötigt. Das `IP`-Register ist acht Bit breit und speichert neben der aktuellen Prioritätsstufe bis zu drei vorherige Prioritätsstufen, wobei die beiden *least-significant*-Bit die aktuelle Stufe repräsentieren. Im Normalfall beträgt die Prioritätsstufe, auf der gearbeitet wird, null. Tritt eine Interruptanforderung auf mit einer Prioritätsstufe, die höher als die aktuelle Stufe ist, so wird der Interruptanforderung stattgegeben.

Die Vergabe von verschiedenen Interruptprioritäten erlaubt es, die Komponenten mit besonders zeitkritischen Aufgaben bei der Ausführung ihrer ISR zu bevorzugen. Die Interrupt-Latenzen einiger Komponenten können somit stark verringert werden, da diese Komponenten nicht mehr auf die Ausführung der ISR von Komponenten mit niedriger Priorität warten müssen. Ein effizientes Design von echtzeitfähigen Systemen ist somit möglich. Welche Komponenten priorisiert werden, ist vom Einsatzzweck abhängig. Bei der Anbindung von zwei Lasermesssystemen sind dies in erster Linie die beiden zur Anbindung verwendeten seriellen Schnittstellen. Es werden bei relativ hoher Datenrate Telegramme empfangen. Wie in 2.5.3 erläutert, müssen pro Sekunde

ungefähr $2 \cdot 27450$ Byte bearbeitet werden, wobei der Verlust eines Bytes unter allen Umständen zu vermeiden ist.

Ablauf der Interruptverarbeitung

Im Folgenden wird der Ablauf der Interruptverarbeitung der Rabbit-3000-CPU dargestellt. Zugrunde gelegt wird, dass es zu einer Interruptanforderung kommt und die Priorität des Interrupts größer ist als die aktuelle Prioritätsstufe. In diesem Fall werden nacheinander folgende Operationen durchgeführt:

- Es wird gewartet, bis der aktuelle Befehl komplett ausgeführt ist.
- Die Bit des `IP`-Registers werden zwei Stellen nach links verschoben und die aktuelle Prioritätsstufe wird auf die Priorität des aufgetretenen Interrupts gesetzt.
- Der Wert des `PC`-Registers wird auf dem Stack gespeichert und die Startadresse der ISR wird in das Register geladen.
- Es wird nun die ISR ausgeführt.

Nachdem die kritischen Operationen dieser Routine ausgeführt sind, wird innerhalb der ISR mit dem Befehl `IPRES` ein Zurücksetzen der aktuellen Prioritätsstufe auf den Wert vor dem Auftreten des Interrupts ausgelöst. Dieser Befehl verschiebt die Bit des `IP`-Registers um zwei Stellen nach rechts und löscht somit die aktuelle Prioritätsstufe. Schließlich erfolgt mit einem `RET` ein Zurückkehren zur Stelle, an der die vorherige Befehlsfolge unterbrochen wurde. Dazu wird vom Stack der vorherige Wert des `PC`-Registers geladen. Zwischen den Befehlen `IPRES` und `RET` können noch weitere Operationen der ISR ausgeführt werden. Es wird dann auf der Prioritätsstufe weitergearbeitet, die vor dem Interrupt gegeben war. Die noch ausstehenden Befehle können dann aber von anderen Interruptanforderungen unterbrochen werden, wenn deren Priorität größer ist als die Prioritätsstufe vor dem Interrupt, der gerade verarbeitet wird.

Durch die frühe Freigabe weiterer Interrupts kann die Interruptlatenz verringert werden, da weitere Interruptanforderungen früher gestartet werden können. Es ist dabei aber zu beachten, dass es zu einer stark verschachtelten Ausführung von Interrupts kommen kann. Denkbare Konsequenz wäre ein Überlaufen des Stacks, da bei jedem Start einer ISR Daten auf dem Stack abgelegt werden und erst gegen Ende der Routine wieder abgeholt werden. Dieses Überlaufen ist aber nur bei sehr starker Interruptaktivität denkbar, die auch ohne Einsatz der frühen Interruptfreigabe zu Instabilitäten führt. Der Prozessor wäre dann nur noch mit Abarbeitung von Interrupts beschäftigt und kommt nicht mehr zur Ausführung des eigentlichen Programms.

Anforderungen an die Interrupt-Service-Routine

Die ISR für die verschiedenen Komponenten müssen mehrere Kriterien erfüllen, um ein einwandfreies Funktionieren des Systems zu gewährleisten.

- Die Ursache des Interrupts muss abgearbeitet werden, so dass die entsprechende Komponente ihre Interruptanforderung zurückzieht.
- Sämtliche Prozessorregister müssen nach der ISR unverändert geblieben sein.
- Die Ausführungszeit muss in einem angemessenen Rahmen liegen.

Die verschiedenen Komponenten der Rabbit-3000-CPU haben jeweils festgelegte Bedingungen, unter denen Interrupts ausgelöst werden. Ebenso sind Bedingungen festgelegt, unter denen die Interruptanforderungen wieder zurückgezogen werden. Bei vielen Komponenten werden die Interruptanforderungen durch das Auslesen des zugehörigen Statusbyte zurückgesetzt. Komplizierter gestaltet sich das Zurücksetzen der Interruptanforderung bei den seriellen Schnittstellen. Dieses ist in 3.2.8 ausführlich erklärt. Wird aufgrund einer fehlerhaft geschriebenen ISR die Interruptanforderung nicht zurückgesetzt, so wird sofort nach Verlassen der ISR aufgrund immer noch vorhandener Interruptanforderung die gleiche ISR wieder ausgeführt. Dieses führt zu einer Endlosschleife und der eigentliche Programmcode kann nicht mehr ausgeführt werden. Ein Systemabsturz ist die Folge.

Die nächste wichtige Anforderung ist, dass der Zustand der Prozessorregister durch die ISR nicht dauerhaft verändert werden darf. Der Grund dafür ist folgender: Durch den Interrupt werden laufende Operationen unterbrochen. Innerhalb dieser Operationen werden Daten in den Prozessorregistern zwischengespeichert. Nach der ISR laufen die Berechnungen an der Stelle weiter, an der die Unterbrechung stattgefunden hat. Werden innerhalb der ISR die Registerinhalte verändert und die ursprünglichen Werte nicht wieder zurückgeschrieben, kommt es zu Fehlern, die zum kompletten Absturz des Programms führen können.

Die Register werden zwingend für Operationen innerhalb der ISR benötigt. Es muss also eine Sicherung und anschließende Wiederherstellung der Registerinhalte durchgeführt werden. Diese Sicherung erfolgt auf dem Stackspeicher, auf den mit den Befehlen `PUSH` und `POP` zugegriffen wird. Bei der Sicherung werden alle Register, die innerhalb der ISR verändert werden, über `PUSH` Operationen gesichert. Der Beginn einer ISR könnte also folgendermaßen aussehen:

```
PUSH AF
PUSH HL
PUSH DE
```

Die Wiederherstellung der ursprünglichen Registerinhalte erfolgt in umgekehrter Reihenfolge:

```
POP DE
POP HL
POP AF
```

Die Ausführungszeit einer ISR wird in Taktzyklen des Prozessors angegeben. Es ist beim Entwurf einer ISR zu berücksichtigen, wie häufig Interruptanforderungen voraussichtlich auftreten werden und welchen Teil der Prozessorauslastung der entsprechenden Komponente zugestanden werden soll. Diese Festlegung muss immer im Hinblick auf ein funktionierendes Gesamtsystem abgewogen werden. Ebenfalls muss bei der Ausführungszeit der ISR bedacht werden, das im *worst-case*-Szenario keine Latenzzeit größer als zulässig ist. Wie groß die maximal zulässige Latenzzeit ist, hängt von der Art der Komponente und von der Konfiguration ab.

Manuelles Setzen von Prioritätsstufen

In bestimmten Situationen kann es erforderlich sein, auftretende Interrupts zu unterdrücken. Solch eine Situation liegt vor, wenn mit Daten gearbeitet wird, die auch in der ISR bearbeitet werden. Hier könnte ein Ausführen der ISR zu einem unpassenden Zeitpunkt Inkonsistenzen

in den Daten hervorrufen. In solchen Abschnitten kann die aktuelle Prioritätsstufe durch die Befehle `IPSET 0` bis `IPSET 3` angepasst werden. Durch diesen Befehl werden die Bit des `IP`-Registers um zwei Stellen nach links verschoben und die aktuelle Priorität auf den gewünschten Wert gesetzt. Nun wird dementsprechend nur noch solchen Interruptanforderungen stattgegeben, deren Priorität höher als die gesetzte ist. Zurückgesetzt wird die Prioritätsstufe mit dem Befehl `IPRES`.

Das Anwenden dieser Methode muss mit großer Vorsicht erfolgen. Bestimmte vorgefertigte Funktionsaufrufe aus Bibliotheken dürfen innerhalb einer solchen Passage nur ausgeführt werden, wenn die Stufe nicht höher als eins ist. Dies liegt daran, dass es innerhalb dieser Funktionen ebenfalls zu interruptgesteuerten Aktionen kommt, die bei höherer Priorität unterdrückt werden würden.

Manchmal kann es auch sinnvoll sein, innerhalb einer ISR die Priorität herabzusetzen. Die ISR kommt dann mit geringer Latenz zur Ausführung, kann aber durch noch zeitkritischere Aufgaben unterbrochen werden. Beim Setzen der niedrigeren Priorität kann es jedoch zu einem Überlaufen des `IP`-Registers kommen. Für den Fall, dass nach dem manuellen Setzen der Priorität nacheinander einige Interrupts mit schrittweise ansteigender Priorität stattfinden, gehen Informationen über die ursprüngliche Priorität verloren, da nur drei vorherige Stufen gespeichert werden können. Dieses kann durch vorheriges Sichern des `IP`-Registers auf dem Stack verhindert werden.

Adresse der ISR

Nach dem Auftreten eines Interrupts muss ein Sprung zur Startadresse (Interrupt Vector) der ISR stattfinden. Diese Adresse hat eine Länge von 16 Bit, die sich wie folgt zusammensetzen. Die sieben höherwertigen Bit werden dem `IIR`-Register (*Internal Interrupt Register*) entnommen. Diese Bit sind somit für alle internen Interrupts identisch. Der Registerinhalt kann jedoch bei Bedarf verändert werden. Die folgenden fünf Bit sind fest vorgegeben und unterscheiden die verschiedenen Komponenten. Die letzten vier Bit sind immer null für alle Adressen. Es ergibt sich durch diese Aufteilung des Speicherbereichs eine maximale Länge der ISR von 16 Byte. Somit ist dieser Bereich nur für extrem kurze ISR ausreichend. Daher erfolgt in der Regel ein Sprung zu einem ausgelagerten Teil der Routine. Der Speicherort der ISR wird von der Entwicklungsumgebung bestimmt und der Sprung zu dem ausgelagerten Teil wird automatisch eingefügt.

Externe Interrupts

Neben den Komponenten innerhalb der Rabbit-3000-CPU können auch externe angebundene Geräte (Netzwerkcontroller, weitere serielle oder parallele Schnittstellen) Interrupts erzeugen. Hierzu stehen zwei Interruptleitungen zur Verfügung. Die externen Interrupts unterscheiden sich von den internen in folgenden Punkten:

- Bei der Adresse werden die ersten sieben Bit dem `EIR`-Register (*External Interrupt Register*) entnommen
- Sobald die ISR zur Ausführung kommt, wird die Interruptanforderung gelöscht

Die Art der Verarbeitung des Interrupts muss entsprechend der Spezifikation des angeschlossenen Geräts erfolgen. Sollen mehr als zwei Geräte angebunden werden, müssen deren Interruptleitungen über eine OR-Logik verknüpft werden und die ISR entsprechend angepasst werden.

Besonderheiten

Bei dem Abarbeiten der Interrupts kann es unter bestimmten Bedingungen zu Besonderheiten kommen. Bei einigen Befehlen wird zusätzlich noch der darauf folgende Befehl ausgeführt, bevor der Sprung zur ISR vollzogen wird. Dieses kommt bei Befehlssequenzen vor, die in der Regel die Struktur des Stacks verändern und atomar ausgeführt werden müssen. Auf diese Weise wird die Konsistenz des Stacks gesichert. Für den Anwender ergeben sich dadurch keine Änderungen, es erhöht sich lediglich die Interrupt-Latenz.

3.2.6 Steuerung der Taktrate

Im Folgenden werden die Komponenten beschrieben, die für die Verarbeitung und Weiterleitung der Taktsignale verantwortlich sind. Zu diesen Komponenten gehören *Fast Oscillator*, *Spectrum Spreader*, *Clock Doubler*, *Global Power Save & Clock Distribution* und *32,786 kHz Clock Input.*

Der Rabbit 3000 verfügt über die Möglichkeit, die Taktrate innerhalb des laufenden Programms zu ändern. Zum Stromsparen lässt sich der Prozessortakt über einen Divisor um den Faktor zwei, vier, sechs oder acht verringern. Noch weiter geht folgender Betriebsmodus: Im „Sleepy“-Modus wird der Rabbit nicht von dem Standardoszillator, sondern vom Quarz der Echtzeituhr getaktet, was eine Taktrate von 32,768 kHz ergibt. Auch dieser Takt lässt sich bei Bedarf noch auf bis zu 2 kHz herabsetzen („Ultra-Sleepy“-Modus). Der Standardtaktgenerator kann hierbei sogar komplett abgeschaltet werden. Die Leistungsaufnahme sinkt durch diese Maßnahmen auf etwa 0,33 mW, was für den Einsatz in batterie- oder akkubetriebenen Applikationen nützlich ist. Während dieses Stromsparmodus können trotzdem in entsprechend niedriger Geschwindigkeit Berechnungen durchgeführt werden. Es können beispielsweise regelmäßig Eingangsports überprüft werden, um bei einer Veränderung des Zustandes das Aufwachen aus dem Schlafzustand zu veranlassen. Schreiboperationen auf entsprechende Register führen zum Wechsel in den jeweiligen Betriebsmodus.

Es wird zwischen Kerntakt des Prozessors und Peripherietakt unterschieden. Der Kerntakt bestimmt die Geschwindigkeit der eigentlichen Rechenoperationen. Komponenten wie die seriellen Schnittstellen oder die PWM-Einheiten werden mit dem Peripherietakt versorgt. Es gibt Betriebszustände, in denen sich der Kerntakt vom Peripherietakt unterscheidet, jedoch kann der Peripherietakt niemals geringer als der Prozessortakt sein.

Im Folgenden soll nun auf die Funktionsweise der Takterzeugung und Taktweiterleitung innerhalb des Rabbit 3000 eingegangen werden. Dieser Abschnitt wird hier besonders ausführlich behandelt, da die Geschwindigkeit der seriellen Schnittstellen direkt vom Peripherietakt abhängt. Fehlerhafte Einstellungen hätten ein Scheitern der Datenübertragung zwischen den Lasermesssystemen und dem Powercore 3800 zur Folge.

Taktquellen

Wie schon erwähnt, verfügt der Prozessor über zwei verschiedene Taktquellen, zum einen den Standardtaktgenerator und zum anderen den Taktgenerator für die Echtzeituhr.

Der Standardtaktgenerator ist mit wenigen externen Bauteilen zu realisieren. Die erforderliche Schaltung ist in [R3000UM, S. 211] dargestellt, dort ist jedoch ein 11,0592 MHz Quarz verwendet. Der Takt des Quarzes kann innerhalb des Rabbit 3000 verdoppelt werden. „XTALB1“ und „XTALB2“ stehen für die Pins des Prozessors, die mit dem Standardtaktgenerator verbunden werden. Die benötigte elektronische Schaltung zum Treiben des Quarzes ist innerhalb des Prozessors realisiert. Hierfür ist die Komponente *Fast Oscillator* verantwortlich.

Der Schaltkreis für die Echtzeituhr ist hingegen komplett extern zu realisieren. Über einen Pin wird das entsprechende Taktsignal dem Rabbit 3000 zugeführt, von der Komponente *32,786 kHz Clock Input* entgegengenommen und zur weiteren Benutzung bereitgestellt.

Weiterleitung

Bevor das Taktsignal dem Kern des Prozessors zugeführt wird, werden noch diverse Stufen durchlaufen, die jeweils das Taktsignal verändern können. In Abbildung 3.6 sind die Verteilungswege der Taktsignale dargestellt. Das Standardtaktsignal durchläuft zunächst den *Spectrum Spreader*. Diese Einheit beeinflusst die Taktrate nicht, sondern sorgt bei Bedarf für eine Absenkung der elektromagnetischen Störstrahlung. Die Funktion und die technischen Hintergründe sollen hier nur oberflächlich beschrieben werden. Zum Verständnis sind umfangreiche Kenntnisse der Signaltheorie nötig. An dieser Stelle soll auf [Lue99] verwiesen werden.

Wenn die elektromagnetische Störstrahlung eines Mikroprozessors im Frequenzspektrum betrachtet wird, zeigen sich besonders starke Emissionen von elektromagnetischer Strahlung auf der Frequenz der Taktrate und den Vielfachen dieser Taktrate. Ziel des *Spectrum Spreader* ist nun, diese Störenergie auf einen breiteren Frequenzbereich zu verteilen. Dazu werden die Taktflanken mit einer variablen Verzögerung weitergeleitet. Die Verzögerungszeit wird laufend verändert, so dass einige Takte kürzer und einige Takte länger ausfallen. Die Gesamttaktrate bleibt über mehrere Takte gemittelt jedoch unverändert. Durch die ständig variierende Frequenz wird auch die Störstrahlung auf verschiedene Frequenzen verlagert. Durch diese Maßnahmen können die Richtlinien der CE[6] oder der FCC[7] problemlos erfüllt werden.

Zu Problemen kann es kommen, wenn die CPU auf maximal zulässiger Taktrate betrieben wird. Hier würde ein Zuschalten des *Spectrum Spreader* bewirken, das einzelne Taktzyklen so kurz werden, dass die Transistoren nicht mehr korrekt schalten können und es zu Instabilitäten kommt. Des Weiteren wirken sich die variierenden Taktflanken auch auf das Timing der Schnittstellen zum Datentransfer aus und es könnten unter Umständen Fehler auftreten. Zu steuern ist der *Spectrum Spreader* mit zwei Registern:

- Über das *Global Clock Modulator 0 Register* (GCM0R) lässt sich die Komponente aktivieren oder deaktivieren.

[6] Communauté Européenne (französischer Begriff für Europäische Gemeinschaft)
[7] U.S. Federal Communications Comission

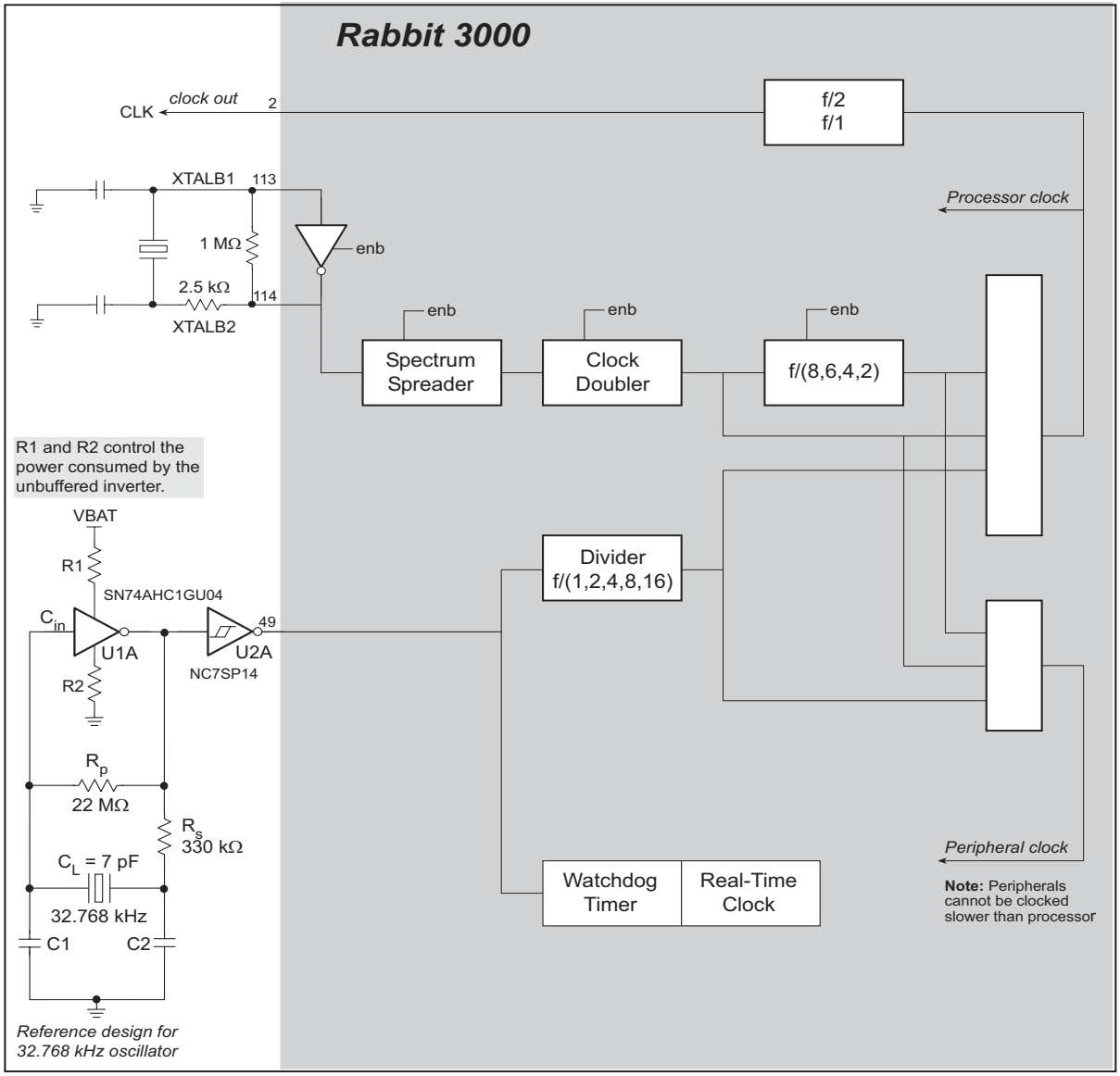


Abbildung 3.6: Verteilungswege der Taktsignale, Quelle: [R3000UM, S. 81]

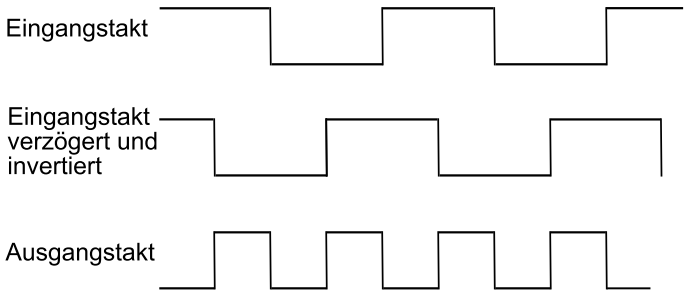


Abbildung 3.7: Funktionsweise eines Taktverdopplers

- Das *Global Clock Modulator 1 Register* (GCM1R) steuert den Grad der Spreizung des Spektrums.

In der Grundeinstellung ist der *Spectrum Spreader* deaktiviert. Bei der Entwicklung des Systems zum Betrieb zweier SICK LMS 200 soll kein Gebrauch vom *Spectrum Spreader* gemacht werden, solange keinerlei Probleme mit elektromagnetischer Störstrahlung auftreten. Hiermit ist kaum zu rechnen, da zum einen sämtliche Geräte am Serviceroboter durch eigene Abschirmung ausreichend geschützt sind und zum anderen der massive Metallkörper des Roboters die Störstrahlung abhalten wird.

Als nächste Stufe passiert das Signal den *Clock Doubler*. Hier wird bei Bedarf die Taktrate verdoppelt. Im Standardfall wird hiervon immer Gebrauch gemacht, da aus Performancegründen die höchstmögliche Taktrate gewünscht wird. Dementsprechend wird im Hardwaredesign ein Quarz mit halber Frequenz verwendet. So wird für eine Taktrate von 51,6 MHz beim Powercore 3800 ein Quarz mit einer Taktrate von 25,8 MHz eingesetzt.

Technisch funktioniert der *Clock Doubler* auf folgende Weise: Der Eingangstakt wird verzögert, invertiert und mit dem unverändertem Eingangstakt über eine XOR-Operation verknüpft. Das Ergebnis dieser Operation stellt nun den neuen Ausgangstakt dar. Grafisch dargestellt ist die Funktionsweise des *Clock Doubler* in Abbildung 3.7 Die korrekte Funktion hängt von der Wahl der richtigen Verzögerungszeit ab. Theoretisch ergibt sich ein optimaler Ausgangstakt, wenn die Verzögerungszeit genau 1/4 des Eingangstaktes ist. Ein Abweichen von diesem Wert hat zur Folge, dass der Ausgangstakt asymmetrisch wird, das heißt, dass das Verhältnis zwischen *low*-Pegel und *high*-Pegel nicht genau 50 zu 50 ist, sondern beispielsweise 45 zu 55. Grobe Abweichungen können Fehlfunktionen hervorrufen und es kommt ebenfalls zu Instabilitäten.

Es ist zu beachten, dass ein gleichzeitiger Betrieb von *Spectrum Spreader* und *Clock Doubler* zu besonders starken Abweichungen des Verhältnisses zwischen *low*-Pegel und *high*-Pegel führt. Zur Steuerung des *Clock Doublers* ist das *Global Clock Double Register* (GCDR) mit dem entsprechenden Wert zu beschreiben. Es kann die Verdopplung des Taktes komplett abgeschaltet werden oder eine entsprechende Verzögerungszeit eingetragen werden. In Tabelle 7-7 im Rabbit 3000 User's Manual [R3000UM, S. 83] sind die Verzögerungszeiten für verschiedene Taktquarze angegeben. Für den verwendeten Quarz von 25,8 MHz wird eine Verzögerungszeit von 8 ns empfohlen. Der theoretisch optimale Wert ist 9,7 ns, weshalb der nächstmöglich einstellbare Wert von 10 ns zu wählen wäre. Bei einem Test lief das System mit beiden Einstellungen stabil. Die Taktverdopplung kann auch über die Entwicklungsoberfläche Dynamic C aktiviert

werden. Dabei werden dem Programm entsprechende Anweisungen zugefügt und der Benutzer muss nicht in die Taktregelung eingreifen.

Die nächste Stufe in der Taktregelung stellt die *Global Power Save & Clock Distribution* genannte Einheit dar. Eingang dieser Stufe sind sowohl der Ausgangstakt des *Clock Doubler* als auch der Ausgang des *32,786 kHz Clock Input*. Aufgabe dieser Einheit ist es, den Kerntakt der CPU und den Peripherietakt mit den gewünschten Taktraten zu versorgen. Hierbei kann entweder der Takt des *Clock Doubler* (bei Bedarf um den Faktor zwei, vier, sechs oder acht verringert) oder der Takt des *32,786 kHz Clock Input* (bei Bedarf bis zu Faktor 16 verringert) gewählt werden. Gesteuert werden die Takte über zwei Register:

- Das *Global Control/Status Register* (GCSR) dient zur Wahl der Taktquelle und zum Einstellen des Teilers des Standardtaktes.
- Das *Global Power Save Control Register* (GPSCR) erlaubt, den Takt des *32,786 kHz Clock Input* noch zu teilen.

Über letzteres Register lässt sich zusätzlich eine weitere Stromsparmöglichkeit zuschalten, die das Timing des Speichers an den verringerten Takt anpasst und damit den Stromverbrauch des Speichers verringert.

Der Standardfall ist, dass sowohl für CPU als auch Peripherietakt direkt der Ausgang des *Clock Doubler* gewählt wird. Bei Änderung des Peripherietaktes ändert sich das Timing verschiedener Komponenten. Auch die seriellen Schnittstellen ändern ihre Baudrate und müssen zur Beibehaltung der ursprünglichen Baudrate neu konfiguriert werden. Da dieses nicht gewünscht ist, soll während der kompletten Laufzeit mit vollem Peripherietakt gearbeitet werden. Es besteht jedoch die Möglichkeit, bei vollem Peripherietakt den CPU-Takt auf ein Achtel zu reduzieren. Dadurch ließe sich bei Leerlauf die Stromaufnahme verringern. Da auf dem Serviceroboter der Stromverbrauch des Rabbit Powercore 3800 zu vernachlässigen ist, wird im Rahmen dieser Arbeit keine Implementierung der Stromsparfunktionen angestrebt.

Es kann also bei allen folgenden Ausführungen stets davon ausgegangen werden, dass zum einen der Prozessorkern mit einer Taktrate von 51,6 MHz läuft und zum anderen die vom Peripherietakt abhängigen Komponenten mit einem konstanten Takt von ebenfalls 51,6 MHz versorgt werden. Diese Voraussetzungen erleichtern die Entwicklung der Anwendungssoftware erheblich. Es können zuverlässige Aussagen über die Performance der CPU gemacht werden und die Taktung der Schnittstellen gestaltet sich deutlich einfacher.

3.2.7 Das Timersystem

Der Rabbit-3000-Mikroprozessor besitzt ein umfangreiches Timersystem. Mit diesem System ist es möglich, den Peripherietakt auf verschiedene weitere Komponenten zu verteilen und dabei individuell zu skalieren. Hierbei können die einzelnen Komponenten nahezu unabhängig voneinander getaktet werden. Die Aufgaben des Timersystems sind folgende:

- Taktung der sechs seriellen Schnittstellen
- Taktung von *Input Capture Unit*, *PWM Unit* und *Quadrature Decoder Unit*
- Timing der Datenübertragung über parallele Schnittstellen
- Ausführungszeitpunkte für spätere Aufgaben festlegen und per Interrupt Ausführung veranlassen

- Erzeugen von periodischen Interrupts

Den Aufbau des Timersystems zeigt Abbildung 3.8. Es wird dort zwischen Timer A und Timer B unterschieden, die verschiedene Aufgaben erfüllen. Das Timersystem A ist für die Taktung der seriellen Schnittstellen, der *Input Capture Unit*, *PWM Unit* und *Quadrature Decoder Unit* verantwortlich. Timer B ermöglicht das Planen zukünftiger Aufgaben. Beide Timersysteme können periodische Interrupts erzeugen. Bei den parallelen Schnittstellen können verschiedene Timerquellen ausgewählt werden.

Eine Interaktion beider Timersysteme besteht nur darin, dass als Eingang für die Taktung des Timersystems B ein Ausgang des Timers A gewählt werden kann.

Das Timersystem A

Als Takteingang für das gesamte Timersystem A steht der Peripherietakt oder der halbierte Peripherietakt zur Auswahl. Diese Einstellung beeinflusst alle weiteren Timer des Systems. Eine Schreiboperation auf das *Timer A Prescale Register* (TAPR) ermöglicht es, die gewünschte Betriebsart auszuwählen.

Hauptbestandteile des Timersystems A sind die zehn Timer A1 bis A10, die in ihrer Funktionsweise prinzipiell identisch sind. Jeder dieser Timer kann seinen Eingangstakt skalieren, also verringern. Der Faktor, um welchen der Timer den Takt verringern soll, wird in das jeweilige *Timer Register* mit dem Namen *Timer A Time Constant X Register* (TATXR) geschrieben, wobei das „X“ für die Zahl zwischen 1 und 10 steht. Aufgrund der 8-Bit breiten Register können Werte zwischen 0 und 255 geschrieben werden. Es wird dann um den Faktor (TATXR+1) skaliert. Der Wert 0 bedeutet also, dass der Eingangstakt unverändert weitergeleitet wird, ein Wert von 1 bewirkt eine Halbierung der Taktrate, bei einem Wert von 255 ist der Ausgangstakt gleich 1/256 des Eingangstaktes. Grafisch dargestellt ist diese Funktionsweise in Abbildung 3.9. Jeder Timer funktioniert als 8-Bit Abwärtszähler. Bei jedem Eingangstakt wird ein Schritt zurückgezählt. Erreicht der Zählerstand den Wert null, wird ein Ausgangspuls generiert und in den Zähler der Wert aus dem *8-Bit Reload Register* geladen, der dem zuvor gespeicherten Wert im TATXR entspricht. Dieser Vorgang wiederholt sich permanent und es wird so die Skalierung des Taktes erreicht.

Im Aufbau des Timersystems A nimmt der Timer A1 eine Sonderstellung ein. Der Ausgang dieses Timers treibt keine Peripheriekomponente, sondern kann als Eingang für die Timer A2 bis A7 gewählt werden. Es kann für die Timer A2 bis A7 aber auch jeweils unabhängig voneinander der Haupttakt des Timersystems A als Eingang gewählt werden, also der Peripherietakt oder die Hälfte davon, je nachdem, was zuvor eingestellt wurde. Dadurch wird eine Kaskadierung von zwei Timern erreicht. Wenn ein Teiler von 256 nicht ausreichend ist, kann somit der Gesamtteiler auf $256 \cdot 256 = 65536$ erhöht werden - bei vorheriger Halbierung des Peripherietaktes sogar auf 131072. Entsprechende Konfigurationen lassen sich jederzeit beliebig ändern, wenn auch eine laufende Datenübertragung dadurch gestört würde. Die Taktausgänge der Timer A2 bis A7 dienen als Eingangstakt für die seriellen Schnittstellen.

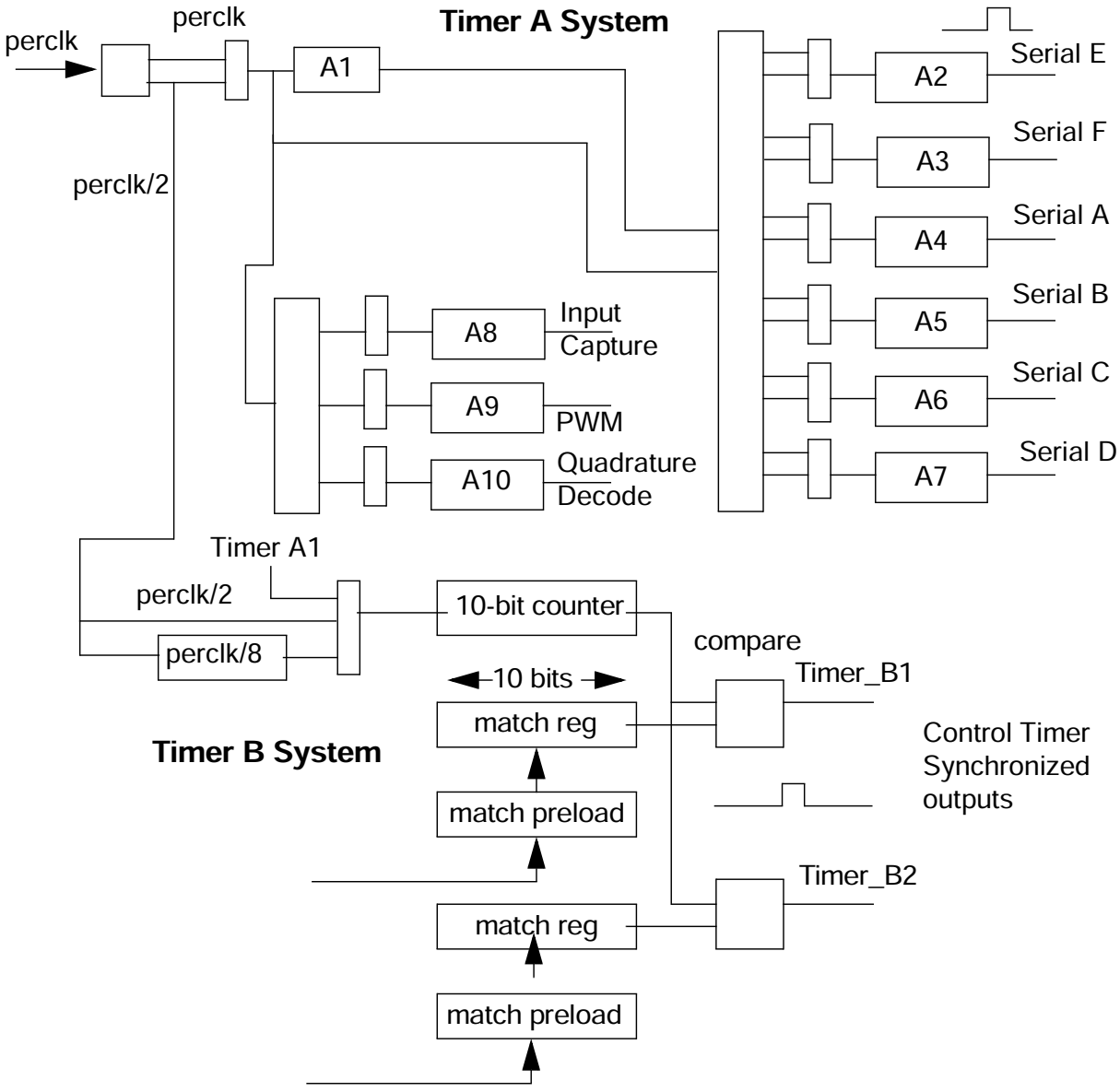


Abbildung 3.8: Aufbau des Timersystems, Quelle: [R3000UM, S. 149]

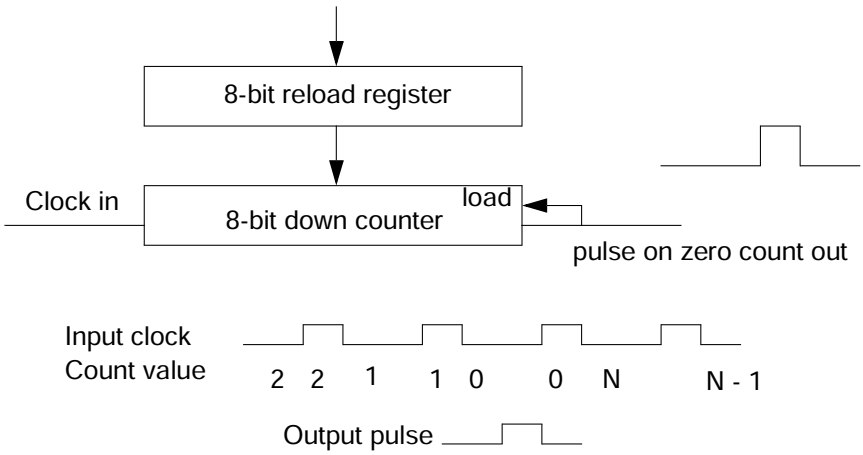


Abbildung 3.9: Funktionsweise eines einzelnen Timers, Quelle: [R3000UM, S. 150]

Verwendung des Timersystems zur Ansteuerung der Lasermesssysteme Im Folgenden wird beschrieben, wie die Konfiguration des Timersystems im Hinblick auf den geplanten Verwendungszweck des Rabbit Powercore 3800 durchzuführen ist. Es sollen zunächst die Anforderungen aufgezählt werden, die an eine optimale Konfiguration des Timersystems zu stellen sind.

- Der serielle Port A soll permanent auf einer Baudrate von 115,2 kBd arbeiten (Debug-Verbindung zum PC).
- Der serielle Port E soll wahlweise 9,6 kBd, 19,2 kBd, 38,4 kBd oder 500 kBd Baudrate schalten können zur Anbindung eines SICK LMS 200.
- Der serielle Port F dient zur Anbindung des zweiten SICK LMS 200 und soll ebenfalls entsprechende Baudraten beherrschen.
- Die Wahl der Baudraten auf Port E und Port F muss unabhängig voneinander erfolgen können.

Wie in Abschnitt 3.2.8 ausgeführt wird, können die seriellen Schnittstellen einzeln auf 8- oder 16-fache Überabtastung eingestellt werden. Am Ausgang des Timersystems muss also jeweils das 8 oder 16-fache des gewünschten Schritttaktes anliegen.

Das am schwierigsten zu erfüllende Kriterium ist der Schritttakt von 500 kBd, da es sich hierbei nicht um eine Standardbaudrate handelt. Ausgangspunkt ist eine Taktrate von 51,6 MHz, die dann geteilt durch einen ganzzahligen Teiler und durch wahlweise 8 oder 16 einen Wert von 0,5 MHz ergeben muss, wobei eine Abweichung bis 3 % nicht als kritisch zu betrachten ist.

Es wird nun der Teiler für den x16-Modus berechnet:

$$\frac{51,6 \text{ MHz}}{t \cdot 16} = 0,5 \text{ MHz} \Rightarrow \ t = 6,45$$

Der nächstmögliche ganzzahlige Teiler wäre 6, was zu einer Abweichung der Baudrate um 8 % führen würde. ($\frac{6,45}{6} = 1,08$) Dieses ist nicht akzeptabel, weswegen der sonst zu favorisierende x16-Modus nicht geeignet ist.

Entsprechende Berechnung nun für den x8-Modus:

$$\frac{51,6 \text{ MHz}}{t \cdot 8} = 0,5 \text{ MHz} \Rightarrow \ t = 12,90$$

Hier wäre der zu wählende ganzzahlige Teiler 13 mit einer Abweichung von 0,8 %. ($\frac{12,9}{13} = 0,992$) Eine Abweichung von 0,8 % kann auch bei einer hohen Baudrate von 500 kBd als unkritisch angenommen werden.

Aufgrund dieser Berechnungen steht fest, dass die mit den LMS 200 verbundenen seriellen Schnittstellen im x8-Modus arbeiten werden. Da die Zahl 13 nicht durch 2 teilbar ist, muss das Timersystem A mit dem vollen Peripherietakt betrieben werden. Ausgehend von diesen festgelegten Werten werden die Teiler für die übrigen Schrittraten berechnet. Der serielle Port A wird nur auf der Standardgeschwindigkeit von 115,2 kBd betrieben, welche auch mit dem x16-Modus erreicht werden kann.

115,2 kBd:

$$t = \frac{51,6 \text{ MHz}}{0,1152 \text{ MHz} \cdot 16} = 27,99$$

Zu wählen ist 28.

38,4 kBd:

$$t = \frac{51,6 \text{ MHz}}{0,0384 \text{ MHz} \cdot 8} = 167,97$$

Zu wählen ist 168.

19,2 kBd:

$$t = \frac{51,6 \text{ MHz}}{0,0192 \text{ MHz} \cdot 8} = 335,94$$

Zu wählen wäre 336, dieses ist nicht direkt möglich.

9,6 kBd:

$$t = \frac{51,6 \text{ MHz}}{0,0096 \text{ MHz} \cdot 8} = 671,88$$

Zu wählen wäre 672, dieses ist ebenfalls nicht direkt möglich.

Die Fehler durch die Abweichung des Teilers vom idealen Wert liegen augenscheinlich deutlich unter 1 % und werden hier nicht mehr gesondert berechnet. Bei den Schrittraten 19,2 kBd und 9,6 kBd ist es nicht möglich, entsprechende Takte direkt mit einem Timer zu erzeugen. Es muss der Timer A1 mit den entsprechenden Timern, in diesem Fall A2 oder A3, kaskadiert werden. Hierzu ist ein gemeinsamer Teiler von 336 und 672 als Taktteiler für den Timer A1 zu wählen. Eine naheliegende Möglichkeit besteht darin, den größten gemeinsamen Teiler zu verwenden, der kleiner oder gleich 256 ist. Dieser beträgt 168, so dass für den Timer A2 oder A3 dann noch der Teiler 2 für die Geschwindigkeit 19,2 kBd und der Teiler 4 für 9,6 kBd zu konfigurieren ist. Bei der Geschwindigkeit 38,4 kBd kann gewählt werden, ob diese erreicht wird, indem direkt der Ausgang des Timer A1 gewählt und dieser Takt nicht weiter geteilt wird, oder ob der Haupttakt des Timersystems A verwendet und durch 168 geteilt wird.

Der Teiler des Timers A1 soll während der Laufzeit unverändert bleiben, während sich die Teiler für Timer A2 und Timer A3 dynamisch ändern, wobei ebenfalls bei Bedarf die Eingangstaktquelle zwischen Peripherietakt und Ausgang des Timer A1 umgeschaltet werden muss. Dieses

ist unumgänglich, da 13 eine Primzahl ist, und somit ein gemeinsamer Teiler für alle Geschwindigkeiten im Timer A1 nicht möglich ist. Bei den Schreiboperationen auf entsprechende Register muss dann noch beachtet werden, dass wie oben beschrieben der gewünschte Teiler minus eins verwendet wird.

Eine Komplikation, die bei Veränderungen des Timersystems auftreten kann, ist die, dass die Verbindung zwischen dem Rabbit 3000 und dem PC abbricht. Beim Programmstart sind folgende Einstellungen vorhanden, mit denen eine Verbindung zum PC mit 115,2 kBd aufrecht erhalten wird.

- Das Timersystem A wird mit halbem Peripherietakt betrieben.
- Die seriellen Schnittstellen befinden sich im x16-Modus.
- Als Teiler für Timer A4 (serieller Port A) ist 14 gewählt.

Es müssen also diese drei Einstellungen in einem möglichst kurzen Zeitrahmen auf die oben bestimmten Werte umgestellt werden. Erfolgt dies nur teilweise, hat dieses einen Abbruch der Verbindung zum PC zur Folge, da die Baudrate nicht mehr mit der Baudrate des PCs übereinstimmt. Das Debuggen der Anwendung ist dann nicht mehr über diese Verbindung möglich.

Eine weitere Funktion der Timer A1 bis A7 ist das Erzeugen von periodischen Interrupts. Auf diese Weise können regelmäßig bestimmte Aufgaben durchgeführt oder Bedingungen überprüft werden. Über das *Timer A Control and Status Register* (TACSR) können Interrupts für jeden dieser Timer ein- und ausgeschaltet werden. Ebenfalls lässt sich über dieses Register der Eingangstakt für das gesamte Timersystem abschalten. Beim Lesen dieses Registers wird darüber Auskunft gegeben, ob die einzelnen Timer A1 bis A7 seit dem letztem Lesen bei null angekommen waren.

Die Timer A8 bis A10 haben als Eingangstakt permanent den Haupttakt des Timersystems A. Der Timer A8 taktet mit seinem Ausgang die *Input Capture Unit*. Der Pulsweitenmodulator wird vom Timer A9 mit dem Taktsignal versorgt. Der *Quadrature Decoder* bezieht seinen Takt vom Timer A10. Auf die Funktionen dieser Gruppen wird in Abschnitt 3.2.9 eingegangen. Das Register *Timer A Control Register* dient zum einen zur Wahl der Eingangstakte für die Timer A2 bis A7, zum anderen lässt sich die Interruptpriorität auf einen Wert zwischen 0 und 3 einstellen.

Das Timersystem B

Das Timersystem B hat bei der Implementierung des Systems zur Anbindung zweier SICK LMS 200 keine Bedeutung, deshalb sei hier nur kurz die Funktion erklärt. Getaktet wird dieses System wahlweise vom halbierten Peripherietakt, vom Peripherietakt geteilt durch acht oder vom Ausgang des Timer A1. Hauptbestandteil dieses Timers ist ein 10-Bit Zähler, der mit gewählter Taktrate kontinuierlich hochzählt, bis es zum Überlauf kommt und der Zähler von vorne beginnt. Der Zählerstand wird kontinuierlich mit dem Wert zweier *Match Register* verglichen. Hat ein *Match Register* den gleichen Wert wie der Zähler, wird je nach *Match Register* ein Takt am Ausgang des Timer B1 oder B2 ausgelöst und es können folgende Aktionen veranlasst werden:

- Die parallelen Schnittstellen können zu gewünschter Zeit Daten ausgeben, auch wenn zu dieser Zeit gerade andere Programmteile ausgeführt werden.
- Es kann ein Interrupt ausgelöst werden, um beliebige weitere Aktionen auszuführen.

Nach Erreichen des Wertes im *Match Register* ist gegebenenfalls der nächste gewünschte Zeitpunkt ausgehend von dem aktuellen Zählerstand zu berechnen und in entsprechende Register einzutragen. Es muss dabei die gewünschte Verzögerungszeit gemessen in Takten des Zählers zum aktuellen Zählerstand addiert werden. Erst nach dem erneuten Programmieren können wieder Interrupts vom Timersystem B ausgehen. Ein autarkes Erzeugen eines Taktsignals wie beim Timersystem A ist somit nicht möglich.

Eine weitere Funktion des Timersystems B ist die des Zeitgebers. Das Zählerregister ist jederzeit auslesbar, und es können so in verschiedenen Programmteilen auf einfache Weise Zeitpunkte gespeichert werden. Hierbei muss noch die Zählgeschwindigkeit bekannt sein. Diese Funktionalität kann jedoch auch von der Echtzeituhr bereitgestellt werden, die jedoch komplizierter zu handhaben ist. Beschrieben wird die Echtzeituhr in Abschnitt 3.2.9. Die 10-Bit breiten Werte sind jeweils auf zwei Register verteilt, wobei die ersten sechs Bit im MSB hierbei keinerlei Relevanz haben.

3.2.8 Serielle Schnittstellen

Nachfolgend geht es um die seriellen Schnittstellen. Es wird ausführlich auf die Aspekte eingegangen, die für die asynchrone Datenübertragung von Bedeutung sind. Entsprechende Kenntnisse über die Funktionsweise der seriellen Schnittstellen sind erforderlich, um die Anbindung der SICK-Lasermesssysteme an das System zu verstehen.

Der Rabbit-3000-Prozessor besitzt sechs serielle Schnittstellen. Diese werden mit *Serial A* bis *Serial F* bezeichnet. Es folgt zunächst ein Überblick über die Funktionen:

- Alle sechs Schnittstellen unterstützen die asynchrone serielle Datenübertragung.
- Bei Bedarf kann die asynchrone Übertragung auch über eine Infrarotschnittstelle erfolgen (IrDA)
- Die Schnittstellen A bis D unterstützen zusätzlich die synchrone Datenübertragung.
- Die Schnittstellen E und F haben einen vier Byte großen FIFO Datenpuffer.
- Der Netzwerkstandard HDLC[8] wird von den Schnittstellen E und F unterstützt.
- Die Ausgangspins der Ports A und B können im Betrieb geändert werden (Umschalten zwischen verschiedenen Kommunikationspartnern möglich).
- Der Rabbit-3000-Prozessor kann direkt von dem seriellen Port A booten.
- Es kann zwischen sieben Bit und acht Bit pro Zeichen umgeschaltet werden und ein zusätzliches Bit angehängt werden.

Die maximale Geschwindigkeit der asynchronen Übertragung entspricht dem Peripherietakt geteilt durch 8. Bei einem Peripherietakt von 51,6 MHz sind dies 6,45 MBd. Wie bereits im vorherigen Abschnitt erwähnt, lässt sich die Geschwindigkeit durch ganzzahlige Teiler verringern. Bei der synchronen seriellen Datenübertragung an den Ports A bis D sind deutlich höhere Datenraten möglich. Hierbei wird zusätzlich zu dem Datensignal das Taktsignal übertragen. Es ist wählbar, ob dieses Taktsignal vom Rabbit 3000 oder vom Kommunikationspartner bereitgestellt wird. Die Geschwindigkeit im synchronen Modus ist maximal halb so groß wie der

[8] High-Level Data Link Control

Peripherietakt. Die Beschränkung der tatsächlichen sinnvoll realisierbaren Datenrate stellt hierbei eher die Geschwindigkeit der Weiterverarbeitung der empfangenen Daten beziehungsweise das Bereitstellen der zu sendenden Daten dar.

Es sollen nun die Register der seriellen Schnittstellen vorgestellt werden. Dabei wird nur auf die Funktionalität eingegangen, die diese bei dem asynchronen Modus haben. Jede der sechs Schnittstellen besitzt sechs Register zur Konfiguration, Überprüfung des Status und zur Datenübertragung. Das „x“ in der Benennung der Register steht für den jeweiligen Buchstaben A bis F.

Das Serial Port Data Register

Für den Datentransfer entscheidend ist das *Serial Port x Data Register*. Beim Lesen dieses Registers wird der Inhalt des Empfangspuffers ausgegeben. In Assembler ist dies über folgenden Befehl möglich (hier für den Port E):

```
IOI LD A,(SEDR)
```

Dieser Befehl lädt das empfangene Byte in den Akkumulator `A`. Von dort aus kann das Byte weiterverarbeitet und gespeichert werden. Das Senden erfolgt ebenfalls über das *Data Register*. Es wird hierbei auf das gleiche Register schreibend zugegriffen.

```
IOI LD (SEDR),A
```

Hier stellt das Register `A` die Quelldaten bereit und das *Data Register* des seriellen Port E ist das Ziel des Transfers. Nach dieser Schreiboperation wird automatisch der Datentransfer gestartet. Zuvor muss in den Akkumulator `A` das gewünschte Datenbyte geladen werden.

Address Register und Long Stop Register

Soll an das Datenbyte noch ein neuntes Bit angehängt werden, kommen das *Address Register* (SxAR) und des *Long Stop Register* (SxLR) zum Einsatz. Beim Verschicken der Daten wird der Schreibzugriff nicht auf das *Data Register* ausgeführt, sondern auf eines der beiden genannten. Beim Schreiben auf das *Address Register* wird dem Datenbyte ein 0-Bit angehängt, und das *Long Stop Register* bewirkt das Anhängen eines 1-Bit. Dieses Anhängen von Bit kann in verschiedenen Fällen sinnvoll sein:

- Datenübertragung mit zwei Stoppbit
  Wird zum Senden von Bytes ständig auf das *Long Stop Register* geschrieben, so kommt zu dem immer vorhandenen Stoppbit noch ein weiteres, ebenfalls gesetztes Bit hinzu. Somit ergibt sich ein doppelt so langes Stoppbit. Ein langes Stoppbit bietet Vorteile bei der Synchronisation der Übertragung und erleichtert vor allem das Einsynchronisieren in eine laufende Übertragung, daher werden bei einigen seriell arbeitenden Geräten zwei Stoppbit empfohlen.
- Senden eines Paritätsbit
  Sollen bei der Übertragung Einbitfehler erkannt werden können, kann an das zu sendende Byte ein Paritätsbit angehängt werden. Es muss zunächst die Parität über das Byte berechnet werden, dann wird abhängig von der Parität das Register zum Schreiben gewählt, so dass die Gesamtparität der gewünschten entspricht.

- Markieren von Bytes als Adressen bei mehreren Kommunikationspartnern (RS-485)
  Kommunizieren mehr als zwei Geräte an einem Bus, so muss mit verschiedenen Adressen gearbeitet werden. Immer, wenn eine Adresse verschickt werden soll, ist entsprechendes Byte in das *Adress Register* zu schreiben. Durch die Verwendung von Adressen können Daten einem bestimmten Gerät zugeordnet werden.

Serial Port Status Register

Das *Serial Port x Status Register* (SxSR) gibt Auskunft über den aktuellen Zustand der seriellen Schnittstelle. Folgende Informationen werden darüber bereitgehalten, ob...

- ein empfangenes Byte gelesen werden kann.
  Wird das Byte aus dem *Data Register* gelesen und sind keine weiteren komplett empfangenen Bytes mehr im FIFO (nur Port E und F), wird entsprechendes Bit im *Status Register* zurückgesetzt.
- das empfangene Byte ein angehängtes 0-Bit hatte.
  Mit der Information kann eine Paritätsprüfung gesteuert werden oder eine Auswertung des Bytes als Adresse veranlasst werden. Ein angehängtes 1-Bit kann nicht extra angezeigt werden, da es als Stoppbit erkannt wird. Bei der Paritätsprüfung muss immer dann davon ausgegangen werden, dass ein 1-Bit mitgesendet wurde, wenn kein 0-Bit angehängt ist.
- es einen Überlauf gab.
  Wenn seit dem letzten Lesen eines empfangenen Byte ein Byte verworfen wurde, weil ein empfangenes Byte nicht rechtzeitig gelesen wurde, wird entsprechendes Bit gesetzt. Dieses Bit wird mit dem Lesen des jeweiligem *Data Register* zurückgesetzt.
- der Sendepuffer voll ist.
  Ist entsprechendes Bit gesetzt, muss mit dem Schreiben weiterer Daten auf das *Data Register* gewartet werden.
- gerade Daten gesendet werden.
  Ist entsprechendes Bit nicht gesetzt, befindet sich der Sender im Leerlauf.

Die Auswertung dieses Registers ist vor allem in der ISR der seriellen Schnittstelle wichtig. Diese wird an späterer Stelle innerhalb dieses Abschnitts beschrieben.

Serial Port Control Register und Serial Port Extended Register

Über das *Serial Port Control Register* (SxCR) und das *Serial Port Extended Register* (SxER) lassen sich die seriellen Schnittstellen konfigurieren. Das *Serial Port Control Register* bietet hierbei folgende Einstellmöglichkeiten:

- Wahl des Betriebsmodus
  Es kann zwischen asynchroner Datenübertragung und synchroner Datenübertragung (bei Port E und F stattdessen HDLC) gewählt werden. Bei der asynchronen Übertragung muss zusätzlich zwischen sieben oder acht Bit pro Zeichen gewählt werden und bei der synchronen Datenübertragung muss angegeben werden, ob als Taktquelle eine externe oder die interne verwendet werden soll.

- Aktivieren oder Deaktivieren des Empfängereingangs
  Wenn gerade kein Empfang gewünscht ist, zum Beispiel während der Konfiguration, kann dieses durch Schreiben der entsprechenden Bit erreicht werden.
- Einstellen der Interruptpriorität oder Deaktivierung der Interrupts
  Die Interrupts der seriellen Schnittstelle können entweder ganz abgeschaltet werden oder es kann eine Priorität zwischen eins und drei zugewiesen werden.
- Umschalten der Ein- und Ausgänge (nur bei Port A und B)
  Die Umschaltung kann sinnvoll sein, wenn im Betrieb zu verschiedenen Zeiten mit verschiedenen Geräten kommuniziert werden soll.

Über das *Serial Port Extended Register* lassen sich folgende Konfigurationen vornehmen:

- Kodierung
  Es kann gewählt werden, ob die Daten über eine kabelgebundene Übertragungsstrecke, wie etwa bei RS-232 oder RS-422, oder über einen Infrarotübertrager (IrDA) übertragen werden.
- Überabtastungsmodus
  Die seriellen Schnittstellen werden für die asynchrone Datenübertragung auf den x16 oder den x8 Überabtastungsmodus eingestellt. Die größere Störungstoleranz besitzt der x16-Modus, während beim x8-Modus höhere Baudraten möglich sind, da der Eingangstakt durch 8 und nicht durch 16 geteilt wird. Auch erlaubt der x8-Modus eine feinere Abstufung der einstellbaren Taktraten. Die Überabtastung ist in Abschnitt 2.3 genauer erläutert.
- Warten auf neuntes Bit
  Hier wird konfiguriert, ob der Empfänger noch auf das angehängte Bit warten soll und dieses auswertet oder ob sofort nach einem komplett empfangenen Byte die Daten weitergegeben werden. Ein möglicherweise angehängtes Paritätsbit geht dabei verloren, jedoch steht das Byte früher zur Verfügung. Der Zeitvorteil entspricht einem Schritt bei der seriellen Datenübertragung.

Die Sonderstellung der seriellen Schnittstelle A

Die serielle Schnittstelle A spielt beim Entwickeln und Testen von Programmen eine entscheidende Rolle. Über diese Schnittstelle erfolgt die Verbindung zum Programmieren des Rabbit Mikroprozessors. Das System kann direkt über die serielle Schnittstelle gebootet werden. Nach dem Bootvorgang wird das Programm übertragen und während der Laufzeit des Programms können Debug-Informationen ausgegeben werden, die dann in der Entwicklungsumgebung dargestellt werden. Dies hat zur Folge, dass die serielle Schnittstelle A sowie das Timersystem A schon zu Programmstart konfiguriert sind. Soll im späteren Betrieb die Schnittstelle anderweitig verwendet werden, kann dies durch entsprechendes Umkonfigurieren erreicht werden.

Aktivieren der Ausgangspins

Folgende Konfigurationen sind zusätzlich notwendig, damit über die seriellen Schnittstellen nicht nur Daten empfangen, sondern auch gesendet werden können. Die Ausgänge des Rabbit-3000-Prozessors sind überbelegt, das heißt, sie können mehrere verschiedene Funktionen ausführen. So teilen sich einige parallele Schnittstellen ihre Ausgangspins mit den seriellen Schnittstellen. Solange auf diese nur lesend zugegriffen wird, stört diese Doppelbelegung nicht weiter und es kann

so der Verantwortung des Nutzers überlassen werden, nur sinnvolle Leseoperationen durchzuführen. Sobald jedoch Schreibzugriffe erfolgen sollen, muss die Kontrolle über den Ausgangspin dem gewünschten Gerät übergeben werden.

In der Grundeinstellung befinden sich die meisten Pins im Lesemodus und sind den parallelen Schnittstellen zugeordnet. Dieser Modus lässt sich über Register der parallelen Schnittstelle umschalten. Für die Ausgänge der seriellen Schnittstellen E und F muss auf die Register des Parallelports G geschrieben werden. Das *Port G Function Register* (PGFR) ermöglicht die Zuweisung der entsprechenden Pins an die Ausgänge der seriellen Schnittstellen. Zusätzlich muss ebenfalls bei den Registern des Parallelports die Datenrichtung bei entsprechenden Pins auf Ausgang geschaltet werden. Dieses erfolgt im Fall der seriellen Schnittstellen E und F über das *Port G Data Direction Register* (PGDDR). Über das *Port G Drive Control Register* (PGDCR) wird noch das korrekte Verhalten des Ausgangstreibers eingestellt. Eine hier fehlerhafte Einstellung hätte zur Folge, dass am Ausgang keine *high*- und *low*-Pegel anliegen würden, sondern je nach logischem Zustand der Ausgang hochohmig oder mit Masse verbunden wäre[9]. Der Parallelport C teilt die Pins mit den seriellen Schnittstellen A bis D. Beim Umlegen der Ein- und Ausgänge der seriellen Schnittstellen A und B müssen in den Registern des Parallelport D entsprechende Einstellungen vorgenommen werden.

Die Verwendung der seriellen Schnittstellen zur Ansteuerung von SICK-Lasermesssystemen

Nunmehr werden die Einstellungen erläutert, die an den seriellen Schnittstellen vorgenommen werden müssen, um eine korrekt funktionierende Verbindung zu den SICK-Lasermesssystemen aufzubauen. Zunächst müssen die Schnittstellen ausgewählt werden, über die die beiden Geräte angebunden werden. Die SICK LMS 200 kommunizieren über eine asynchrone serielle Verbindung, daher ist im Prinzip jede der seriellen Schnittstellen nutzbar. Da die Schnittstellen E und F jeweils 4 Byte große Sende- und Empfangspuffer haben, sind diese beiden Ports besonders geeignet. Für den Fall, dass Daten nicht rechtzeitig abgeholt werden, ist die Gefahr eines Datenverlusts deutlich geringer als bei den anderen Ports ohne Empfangspuffer. Die Daten dürfen auch bei einem vorhandenen Empfangspuffer nicht über einen längeren Zeitraum schneller eintreffen als sie weiterverarbeitet werden können, jedoch kann für den Fall, dass nur kurzzeitig keine Rechenzeit zur Verfügung steht, ein Datenverlust verhindert werden.

In den entsprechenden Registern der Ports E und F wird demnach der asynchrone Modus mit acht Bit pro Zeichen eingestellt. Der Empfängereingang wird aktiviert und die Interrupts werden eingeschaltet, da die zeitkritische Weiterverarbeitung interruptgesteuert erfolgen soll. Die Einstellung der Priorität wird auf zwei gesetzt. Diese Einstellung wird in Abschnitt 5.3 begründet.

Bei der Wahl der Kodierung wird die Standardkodierung für die kabelgebundene Übertragung gewählt. Ein neuntes Bit braucht nicht abgewartet zu werden, da kein Paritätsbit verschickt wird. Wie in Abschnitt 3.2.7 erläutert, ist der x8 Überabtastungsmodus zu wählen.

Interruptverhalten der seriellen Schnittstellen

Im Folgenden wird das Interruptverhalten der seriellen Schnittstellen beschrieben. Zunächst einmal muss entschieden werden, ob der Betrieb der Schnittstellen interruptgesteuert erfolgen

[9] Diese Schaltung wird beim Betrieb an einem Bussystem wir beispielsweise I^2C verwendet

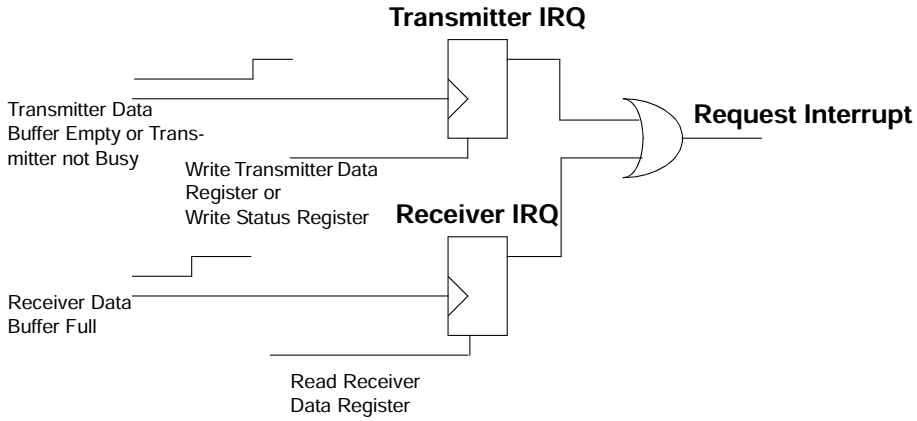


Abbildung 3.10: Erzeugung von Interrupts der seriellen Schnittstelle, Quelle: [R3000UM, S. 179]

soll oder nicht. Ist der Empfang einer Serie von Daten gefordert, bei der kein Byte verloren gehen darf, so wird in der Regel interruptgesteuert gearbeitet. Wenn jedoch stets nur auf ein wiederholt gesendetes Statusbyte gewartet wird, welches zum Beispiel Messdaten eines Analog-digital-Wandlers enthält, so kann der Betrieb auch ohne Interruptsteuerung erfolgen. Es werden dann die Daten bei Bedarf gelesen und bewusst der Verlust einiger Messdaten in Kauf genommen.

Wird interruptgesteuert gearbeitet, werden folgende Ereignisse durch Interrupts angezeigt:

- Es sind empfangene Daten abrufbar.
- Der Sendepuffer ist leergelaufen.
- Der Sender ist in den Leerlauf geraten.

Diese drei verschiedenen Ereignisse führen zum Auslösen des gleichen Interrupts, weshalb immer die gleiche ISR ausgeführt wird. Aus diesem Grund muss innerhalb der ISR die Ursache des Interrupts abgeklärt und entsprechend reagiert werden. Jede der sechs seriellen Schnittstellen besitzt jedoch eine eigene ISR, hier muss nicht weiter die Quelle des Interrupts geklärt werden.

Die Unterscheidung der verschiedenen Interruptursachen erfolgt durch Auswertung des *Serial Port Status Register*. Es wird zunächst geprüft, ob entsprechendes Bit innerhalb dieses Registers anzeigt, dass ein empfangenes Byte gelesen werden kann. Anderenfalls liegt die Ursache des Interrupts darin, dass das *Data Register* des Senders leergelaufen ist oder dass die Übertragung beendet ist und der Sender sich fortan im Leerlauf befindet. Grafisch dargestellt ist dieses Verhalten in Abbildung 3.10. Hier entscheidet der Pegel der Leitung mit der Bezeichnung „Request Interrupt“ darüber, ob die entsprechende serielle Schnittstelle eine Interruptanforderung an den Prozessorkern sendet oder nicht. Diese Leitung wird über eine OR-Verknüpfung der Interruptanforderungen von Sender (*Transmitter IRQ*) und Empfänger (*Receiver IRQ*) beschaltet. Sowohl Sender als auch Empfänger verwenden bei der Entscheidung über die Interruptanforderung ein vorderflankengesteuertes Flipflop. Die Funktionsweise und die verschiedenen Typen von Flipflops sind in [TG01] beschrieben. An den linken Seiten der Flipflops sind die Zuleitungen der Flipflops dargestellt und die Auslösebedingungen vermerkt. Tritt die Bedingung ein, wird die Interruptanforderung ausgelöst. Des Weiteren sind auch die Bedingungen dargestellt, die zum Zurücksetzen der Interruptanforderung auftreten müssen. Diese finden sich an den Leitungen, die in der Grafik an der unteren Seite der Flipflops angeschlossen sind.

Bei der Interruptanforderung des Empfängers erfolgt das Zurücksetzen durch Lesen des *Serial Port Data Register*. Dieses Register liefert beim Lesen die empfangenen Daten. Innerhalb der ISR können diese nun weiterverarbeitet und gespeichert werden. Es ist somit möglich, die Daten kurz nach dem Eintreffen zu verarbeiten, um den Empfangspuffer möglichst schnell wieder zu leeren und den Empfang weiterer Daten zu ermöglichen. Für die seriellen Schnittstellen E und F ergibt sich die Besonderheit, dass diese einen Empfangspuffer besitzen. Sollte dieser Puffer gefüllt sein und es wird das *Serial Port Data Register* gelesen, so erfolgt kein Zurücksetzen der Interruptanforderung.

Beim Sender kann die Zurücksetzung der Interruptanforderung auf zwei Arten erfolgen:

- Schreiben eines Byte in das *Data Register*
- Schreiben eines Byte in das *Status Register*

Für den Fall, dass gerade eine Übertragung stattfindet, soll beim Leerlaufen des Sendepuffers das Schreiben des nächsten Byte auf das *Data Register* erfolgen. Auf diese Weise ermöglicht es die Interruptsteuerung, die Sendeeinheit immer rechtzeitig mit Daten zu versorgen, so dass diese nicht in den Leerlauf gerät. Ist das Senden eines weiteren Byte nicht gewünscht, so kann die Interruptanforderung auch durch Schreiben eines Bytes auf das *Status Register* zurückgesetzt werden. Der Inhalt dieses Bytes spielt keine Rolle und wird ignoriert. Dennoch ist diese Operation wichtig, da in der ISR dafür gesorgt werden muss, dass die Interruptanforderung abgearbeitet wird. Dieses ist in Abschnitt 3.2.5 genauer erläutert.

Wird ein Interrupt aufgrund des leergelaufenen Sendepuffers durch Schreiben auf das *Status Register* verarbeitet, folgt in Kürze ein weiteres Interrupt, da die Sendeeinheit nun in den Leerlauf gerät. Dieses wird ebenfalls durch Schreiben auf das *Status Register* zurückgesetzt. Sinn dieses doppelten Interrupts ist, dass für einige Anwendungen der genaue Zeitpunkt bekannt sein muss, zu dem eine Übertragung beendet ist. Bei Halbduplex-Verbindungen wie RS-485 muss nach abgeschlossener Übertragung der Transceiver vom Sende- auf den Empfangsmodus umgeschaltet werden.

Beim Interrupt des Senders ist noch zu beachten, dass es zur Interruptanforderung nur beim Auftreten eines der beiden Ereignisse kommt. Nach zweifacher Abarbeitung der ISR können diese Ereignisse weiter bestehen, ohne dass neue Interrupts ausgelöst werden. Ansonsten würden ständig Interrupts erzeugt, solange nicht irgendwelche Daten gesendet werden. Dieses Verhalten wird durch die vorderflankengesteuerten Flipflops erreicht.

Vorstellung einer einfachen Interrupt-Service-Routine

Im Folgenden soll nun das Gerüst für eine ISR vorgestellt werden. Diese wird noch keinerlei praktische Funktion besitzen, sondern die empfangenen Daten verwerfen. Es wird davon ausgegangen, dass das Senden von Daten außerhalb der ISR durch entsprechende Schreiboperationen auf das *Data Register* stattfindet. Die ISR ist in Assembler geschrieben und verarbeitet die Interrupts der seriellen Schnittstelle E.

```
#asm
porte_isr::
   PUSH AF              ; Speichern der vorherigen Registerinhalte
   IOI LD A,(SESR)      ; Laden des Statusbyte
```

```
    CP 0x80              ; Prüfen, ob erstes Bit gesetzt ist
    JP P,porte_receive   ; wenn gesetzt, zu porte_receive springen

porte_transmit:
    IOI LD (SESR), A     ; Interruptanforderung löschen
                         ; (beliebiges Byte schreiben)
    IPRES                ; eigene Interruptpriorität zurücksetzen
    POP AF               ; vorherige Registerinhalte wiederherstellen
    RET                  ; ISR verlassen

porte_receive:
    IOI LD A,(SEDR)      ; empfangenes Byte aus dem Data Register lesen
    IPRES                ; eigene Interruptpriorität zurücksetzen
    POP AF               ; vorherige Registerinhalte wiederherstellen
    RET                  ; ISR verlassen
#endasm
```

Beim Auslösen eines Interrupts der seriellen Schnittstelle E wird fortan die ISR ausgeführt. Es wird ein Sprung auf die Adresse ausgeführt, die hier mit `porte_isr` benannt ist und zur Kompilierzeit durch die physikalische Adresse ersetzt wird. Im Programmcode muss dem System zuvor noch bekanntgegeben werden, dass es sich bei `porte_isr` um die ISR der seriellen Schnittstelle handelt.

Zunächst erfolgt die Sicherung der Prozessorregister, mit denen innerhalb der ISR gearbeitet wird. In dem gezeigten Beispiel wird nur das `AF`-Register mit dem Befehl `PUSH AF` auf dem Stack gespeichert.

Anschließend wird das Statusbyte des seriellen Port E in den Akkumulator `A` geladen. Dies erfolgt durch den Befehl `IOI LD A,(SESR)`. Das höchstwertige Bit des Statusbyte gibt an, ob im Empfangspuffer ein Byte zum Lesen vorhanden ist. Um dieses höchstwertige Bit auszuwerten, wird das gesamte Byte mit 0x80 verglichen (`CP 0x80`). Der Vergleich wird intern durch eine Subtraktion realisiert. War das höchstwertige Bit gesetzt, so hat das gesamte Byte einen Wert von mindestens hexadezimal 0x80 und das Ergebnis dieser Subtraktion ist nicht negativ. In dem *Flags Register* des Prozessors, das zusammen mit dem Akkumulator gesichert wurde, sind nach dem Vergleich entsprechende Bit gesetzt. In dem konditionalen Sprung `JP P, porte_receive` wird das *Flags Register* überprüft und im Endeffekt der Sprung durchgeführt, wenn das höchstwertige Bit gesetzt war.

Hier teilt sich der Ablauf der ISR in zwei Teile. Bei Ausführung des Sprungs wird bei dem Label `porte_receive` weitergearbeitet. Ansonsten wird der Code ausgeführt, der hinter dem Label `porte_transmit` platziert ist. Letzteres Label ist nur aus optischen Gründen eingefügt und hat keinerlei Funktionalität als Sprungadresse.

Wird der Sprung nicht ausgeführt, wird mit dem Befehl an der nächsten Speicheradresse weitergearbeitet. Es folgt in diesem Fall das Schreiben eines Byte auf das *Status Register*. Wie bereits ausführlich erläutert, erfolgt so das Zurücksetzen des Sender-Interrupts. Die darauf folgenden drei Anweisungen dienen zur Beendigung der ISR. Der Befehl `IPRES` setzt die Prioritätsstufe auf den vor der Ausführung dieser ISR gültigen Wert zurück. Dieses erfolgt zu einem Zeitpunkt, wo der zeitkritische Teil der ISR bereits ausgeführt ist. Es kann nun anderen Interruptanforderungen stattgegeben werden. Mit dem Befehl `POP AF` wird der ursprüngliche Inhalt des Akkumulators

und der Inhalt im *Flag Register* wiederhergestellt. Schließlich führt der Befehl `RET` zum Verlassen der ISR und zur Weiterausführung der zuvor unterbrochenen Codefolge.

Sollte zuvor der Sprung zu `porte_receive` ausgeführt worden sein, so wird als nächstes das empfangene Byte mit dem Befehl `IOI LD A,(SEDR)` in den Akkumulator geladen. An dieser Stelle würde eigentlich die Weiterverarbeitung des empfangenen Bytes folgen. In der vorgestellten ISR unterbleibt diese jedoch und das Byte wird durch Überschreiben des Akkumulators mit dem ursprünglichem Wert gelöscht.

3.2.9 Weitere Funktionseinheiten

Der folgende Abschnitt befasst sich mit den Ausstattungsmerkmalen der Rabbit-3000-CPU, deren Kenntnis nicht zwingend zum Betrieb zweier SICK-Lasermesssysteme notwendig ist. Es wird daher nur knapp auf die Funktionen und deren Benutzung eingegangen und der Einsatz dieser Techniken im Rahmen der Entwicklung des Systems dargestellt.

Watchdog Timer

Der *Watchdog Timer* ist eine Komponente, die Systemabstürze erkennen kann, und dann selbstständig einen Reset durchführt. Diese Erkennung funktioniert auf die Weise, dass in einem Programmteil, der möglichst regelmäßig durchlaufen werden sollte, ein Schreibzugriff auf das *Watchdog Timer Control Register* (WDTCR) erfolgt. Je nach geschriebenem Byte wird ein Timeout zwischen 0,25 und 2 Sekunden gesetzt. Wenn innerhalb dieser Zeitspanne kein weiterer Schreibzugriff auf das Register erfolgt, wird ein Reset ausgelöst. Technisch realisiert ist diese Funktion durch einen 17-Bit-Zähler, dessen Zähltakt vom *32,786 kHz Clock Input* vorgegeben wird. Bei Bedarf kann auch der Peripherietakt zum Zählen gewählt werden.

Kommt es aufgrund eines Programmfehlers dazu, dass das Programm sich in einer Endlosschleife verfängt, erfolgt nach kurzer Zeit ein Neustart des Systems. Ebenfalls denkbar wäre ein Absturz durch elektromagnetische Störungen, welcher ebenfalls abgefangen werden könnte. Für die korrekte Funktion des *Watchdog Timers* muss der Schreibzugriff auf das Register an einer günstigen Stelle erfolgen. Es müssen Konstellationen durchdacht werden, in denen dieser Programmteil selbst Teil einer Endlosschleife werden könnte. Ebenfalls müssen reguläre Programmabläufe analysiert werden, in denen es nicht innerhalb des gesetzten Timeouts zum Schreiben auf das Register kommen könnte.

Ein Einsatz des Watchdog Timers soll im Rahmen dieser Arbeit angestrebt werden. Unvorhersehbare Betriebssituationen können abgefangen werden und im Falle eines Absturzes ist nicht zwangsweise das Trennen der Stromversorgung notwendig.

Input Capture Unit

Mit den zwei *Input Capture Units* kann jeweils eine Zeitmessung zwischen zwei Ereignissen erfolgen. Hierzu wird eine Start- und eine Stoppbedingung konfiguriert. Eine solche Bedingung kann ein Pegelwechsel auf einem Eingangspin der Rabbit-3000-CPU sein, wobei die Art des Pegelwechsels (Vorder- oder Rückflanke) ebenfalls berücksichtigt werden kann. Der Pin kann für die

Start- und die Stoppbedingung relativ frei unter den Pins der parallelen Schnittstellen gewählt werden. Es kann die Zahl der Takte zwischen Start- und Stoppbedingung gemessen werden. Die Zählrate hängt vom Peripherietakt und der Konfiguration des Timers A8 ab. Gezählt wird auf einem 16-Bit Zähler, wobei der Zähler kontinuierlich laufen kann oder nur vom Auftreten der Start-Bedingung bis zum Auftreten der Stoppbedingung. In letzterer Konfiguration kann die Breite eines Taktpulses gemessen werden. Zum Beispiel lässt sich auf diese Weise die Baudrate einer seriellen Schnittstelle bestimmen, vorausgesetzt, es wird ein geeignetes Datenmuster übertragen. Eine Implementierung dieser Baudratenbestimmung wird nicht angestrebt, da der Verbindungsaufbau mit den SICK-Lasermesssystemen wie in 2.2.4 beschrieben erfolgen kann.

Es können wahlweise beim Auftreten der Start- und Stoppbedingung Interrupts ausgelöst werden, um die Auswertung zu starten. Wenn keine Zeitmessung durchgeführt wird, können bei Aktivierung der Interrupts die Eingangspins als zusätzliche externe Interrupteingänge genutzt werden.

Quadrature Decoder

Die zwei *Quadrature Decoder* sind Einheiten, die die Drehbewegung und Drehrichtung einer Achse bestimmen. Dies erfolgt durch optische Abtastung einer Scheibe, die in lichtundurchlässige und durchsichtige Streifen aufgeteilt ist. Erfolgt die Abtastung von zwei versetzt angebrachten Lichtquellen, kann je nach Abfolge der gemessenen Abtastwerte auf die Drehrichtung geschlossen werden. Die Zahl der Schritte wird in einem 8-Bit-Register gespeichert. Eine ausführliche Beschreibung dieses Messverfahrens findet sich in [Zha05].

Parallele Schnittstellen

Die parallelen Schnittstellen dienen zur Datenübertragung. Parallele Datenübertragung bedeutet, dass auf mehreren Datenleitungen ein Datenwort übertragen wird. Bei der Rabbit-3000-CPU haben die Schnittstellen eine Breite von acht Bit. Insgesamt sind sieben dieser Schnittstellen vorhanden, wobei die Ausgangspins mit anderen Komponenten geteilt sind und somit eventuell nicht alle nutzbar sind. Die Pins der parallelen Schnittstellen können als Eingang oder Ausgang konfiguriert werden. Dieses erfolgt über die jeweiligen Register der parallelen Schnittstellen.

PWM-Units

Die Rabbit-3000-CPU besitzt vier pulsbreitenmodulierte Ausgänge. Bei dieser Art der Ansteuerung wird der Ausgang in regelmäßigen Abständen zwischen *low*-Pegel und *high*-Pegel umgeschaltet. Das Verhältnis zwischen *low*- und *high*-Pegel kann dabei variiert werden, ebenso wie die absolute Zeitdauer des Durchlaufens beider Zustände.

Über solche Ausgänge kann die Leistung elektrischer Verbraucher (Glühlampen, LEDs, Motoren, Heizungen, etc.) geregelt werden. Zusätzlich werden noch leistungsfähige Transistoren zum Schalten benötigt. Da die Verbraucher nicht die ganze Zeit mit Strom versorgt werden, sinkt die Leistungsaufnahme und entsprechend auch die Leistungsabgabe. Ein weiterer Einsatzzweck ist die Steuerung von Modellbauservos, die ihre Sollposition über eine analoge Schaltung der Breite eines Pulses entnehmen.

Ein Einsatzzweck innerhalb der Entwicklung wäre die Möglichkeit, bei Bedarf die Helligkeit der Entfernungsanzeige zu steuern, da es aufgrund des Einsatzes von LEDs moderner Bauart zur Störung anderer Sensoren kommen könnte (zum Beispiel ein farblich verfälschtes Kamerabild).

Slave Port

Der *Slave Port* dient zum Zusammenschalten zweier Rabbit CPUs. Die Leitungen sind mit dem ersten Parallelport geteilt, so dass nur eine der beiden Funktionen zur Zeit nutzbar ist. Beim Zusammenschluss wird eine CPU als Master, die andere als Slave konfiguriert. Die Aufgaben für einen zweiten Prozessor können zum Beispiel in der Steuerung von Bewegungsabläufen liegen, wenn extrem schnelle Reaktionszeiten gefordert sind. Des Weiteren kann die Abarbeitung eines Kommunikationsprotokolls den Einsatz einer zweiten CPU erfordern.

External Chip Interface

Über die *External Chip Interface* genannte Einheit lassen sich bis zu 8 weitere Peripheriegeräte ansteuern. Angesprochen werden die externen Komponenten, indem den Operationen, die auf Speicher zugreifen, der Präfix `IOE` vorangestellt wird. Statt eines normalen Speicherzugriffs erfolgt nun die Ansteuerung eines der acht externen Geräte, abhängig von den drei höchstwertigen Bit der Speicheradresse des Befehls.

In der aktuellen Version der Rabbit-3000-CPU kann der Zugriff wahlweise über einen speziellen Bus oder den normalen Speicherbus erfolgen. Die Zugriffe über den Speicherbus unterscheiden sich von normalen Speicherzugriffen durch erweiterte Einstellmöglichkeiten des Timings. Auf diese Art ist der Ethernet-Controller auf dem Rabbit Powercore 3800 angebunden. Da hierzu entsprechende Bibliotheken zur Ansteuerung bereitgestellt werden, braucht der Chip nicht auf Assembler-Ebene angesprochen zu werden. Bei der Benutzung des speziellen Bus ist zu beachten, dass dieser mit 6-Bit Adressen arbeitet, so dass der adressierbare Bereich nur bei 64 Byte pro Gerät liegt.

Real Time Clock

Die Echtzeituhr ist als 48-Bit-Zähler ausgelegt, aufgeteilt in sechs Register mit jeweils acht Bit. Die Zählrate ist 32,768 kHz, so dass ein Überlauf theoretisch nach 272 Jahren auftritt. Da das höchstwertige Bit ignoriert wird, halbiert sich diese Zeitspanne. Auf die sechs Register kann einzeln zum Auslesen der aktuellen Zeit zugegriffen werden. Vor dem Lesen muss durch einen Schreibvorgang auf eines der Register veranlasst werden, dass der Inhalt der sechs Register in einem Puffer zwischengespeichert wird. Der geschriebene Wert wird dabei ignoriert. Beim Lesevorgang wird immer der Wert des Puffers ausgegeben. Ansonsten könnte es zu Ungenauigkeiten kommen, da zwischen den Auslesevorgängen die Zeit weiterläuft. Dies wäre besonders kritisch, wenn es zu Zählerüberläufen kommt. Bei einem später folgenden Lesevorgang muss erneut durch einen Schreibzugriff der Puffer mit dem aktuellen Wert geladen werden.

Eine Konvention schreibt vor, dass der Nullstand des Zählers dem Zeitpunkt 0:00 Uhr am 1. Januar 1980 entspricht. Das Einstellen der Zeit erfolgt nicht durch direkte Schreibzugriffe, sondern

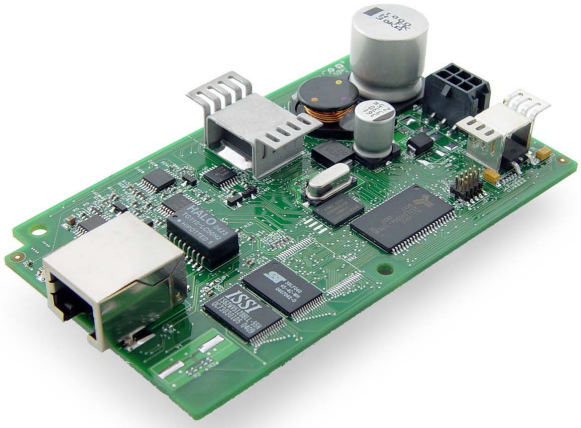


Abbildung 3.11: Foto des Rabbit Powercore 3800

durch gleichzeitiges Inkrementieren beliebig vieler der sechs Register. Sofern eine Pufferbatterie angeschlossen ist, funktioniert die Uhr auch bei nicht angeschlossener Stromversorgung. Ein möglicher Einsatzzweck im Rahmen dieser Arbeit bestünde darin, die Messdatentelegramme mit einem Zeitstempel zu versehen, um effektiv kontrollieren zu können, wie aktuell ein Telegramm ist.

Periodic Interrupt

Diese Funktionseinheit dient zum regelmäßigen Auslösen eines Interrupts. Der Interrupt tritt alle 488 μs auf, sofern die Einheit aktiviert ist. Es können interruptgesteuert regelmäßig anfallende Aufgaben ausgeführt werden. Eine Implementierung dieser Funktion wird nicht angestrebt.

3.3 Rabbit Powercore 3800

Im folgenden Teil dieser Arbeit soll der Rabbit Powercore 3800 detailliert vorgestellt werden. Die verschiedenen Ausstattungsmerkmale werden betrachtet und deren Funktion wird im Hinblick auf die Entwicklung des Systems zur Anbindung zweier SICK LMS 200 erläutert. Beim Rabbit Powercore 3800 handelt es sich um einen kompakten Einplatinen-Computer basierend auf einer Rabbit-3000-CPU mit 51,6 MHz. In 3.11 ist ein Foto des Rabbit Powercore 3800 zu sehen. Die Abmessungen der Platine betragen 60 mm x 102 mm x 28 mm.

3.3.1 Speicherausstattung

Der Powercore 3800 ist mit 512 kByte Flash-Speicher zur dauerhaften Speicherung des auszuführenden Programms ausgestattet. Als Arbeitsspeicher sind 1 MByte SRAM[10] vorgesehen. Dieser

[10] statischer Speicher

ist aufgeteilt in zwei Blöcke von je 512 kByte, wobei einer dieser Blöcke über eine Batterie auch ohne Betriebsspannung die Daten hält. Zur persistenten Speicherung von Daten ist ein *serial flash*-Datenspeicher in einer Größe von ebenfalls einem MByte vorhanden. Ein möglicher Einsatzzweck ist das Anlegen eines Dateisystems und das Abspeichern von Webseiten beim Aufbau eines kleinen Webservers.

3.3.2 Netzwerkschnittstelle

Zur Anbindung des Systems an ein lokales Netzwerk verfügt der Rabbit Powercore 3800 über einen Netzwerkcontroller der Firma Realtek. Dieser ist kompatibel zu 10 MBit Netzwerken. Auf der Platine ist eine RJ-45-Buchse zum Anschluss eines Netzwerkkabels verbaut. Über Leuchtdioden wird eine erfolgreiche Verbindung und eine laufende Datenübertragung angezeigt.

3.3.3 Spannungsversorgung

Der Rabbit Powercore 3800 ist mit Spannungswandlern für die Versorgung aller vorhandenen Komponenten ausgestattet. Zusätzlich werden Spannungen von 3,45 V und 5 V an dem 50-poligen Motherboardanschluss bereitgestellt. Somit können externe Verbraucher mit einer Stromaufnahme bis zu 1,85 Ampere (5 Volt) bzw. 0,55 Ampere (3,45 Volt) ohne zusätzliches Netzteil betrieben werden. Die Leistungsaufnahme beträgt laut [PCUM] ca. 2,7 Watt zuzüglich Stromverbrauch externer Verbraucher und Wandlerverluste. Der Rabbit Powercore 3800 selbst kann auf drei verschiedene Arten mit Strom versorgt werden:

- geregelte Gleichspannung 5 Volt
- ungeregelte Gleichspannung 8-43 Volt
- Wechselspannung im Bereich von 12-36 Volt

Es wird ein Spezialtransformator mitgeliefert, über den der Rabbit Powercore 3800 mit Wechselspannung versorgt werden kann. Der Grund, warum überhaupt dem Modul Wechselspannung zugeführt wird und diese nicht im Transformator gleichgerichtet wird, ist folgender: Der Nulldurchgang der Wechselspannung kann erkannt werden und es können dazu synchron Triacs geschaltet werden. Dabei handelt es sich um Leistungsschalter für Wechselströme, die zur Regulierung der Leistung eingesetzt werden.

Der Serviceroboter des Arbeitsbereich TAMS verfügt über eine 48 Volt Spannungsversorgung über Bleiakkumulatoren. Diese Spannung kann nicht direkt verwendet werden, da die 43 Volt überschritten werden würden. Der Steuerrechner des Roboters besitzt ein Netzteil, das die 48 Volt unter anderem zu 5 Volt und 12 Volt konvertiert, damit Standard-PC-Komponenten versorgt werden können. Die geregelte 5 Volt-Spannung könnte an den Eingang der geregelten Gleichspannung gelegt werden. Alternativ kann der 12 Volt-Ausgang am Eingang für die ungeregelte Gleichspannung angeschlossen werden. Da über die Qualität des Netzteiles keine Informationen vorliegen, erscheint letzteres als die sicherere Lösung. Von der Effizienz her ist dieses Vorgehen sicherlich nicht optimal, da die Spannung einmal zusätzlich gewandelt werden muss, was zu entsprechenden Verlusten führt. Jedoch liegt die Leistungsaufnahme im Vergleich zu den anderen auf dem Serviceroboter installierten Komponenten in einem Bereich, der zu vernachlässigen ist.

Eine Knopfzelle der Bauart CR 2032 sorgt im ausgeschalteten Zustand für die Spannungsversorgung der Echtzeituhr und eines Teiles des Arbeitsspeichers. Die Lebensdauer der Batterie beträgt ungefähr vier Jahre, wobei im Betrieb der Batterie kein Strom entnommen wird.

3.3.4 50-Pin Motherboard Connector

Der Rabbit Powercore 3800 ist darauf ausgelegt, über einen 50-poligen Konnektor auf ein vom Nutzer zu entwerfendes Motherboard gesteckt zu werden. Über die Leitungen erfolgt die Datenübertragung sowie die Spannungsversorgung, wobei zum einen die Betriebsspannung dem Powercore 3800 zugeführt werden kann und zum anderen die konvertierten Spannungen von 3,45 Volt und 5 Volt dem Motherboard zur Verfügung gestellt werden. Die Datenleitungen des Prozessors werden direkt an das Motherboard weitergeleitet. Im einzelnen werden folgende Anschlüsse bereitgestellt:

- externer Datenbus(Adress-, Daten- und Chip-Select-Leitungen)
- fünf serielle Anschlüsse
- ein paralleler Ausgang(die übrigen sind aufgrund Überbelegung nicht nutzbar)
- vier pulsbreitenmodulierte Ausgänge
- Anschlüsse des *Quadrature Decoder*
- Eingänge für externe Interrupts

Nicht nach außen zugänglich ist hingegen der serielle Anschluss A. Dieser wird ausschließlich für die Programmierung und das Debuggen verwendet und ist auf dem Rabbit Powercore 3800 intern verbunden. Ebenfalls wird der Speicherbus mit Daten- und Adressleitungen nicht auf den Konnektor gelegt, da aufgrund der hohen Datenrate Störungen in der Datenübertragung auftreten könnten. Dieser Bus ist intern mit den Speicherchips und dem Netzwerkcontroller verbunden.

Prototyping-Board

Das Prototyping-Board dient als Prototyp für das zu entwickelnde Motherboard. Es handelt sich um eine zusätzliche Platine, die hier auf keiner Abbildung gezeigt ist, da sie im Rahmen dieser Arbeit nicht verwendet wird. Mit diesem Board können direkt einige Demonstrationsprogramme ausgeführt werden. Viele übliche Komponenten sind bereits auf dem Board vorhanden:

- RS-232-Treiber
- Analog-digital- und Digital-analog-Wandler
- Taster und LEDs
- Triacs und Lampen
- Rabbit Net Port (Verbindung zwischen einem Master und einem Slave)

Auf freiem Platinenplatz können noch zusätzliche Komponenten installiert werden, deren Stromversorgung über das Prototyping-Board erfolgen kann.

3.3.5 Programmierport

Auf dem Rabbit Powercore 3800 befindet sich ein 10-poliger Konnektor zum Programmierkabel. Der Konnektor ist zum einen mit den Datenleitungen der seriellen Schnittstelle A verschaltet, zum anderen kann über das Programmierkabel ein Reset des Prozessors ausgelöst werden. Des Weiteren sind spezielle Pins des Prozessors mit diesem Konnektor verbunden. Ist ein Programmierkabel angeschlossen, so bootet das System nicht vom Flash-Speicher, sondern direkt über die serielle Schnittstelle. Nach dem Übertragen des kompilierten Programms ist über die serielle Verbindung das Debuggen möglich. Ist eine Debug-Verbindung gewünscht und soll dabei keine erneute Programmierung erfolgen, kann das Programmierkabel auf eine andere Art angesteckt werden, so dass entsprechende Pins des Prozessors nicht gebrückt werden.

3.3.6 Ramp Generator

Die Funktion des *Ramp Generators* besteht darin, eine zeitlich linear ansteigende Spannung von -0,05 Volt bis 3,1 Volt zu erzeugen. Die Zeit, in der die Spannung von 0 Volt auf 3,1 Volt steigt, ist bekannt und beträgt 1,9 ms. Mit Hilfe von Komparatorschaltungen, die zum Teil schon auf dem Rabbit Powercore 3800 verbaut sind, ist es möglich, diese Spannung mit weiteren Spannungen zu vergleichen. Anhand des Zeitpunktes, an dem die Spannung des *Ramp Generators* die zu messende Spannung übersteigt, kann deren Höhe abgeschätzt werden. Innerhalb der Rabbit-3000-CPU wird dieser Vergleich über die *Input Capture Unit* aufgenommen, wobei der Startzeitpunkt das Überschreiten von 0 Volt darstellt und der Stoppzeitpunkt das Überschreiten der zu messenden Spannung. Der Zählerstand verhält sich proportional zur Spannung. Zur Kalibrierung ist auf dem Rabbit Powercore 3800 eine präzise 2,5 Volt Spannungsquelle vorhanden. Über entsprechende Schaltungen sind über reine Spannungsmessungen hinaus auch Temperatur- und Widerstandsmessungen möglich.

3.4 Dynamic-C-Entwicklungsumgebung

Dynamic C ist eine Entwicklungsumgebung für Eingebettete Systeme mit Rabbit-Prozessoren. Alle zur Entwicklung notwendigen Arbeitsschritte können in dieser Umgebung abgewickelt werden. So ist zum Schreiben des Codes ein Texteditor integriert. Programmiert wird der Rabbit Powercore 3800 in einer ISO/ANSI C ähnlichen Sprache, die speziell auf Eingebettete Systeme zugeschnitten ist. Es kann auch passagenweise direkt in Assembler programmiert werden. Die Arbeitsschritte Kompilieren, Linken und Übertragen werden in Dynamic C zu einem untrennbaren Schritt zusammengefasst. Ein Programm wird also immer als Ganzes direkt in den Speicher des angeschlossenen Rabbit Powercore 3800 kompiliert. Die Entwicklungsumgebung stellt zudem umfangreiche Debbuging-Möglichkeiten bereit. In Abbildung 3.12 ist ein Screenshot der Entwicklungsumgebung zu sehen.

3.4.1 Unterschiede Dynamic C zu ISO/ANSI C

Im folgenden Abschnitt sollen die Unterschiede zwischen ISO/ANSI C und Dynamic C dargelegt werden. Die Softwareentwicklung für Eingebettete Systeme unterscheidet sich von der für

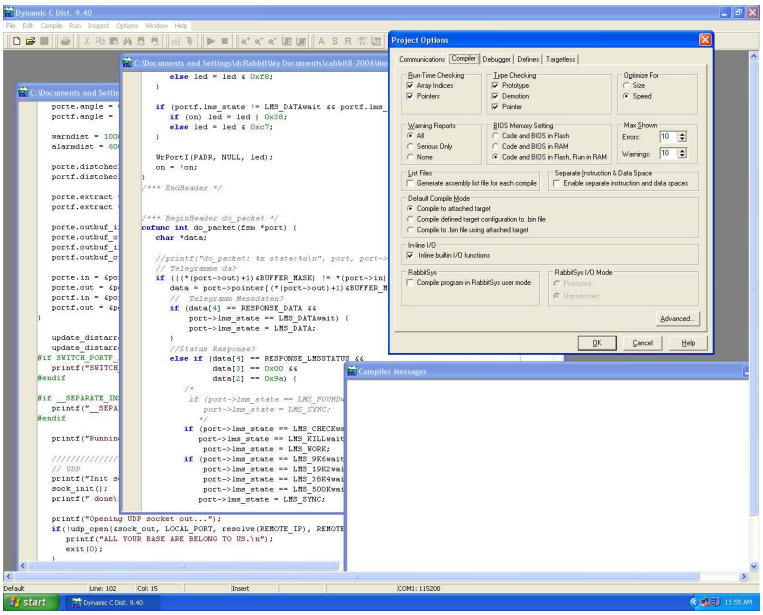


Abbildung 3.12: Screenshot der Dynamic-C-Entwicklungsumgebung

Desktop-Rechner in vielen Punkten. Der Standard-C-Compiler geht davon aus, dass auf dem System bereits ein Betriebssystem läuft und dem Programm ein bereinigter Speicherbereich zusteht. Eingebettete Systeme hingegen können batteriegestützte Speicher haben, die zum Programmstart schon Daten enthalten. Dieses kann auch gewünscht sein, wenn das System mit den gespeicherten Daten weiterarbeiten soll.

Es ergibt sich für den Programmierer folgende Abweichung: Wird bei der Initialisierung einer Variablen ein Wert zugewiesen (z.B. `int i=2;`), wird der Wert als Konstante im Flash-Speicher abgelegt und kann zur Laufzeit nicht mehr verändert werden. Ist dieses nicht gewünscht, muss der Variablen in einer `#GLOBAL_INIT` genannten Passage der Wert zugewiesen werden. Diese Passage wird nach Programmstart einmal aufgerufen und kann mehrere Variablen mit Werten belegen. Erfolgt keine Wertzuweisung, so hängt der Wert von dem momentanen Zustand des Speichers ab.

Bei der Einbindung externer Bibliotheken gibt es ebenfalls Unterschiede. Während das Einbinden in ISO/ANSI C über eine `#INCLUDE`-Anweisung erfolgt, müssen in Dynamic C benutzte Bibliotheken mit der Anweisung `#USE` bekanntgegeben werden. Innerhalb der Bibliotheken müssen in Kommentaren bestimmte Schlüsselwörter eingefügt werden, um Funktionen zu deklarieren. Diese sind in [DynCUM, S. 39] genauer beschrieben. Dieser Unterschied liegt darin begründet, dass der Compiler das gesamte Programm als einen Teil kompiliert und nicht die Bibliotheken einzeln kompiliert und zusammenfügt. In klassischen C Umgebungen brauchen nur die Programmteile neu übersetzt werden, die verändert wurden. Aufgrund der beschränkten Größe der Dynamic-C-Programme ergibt sich hierdurch allerdings keine große Verzögerung. Zeiger auf Funktionen sollten nur verwendet werden, wenn keine Argumente beim Funktionsaufruf übergeben werden. Der Dynamic-C-Compiler kann in dem Fall die Korrektheit der übergebenen Argumente nicht prüfen.

Des Weiteren ergibt sich die Abweichung, dass der Compiler als Datentyp keine Bitfelder unterstützt. In Dynamic C wurden auch viele Konstrukte hinzugefügt, die in ISO/ANSI C nicht existieren. Es können spezielle Funktionsketten deklariert werden, die erlauben, dass verschiedene Code-Segmente bei Aufruf einer Funktion ausgeführt werden. Neue Möglichkeiten ergeben sich hierdurch nicht, es besteht jedoch die Möglichkeit, ohne großen Aufwand bestehende Funktionsketten um weitere Teilfunktionen zu erweitern. Viele weitere Konstrukte dienen dem Multitasking und werden in folgendem Abschnitt vorgestellt.

3.4.2 Multitasking

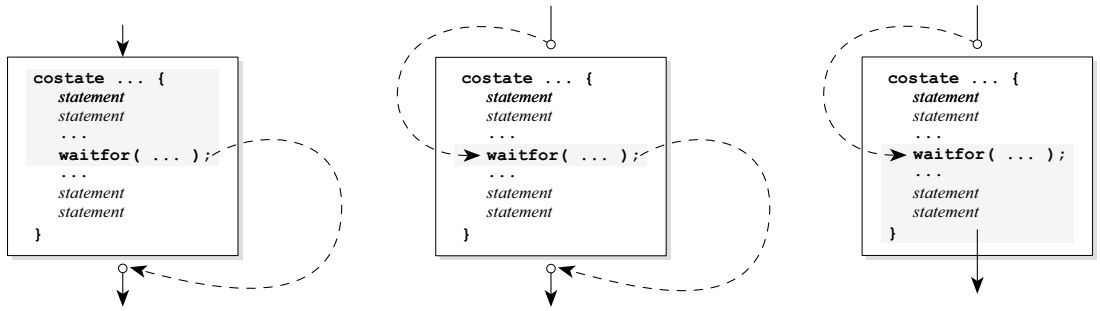
In der Regel erfolgt der Betrieb des Rabbit Powercore 3800 ohne ein laufendes Betriebssystem. Dies hat zur Folge, dass keine Möglichkeit besteht, Programme in verschiedene nebenläufige Threads zu unterteilen, denen jeweils Rechenzeit zugeteilt wird. Aus diesem Grund wurden verschiedene Möglichkeiten in Dynamic C eingeführt, um trotzdem das parallele Ausführen von nebenläufigen Programmteilen zu ermöglichen. Da immer nur ein Programmteil zur Zeit ausgeführt werden kann, müssen verschiedene Strukturen geschaffen werden, um eine Koordination zwischen den verschiedenen Teilen zu ermöglichen. Hierbei wird zwischen kooperativem und präemptiven Multitasking unterschieden. Beim kooperativem Multitasking geben die einzelnen Aufgaben die Kontrolle freiwillig wieder an die nächste Aufgabe ab, während beim präemptiven Multitasking eine Aufgabe für eine vorher festgelegte Zeit ausgeführt wird.

Kooperatives Multitasking

Für das kooperative Multitasking sind in Dynamic C die Kontrollstrukturen `costate` und `cofunc` vorgesehen. Die Benutzung der Costatements soll an einem Beispiel erläutert werden:

```
while(1) {
   costate {                    // Aufgabe 1
      x = y + i;
      waitfor( DelaySec(30L) );
   }
   costate {                    // Aufgabe 2
      i++;
      waitfor( DelaySec(1L) );
   }
   costate {                    // Aufgabe 3
      function3();
      waitfor( DelaySec(60L) );
   }
}
```

Innerhalb einer endlosen `while`-Schleife werden verschiedene Costatements ausgeführt, in diesem Beispiel drei. Die Anweisungen in einem Costatement werden ausgeführt, bis innerhalb des Costatements eine `waitfor()`-Funktion auftritt. An diesem Punkt wird die Kontrolle dem nächsten Costatement übergeben. Sind auf diese Art alle Costatements abgearbeitet, beginnt der Durchlauf durch die `while`-Schleife erneut. Bei jedem Costatement wird nun überprüft, ob die Bedingung bereits eingetreten ist, und gegebenenfalls wird mit der Anweisung nach der



(a) erster Schleifendurchlauf (b) Warten auf Ereignis (c) weiteren Code ausführen

Abbildung 3.13: Ablauf der Costatement Abarbeitung, Quelle: [DynCUM, S. 50]

`waitfor()`-Funktion weitergearbeitet. Sollte ein Costatement abgearbeitet sein, wird es beim nächsten Schleifendurchlauf erneut gestartet. Dieser Ablauf ist in Abbildung 3.13 dargestellt.

Beim Verwenden der Costatements liegt es in der Verantwortung des Programmierers, den Ablauf des Gesamtprogramms soweit zu analysieren, dass alle zeitkritischen Costatements oft genug zur Ausführung kommen. Hierbei muss bemessen werden, welche maximale Latenz sich ergibt, wenn alle anderen Costatements sich gerade in einem Zustand befinden, in dem sie viel Rechenzeit benötigen. Eine mögliche Last durch auftretende Interrupts ist dabei ebenfalls zu berücksichtigen. Eine Möglichkeit, auf die Ausführung von Costatements Einfluss zu nehmen, ist das Einfügen von `yield()`-Statements. Diese führen dazu, dass die Ausführung des aktuellen Costatements unterbrochen und beim nächsten Aufruf dieses Statements an entsprechender Position fortgesetzt wird. Auf diese Weise können Costatements, die besonders lange Ausführungszeit, aber geringere Priorität haben, zusammen mit zeitkritischen Costatements ausgeführt werden. Werden die `yield()`-Statements regelmäßig aufgerufen, kann entsprechend oft überprüft werden, ob in anderen Costatements Code auszuführen ist. Es bestehen noch einige weitere Möglichkeiten zur Steuerung von Costatements. Diese sollen hier nur kurz erwähnt werden:

- Abbrechen von Costatements mit dem `abort()`-Befehl
- Konfigurieren, ob Costatements nach dem Anhalten beim nächsten Durchlauf weiter ausgeführt werden
- Costatements als Datentyp direkt ansprechen und konfigurieren

Innerhalb der Costatements empfiehlt es sich, statt normaler Funktionen Cofunctions zu verwenden. Cofunctions ähneln normalen Funktionen darin, dass sie ebenfalls einen Rückgabewert haben können und der Funktion beim Aufruf Argumente übergeben werden können. Diese Funktionen können jedoch nur innerhalb eines Costatements wie in folgendem Beispiel mit dem Befehl `waitfordone` aufgerufen werden:

```
waitfordone x = Cofunc1(y);
```

Innerhalb dieser Cofunction kann ebenfalls mit `yield()`- und `waitfor()`-Funktionen die Kontrolle abgegeben werden. Wenn klassische Funktionen innerhalb von Costatements ausgeführt werden, blockieren diese während ihrer Ausführungszeit zwangsweise die übrigen Costatements.

Präemptives Multitasking

Beim präemptiven Multitasking wird die Kontrolle den einzelnen Tasks zwangsweise entzogen. Dies erfolgt nach einem spezifizierbaren Zeitraum, der in Vielfachen von 1/1024 Sekunde angegeben wird. Statt Costatements werden nun Slice Statements unter Angabe der maximalen Ausführungszeit erzeugt. Innerhalb der Slice Statements führen `yield()`-, `waitfor()`-, und `abort()`-Anweisungen ebenfalls zum vorzeitigen Verlassen des Codeblocks. Erfolgt keine dieser Anweisungen, wird die Ausführung des Codeblocks nach abgelaufener Zeit unterbrochen und beim nächsten Eintritt in das Slice Statement an entsprechender Stelle fortgeführt. Innerhalb einer globalen `while`-Schleife können Costatements und Slice Statements gemischt verwendet werden.

3.4.3 Programmierung in Assembler

Innerhalb Dynamic C kann passagenweise in Assembler programmiert werden. Dabei werden direkt Maschinenbefehle statt C-Befehle verwendet. Trotz leistungsfähiger Compiler kann die Effizienz durch einen handgeschriebenen Assembler-Code noch optimiert werden, so dass Ausführungszeit oder auch Codelänge verbessert werden. Eine Assembler-Passage wird mit `#asm` eingeleitet und endet mit `#endasm`. Ein entscheidender Vorteil besteht darin, dass exakte Aussagen über die Ausführungszeit gemacht werden können, da bei jedem Assemblerbefehl diese Zeit fest vorgegeben ist. Besonders bei Interrupt-Service-Routinen ist die genaue Kenntnis der Ausführungszeit von Vorteil, da die dort auszuführenden Aufgaben oft zeitkritisch sind.

Ein Problem, das sich beim Mischen von C- und Assembler-Code ergibt, ist, dass die Konsistenz der Register gewahrt werden muss. Innerhalb von komplett in Assembler geschriebenen Funktionen erledigt dieses der Compiler. Werden jedoch in Assembler-Passagen oder auch in Interrupt-Service-Routinen die Registerinhalte verändert, so müssen diese wiederhergestellt werden, damit es nicht zu Fehlern kommt. Dieses wurde bereits ausführlich in Abschnitt 3.2.5 erklärt.

3.4.4 Funktionen und Bibliotheken

Dynamic C wird mit einer Vielzahl von Bibliotheken ausgeliefert. Die Anwendungsentwicklung wird somit deutlich erleichtert, da für die Standardaufgaben schon fertige Lösungen existieren. Einige Aufgabengruppen, zu denen Funktionen angeboten werden, sind folgende:

- Verschlüsselung
- Prüfsummenberechnung
- Dateisysteme
- Fliesskommaarithmetik
- Bearbeiten von Strings
- TCP/IP Kommunikation

Ebenfalls ist das echtzeitfähige Betriebssystem MicroC/OS-II Bestandteil von Dynamic C. Im Rahmen dieser Arbeit soll das Betriebssystem aber nicht verwendet werden, da die Programmierung möglichst hardwarenah erfolgen soll. Nur so können verlässliche Aussagen über die

Performance gemacht werden, da der Einfluss eines Betriebssystems auf den Programmablauf nicht überschaubar wäre.

3.4.5 Kompilieren und Debuggen

In folgendem Abschnitt geht es um den Kompiliervorgang und das Debuggen. Während des Kompiliervorganges wird das Programm in den Speicher des Rabbit Powercore 3800 geschrieben. Hierbei stehen verschiedene Möglichkeiten zur Auswahl:

- Compile to Ram
- Compile to Flash
- Compile to Flash, Run in Ram

Wird das Programm direkt in den Arbeitsspeicher geladen, so bleibt es nur erhalten, solange die Stromversorgung bestehen bleibt. Es kann für reine Testprogramme trotzdem sinnvoll sein, diese nur in dem RAM auszuführen, da ein Flash-Speicher eine begrenzte Zahl an Schreibzyklen hat. In diesem Modus wird der gesamte ein Megabyte große Adressraum auf den RAM abgebildet.

Wird das Programm in den Flash-Speicher kompiliert, so bleibt es persistent erhalten. Der Rabbit Powercore 3800 führt dieses Programm nach dem Einschalten der Betriebsspannung automatisch aus, wenn kein Programmierkabel angeschlossen ist. In diesem Kompiliermodus ist der Speicherbereich, in dem Code und Konstanten gespeichert sind, auf den Flash-Speicher abgebildet. Die Variablen und der Stack liegen auch hier in dem RAM-Bereich, da sich diese Daten häufig ändern, was zu einer stark verkürzten Lebensdauer des Flash-Speichers führen würde. Außerdem erfordern Schreibzugriffe auf den Flash-Speicher des Rabbit Powercore 3800 eine lange Ausführungszeit im Bereich von 5 ms bis 20 ms. Der Kompiliermodus „Compile to Flash, Run in Ram“ arbeitet ähnlich wie das Kompilieren in den Flash-Speicher. Der Unterschied besteht darin, dass zum Programmstart der Code komplett in den RAM geladen wird und von dort aus ausgeführt wird. Dazu wird im Betrieb die Speicherabbildungsstrategie verändert. Ein Vorteil besteht darin, dass es bei Speicherzugriffen auf den Code-Bereich oder auch bei fehlerhaften Zugriffen nicht zu Schreiboperationen auf den Flash-Speicher kommt. Jedoch geht durch die doppelte Speicherung des Codes Speicherplatz verloren, der durch geschicktes Ansteuern sonst komplett genutzt werden könnte.

Die Datenverbindung zwischen dem Rabbit Powercore 3800 und dem Entwicklungsrechner wird über ein serielles Programmierkabel hergestellt. Dieses kann auf zwei verschiedene Arten verbunden werden. Im Programmiermodus startet der Rabbit Powercore 3800 nicht das Programm aus dem Flash-Speicher, sondern versucht, direkt über die serielle Schnittstelle zu booten. Dies gelingt, sobald seitens der Entwicklungsumgebung der Kompiliervorgang gestartet wird. Die nötigen Operationen werden ausgeführt und das System wird in die Lage versetzt, das Programm entgegenzunehmen. Die Baudrate liegt beim Verbindungsaufbau zunächst bei 2400 Baud und wird zur Übertragung auf 115,2 kBd gesteigert. Das Programm kann dann gestartet werden und das Debuggen kann ebenfalls bei dieser Anschlussart des Kabels erfolgen. Beim Debug-Modus startet der Rabbit Powercore 3800 das schon vorhandene Programm und nutzt diese Verbindung nur zum Debuggen. Die wichtigsten Möglichkeiten, die das Debuggen bereitstellt, werden im Folgenden vorgestellt.

printf() Über die Funktion `printf()` können Meldungen ausgegeben werden. Diese erscheinen dann in einem Ausgabefenster innerhalb der Entwicklungsumgebung. Es lassen sich so Werte von Variablen überprüfen und Informationen über die aktuelle Position der Programmausführung innerhalb des Quellcodes gewinnen. Die Daten werden dabei über die serielle Schnittstelle versendet, was eine zusätzliche Systemauslastung erzeugt. Sollten Komplettabstürze des Systems auftreten, können auch keine Informationen mehr gesendet werden. Der fehlerhafte Abschnitt im Programm muss also gesucht werden, indem nach jedem eventuell kritischen Befehl eine Meldung ausgegeben wird.

Breakpoints Durch das Setzen von Breakpoints kann das Programm an beliebigen Stellen unterbrochen werden. Die Programmausführung erfolgt zuvor in voller Geschwindigkeit. Stoppt die Programmausführung an einem Breakpoint, können weitere Debug-Funktionen angewendet werden.

Single Stepping Das Programm kann Befehl für Befehl ausgeführt werden. Der Ablauf kann dabei auf Quellcodeebene verfolgt und analysiert werden.

Watch Expressions In der Entwicklungsumgebung kann ständig der Zustand von verschiedenen Variablen überwacht werden. Ebenfalls lassen sich logische Ausdrücke oder Teile von diesen beobachten. Das ständige Auswerten erzeugt zusätzliche Systemauslastung, aus diesem Grund kann das Timing des Programms beeinflusst werden.

Evaluate Expressions Die Evaluate Expressions dienen zum manuellen Setzen von Variablenzuständen. Dazu muss das Programm an entsprechender Stelle unterbrochen werden und an gewünschter Stelle die Zuweisung in der Entwicklungsumgebung eingegeben werden.

Memory Dump Bei gestopptem Programmablauf kann der Zustand des Speichers auf der Seite der Entwicklungsumgebung ausgelesen werden. Es ist ebenfalls möglich, den gesamten Speicherinhalt in eine Datei zu schreiben.

Für die genannten Debug-Funktionen muss in das Rabbit-3000-Programm ein Debug-Kernel mit eingebunden werden. Dieses kann in der Entwicklungsumgebung aktiviert werden. Es wird zusätzlicher Speicher auf dem Rabbit Powercore 3800 benutzt und auch die Systemauslastung steigt. Wird kein Debug-Kernel verwendet, bricht die serielle Verbindung sofort nach Programmstart ab.

3.5 Zusammenfassung

Es wurde ein Konzept für einen Realisierungsweg aufgestellt und eine Entscheidung für die zu verwendende Hardware getroffen. Der Systementwurf findet auf Basis eines Powercore 3800 statt, der auf dem Rabbit-3000-Mikroprozessor basiert. Die Funktionseinheiten dieses Prozessors wurden vorgestellt. Bei den Beschreibungen wurde auf die Konfiguration der Komponenten

zur Anbindung zweier SICK LMS 200 eingegangen. Es konnte so ermittelt werden, dass die Implementierung des Systems theoretisch mit der gewählten Hardware möglich ist. Des Weiteren wurden der Powercore 3800 und die Entwicklungsumgebung Dynamic C vorgestellt.

4 Prototyp mit Schnittstellenanbindung

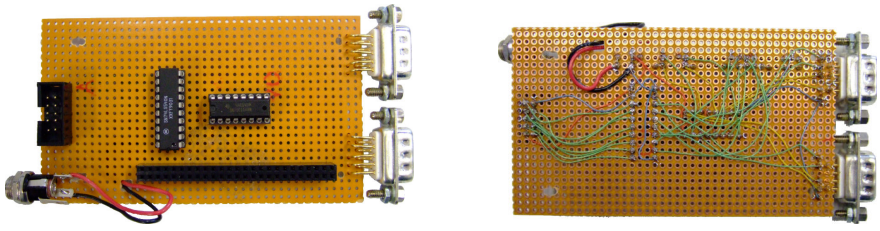
In diesem Kapitel wird der entwickelte Prototyp vorgestellt. An diesem wird die Funktionsfähigkeit der Hardware und der vorhandenen Schnittstellen getestet. Zunächst wird die Trägerplatine für den Rabbit Powercore 3800 mit der darauf befindlichen Hardware vorgestellt. Im zweiten Abschnitt folgt die Beschreibung eines seriellen Testgenerators, der entwickelt wird, um die zuverlässige Funktion der seriellen Datenübertragung bei 500 kBd zu testen. Der dritte Abschnitt behandelt die Anbindung des Prototypen an das Hostsystem über Ethernet.

4.1 Hardwareprototyp

Der Hardwareprototyp setzt sich aus einem Rabbit Powercore 3800 und einer selbst entwickelten Platine zusammen. Eine Fotografie der Platine ist in Abbildung 4.1 zu sehen. Verbunden sind die beiden Platinen über eine 50-polige Pfostenleiste (Abbildung 4.1 (a) unten). Über diese Pfostenleiste stehen alle Schnittstellen des Rabbit 3000 zur Verfügung. Die Trägerplatine vereint Anschlüsse und Bauteile, um den Powercore 3800 mit weiteren Komponenten zu verbinden.

Die Stromversorgung kann wahlweise vom Powercore 3800 über das mitgelieferte Netzteil oder über die selbstgebaute Trägerplatine erfolgen. Über die Pfostenleiste wird die jeweils andere Platine mitversorgt. Der Powercore 3800 besitzt einen Spannungswandler, so dass eine 12-Volt-Gleichstromquelle genutzt werden kann. Dieser Spannungswandler kann auch über die Trägerplatine gespeist werden. Der Anschluss für die Stromversorgung ist unten links in Abbildung 4.1 (a) zu sehen.

Die Lasermesssysteme werden mit einem 9-poligen Sub-D-Stecker ausgeliefert. Auf der Trägerplatine sind dazu passende Buchsen. Außer dem passenden Steckkontakt muss der Spannungspegel von den RS-422-Lasermesssystem-Schnittstellen auf die CMOS-Pegel des Powercore 3800 angepasst werden. Für die Pegelumsetzung ist ein SN75C1168-Transceiver-Chip auf der Platine. Der Transceiver-Chip ist in Abbildung 4.1 (a) von den beiden Chips in der Mitte der rechte



(a) Oberseite (b) Unterseite

Abbildung 4.1: Foto der Trägerplatine

(a) RS-422-Driver (b) RS-422-Receiver

Abbildung 4.2: Logikdiagramm vom SN75C1168 Transceiver

Input	Enable	Output	
D	DE	Y	Z
H	H	H	L
L	H	L	H
X	L	Z	Z

(a) RS-422-Driver

Differential Input A-B	Output R
$V_{ID} \geq 0,2$ V	H
-0,2 V<V_{ID}<0,2 V	?
$V_{ID} \leq -0,2$ V	L
open	H

(b) RS-422-Receiver

Tabelle 4.1: Funktionstabelle vom SN75C1168 Transceiver

Baustein. Der SN75C1168 besitzt zwei RS-422-Transmitter/Driver und zwei RS-422-Receiver. Damit ist er in der Lage, für zwei Lasermesssysteme die Pegel zu wandeln.

In Abbildung 4.2 ist das Logikdiagramm des Tranceiver-Chips für einen Transmitter/Driver in (a) und einen Receiver in (b) abgebildet. Die Tabelle 4.1 zeigt die zum Transceiver gehörende Funktionstabelle.

Auf der Trägerplatine befindet sich im Weiteren ein SN74LS540N-Chip als Treiber zur Ansteuerung von LEDs. Der Rabbit 3000 liefert nicht genug Strom, um die LEDs direkt zu treiben. Diese befinden sich auf einer zusätzlichen Platine und werden über ein Flachbandkabel an die Trägerplatine angebunden. Trägerplatine und Rabbit Powercore 3800 sind in ein Gehäuse eingebaut. Auf der Vorderseite sind LEDs eingelassen (Abbildung 4.3), sie zeigen Ergebnisse der Vorverarbeitung an (siehe Abschnitt 5.5.1). Die untere blaue LED ist direkt an die Stromversorgung gekoppelt.

4.2 Serieller Testgenerator

Für einen Test der seriellen Schnittstelle des Rabbit 3000 und der angebundenen Transceiver-Chips muss geprüft werden, ob das Empfangen von Daten erwartungsgemäß funktioniert. Eine Möglichkeit wäre hierbei, ein SICK-Lasermesssystem mit dem Prototyp zu verbinden. Dies wäre aber mit zahlreichen Nachteilen verbunden.

Zum einen würde eine frühzeitige Aufnahme von Tests mit einem realen Lasermesssystem dazu führen, dass dieses am Arbeitsbereich TAMS länger als nötig für andere Projekte blockiert wäre. Des Weiteren würde der relativ komplexe Kommunikationsablauf zwischen einem SICK-Lasermesssystem und dem Kommunikationspartner eine Vielzahl von Fehlerquellen bieten. Eine Verifikation der Testergebnisse würde sich schwierig gestalten, da hierzu die empfangenen Daten

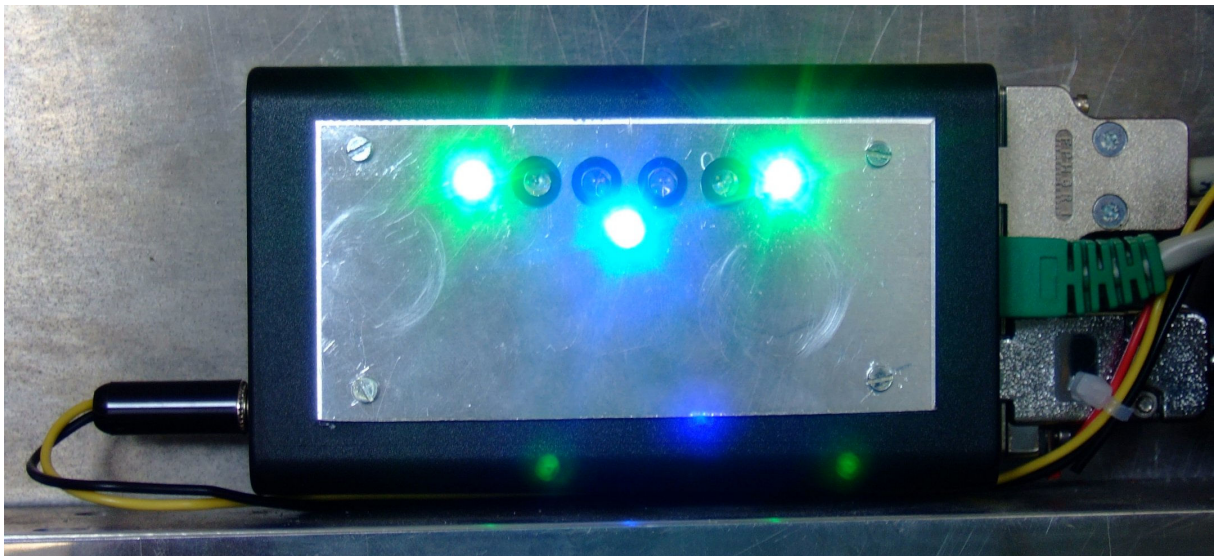


Abbildung 4.3: Fertig gebautes Gehäuse mit LEDs

ausgewertet werden müssten und eine Übereinstimmung mit den Sollwerten schwer zu überprüfen wäre.

Aus diesen Gründen wurde entschieden, einen seriellen Testsender zu konstruieren. Dieser soll über eine RS-422-Schnittstelle verfügen und mit einer Baudrate von 500 kBd arbeiten. Das gesendete Datenbyte soll dabei frei wählbar sein. Auf der Seite des Rabbit Powercore 3800 sollen die Daten dann empfangen werden. Die Korrektheit der Daten wäre somit sofort überprüfbar, da im Testaufbau bekannt ist, was gesendet wurde.

Hierfür wurden zwei verschiedene Möglichkeiten in Betracht gezogen:

- Aufbau einer entsprechender Schaltung aus elektronischen Bauteilen (Realisierung als diskreter Aufbau)
- Realisierung durch programmierbare Hardware (FPGA-Entwicklungsplatine, Codierung der Schaltung in VHDL, Anpassen des Pegels über RS-422-Transmitter)

4.2.1 Realisierung als diskreter Aufbau

In [Kai93] ist die Schaltung eines RS-232-Testgenerators beschrieben. Diese ist in Abbildung 4.4 dargestellt. Mit dieser Schaltung kann wiederholt ein frei wählbares Byte über eine RS-232-Verbindung gesendet werden.

Durch Modifikation dieser Schaltung ließe sich ein RS-422-Testgenerator entwickeln. Dazu soll hier kurz die Funktionsweise der Schaltung erklärt werden: Der Timer 555 dient als Taktgenerator. Über das 10 kΩ Potentiometer lässt sich der Ausgangstakt variieren. In der dargestellten Schaltung soll der Takt bei 1200 Hz liegen, damit eine Übertragungsrate von 1200 Baud erreicht wird. Der Timerblock würde bei der Entwicklung des seriellen Testgenerators durch einen externen Frequenzgenerator ersetzt werden, da eine Frequenz einer ganz anderen Größenordnung nötig wäre (0,5 MHz). Dieser würde dann an den Clock-Eingang des Dekadenzählers 4017

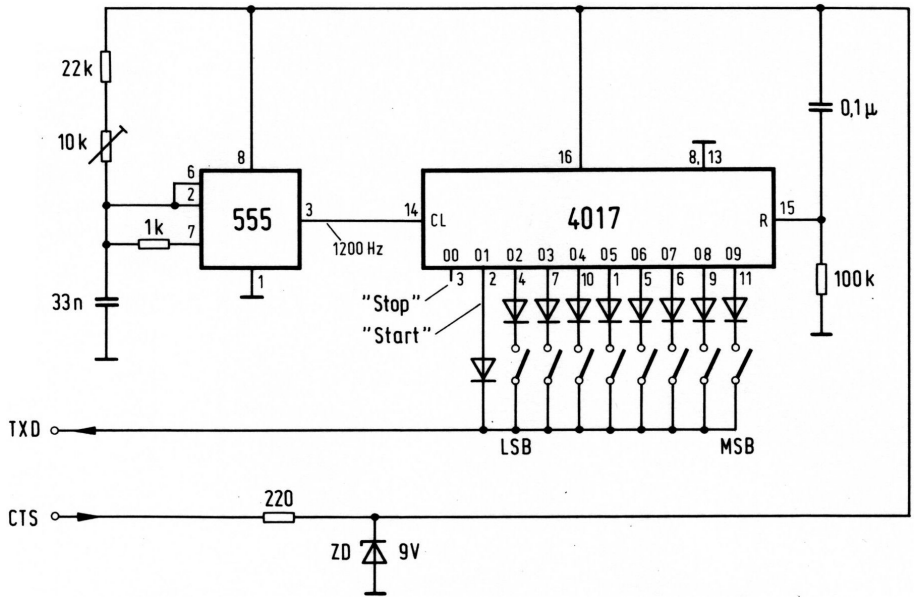


Abbildung 4.4: Hardwarerealisierung eines seriellen Testsenders, Quelle: [Kai93]

angeschlossen werden. Die Funktion des Dekadenzählers besteht darin, dass nacheinander die Ausgänge 00 bis 09 durchgeschaltet werden, wobei zur Zeit immer nur ein Ausgang aktiv ist. Mit den Schaltern, die sich am besten als DIP-Switches realisieren ließen, wäre das zu sendende Byte einzustellen. Bei geschlossenem Schalter wird die TXD-Leitung auf *high*-Pegel gezogen, wenn entsprechender Ausgang am Dekadenzähler gerade aktiv ist. Ist der Schalter offen, bleibt die TXD-Leitung auf *low*-Pegel. Das Stoppbit erzeugt immer einen *low*-Pegel, das Startbit immer einen *high*-Pegel, deshalb sind die Ausgänge 00 und 01 nicht schaltbar. Der Pegel an der TXD-Leitung muss noch mit einem zusätzlichem RS-422-Transmitter auf die differentielle Übertragungsmethode angepasst werden. Zusätzlich muss noch sichergestellt werden, dass der Dekadenzähler für eine Frequenz von 0,5 MHz geeignet ist.

Auf diese Art ließe sich kostengünstig ein RS-422-Testgenerator realisieren, mit dem die Übertragungsstrecke und der Datenempfang auf Seite des Rabbit-3000-Mikroprozessor getestet werden könnten. Die Grenzen dieser Realisierungsmöglichkeit sind jedoch, dass sich immer nur einzelne Byte senden lassen. Bytesequenzen lassen sich nicht generieren, auch nicht durch Benutzung der DIP-Switches, da bei einer Baudrate von 500 kBd jede Sekunde 50000 Byte gesendet werden und in dieser Geschwindigkeit natürlich nicht geschaltet werden kann. Aus diesen Gründen wurde dieser Testgenerator als diskreter Schaltungsaufbau nicht realisiert.

4.2.2 Programmierung in VHDL

Am Arbeitsbereich TAMS sind FPGA[1]-Entwicklungsboards der Firma Altera vorhanden. Mit VHDL lässt sich das Verhalten der Hardware direkt beschreiben. Beim Synthetisieren wird aus der Verhaltensbeschreibung der Code für die FPGA-Entwicklungsplatine generiert, der dann bewirkt, dass sich diese entsprechend der zuvor angegebenen Verhaltensbeschreibung verhält.

[1] *Field Programmable Gate Array*

Weitere Informationen zu VHDL sind in [Mae05] zu finden. Der Testgenerator soll folgende Funktionen bereitstellen:

- wiederholtes Senden eines vorgegebenen Byte mit variabler Pause zwischen den einzelnen Aussendungen
- Senden einer bestimmten Bytefolge (z.B. eines Zählers)
- Senden von Telegrammen, die den Spezifikationen des LMS 200 entsprechen

Die Frequenz soll dabei möglichst variabel sein, um Tests bei verschiedenen Baudraten durchführen zu können. Für die Sendeleitung der seriellen Übertragungsstrecke wird ein Ausgang des FPGA-Boards verwendet. Der Ausgang hat einen CMOS-Pegel, also 0 V für einen *low*-Pegel und 3,3 V als *high*-Pegel. Wie bei dem vorgestellten hardwarebasierten Testgenerator muss das Signal über einen geeigneten Treiberchip für den Übertragungsweg auf RS-422-Pegel gewandelt werden.

Implementierung des Testgenerators

Im Folgenden sollen die Abläufe innerhalb der in VHDL beschriebenen Hardware dargestellt werden. Hierbei soll nur auf die Funktionen eingegangen werden, der VHDL-Code wird nicht weiter beschrieben. Auf der beiliegenden CD findet sich der Quellcode im Verzeichnis `/quellcode/vhdl`.

Die Schaltung wird von einem 33,3 MHz Quarz getrieben. Dieser Takt kann innerhalb des FPGA-Boards variiert werden, indem ein Multiplikator und ein Divisor angegeben wird. Eine Multiplikation um den Faktor drei und anschließende Division durch zwei ergibt eine Taktrate von 50 MHz, ein Vielfaches der benötigten 0,5 MHz. Dies ist wichtig, um mit einem Teiler möglichst genau die gewünschte Frequenz zu erzeugen. Realisiert wird dieser Teiler durch einen Zähler, der kontinuierlich von 0 bis 100 mit einer Zählrate von 50 MHz läuft. Beim Zählstand von 100 wird der Zähler zurückgesetzt und das nächste Bit am Ausgang angelegt, sofern nicht gerade eine Pause zwischen zwei Byte erfolgen soll.

Innerhalb des Chips wird ein Zufallszahlengenerator benötigt, der folgende Aufgaben hat:

- Generieren von Pausen zwischen zwei Byte mit zufälliger Länge
- Erzeugen von mehreren Byte mit zufälligem Inhalt
- Erzeugen von Telegrammen mit zufälliger Länge

Da ein echter Zufallszahlengenerator nicht implementierbar ist, wird ein LFSR[2] zur Implementierung eines Pseudozufallszahlengenerators verwendet. Pseudozufallszahlen sind eine feste Abfolge von Werten, die jedoch wie zufällig generierte Werte aussehen. In [Xil03] ist beschrieben, wie mit einem LFSR eine Zahlenfolge mit 65-Bit-Werten generiert werden kann. Es wird das Polynom $p(x)\ X_{65}\ X_{47}\ X_0$ (Fibonacci-Notation) verwendet. Wird der 65-Bit-Wert als Binärzahl betrachtet, so berechnet sich die nächste Zahl nach folgendem Muster: Alle Binärstellen werden um eine Position nach rechts verschoben. Die am linken Ende freiwerdende Position wird über eine XOR-Verknüpfung des 47. und des 65. Bit (Werte der Positionen vor dem Verschieben) berechnet. Initialisiert wird das Register mit einem beliebigen Wert außer null, da sich dieser Wert nie ändern würde. In der Abfolge kommen alle Werte vor, außer dem Wert null. Es stehen

[2] *Linear Feedback Shift Register*

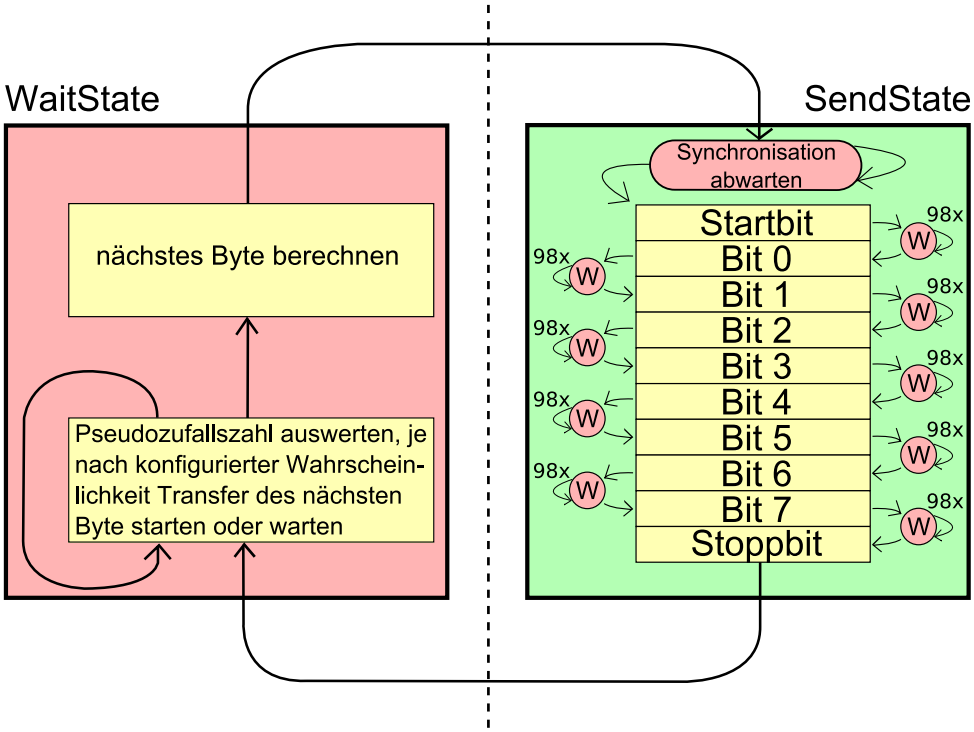


Abbildung 4.5: Verhalten des RS-422 Testgenerators

somit Zufallszahlen mit bis zu 65 Bit bereit. Wenn kürzere Zahlen benötigt werden, können die restlichen Bit ignoriert werden.

Das Verhalten des seriellen Testgenerators lässt sich als Automat mit zwei Zuständen beschreiben. Diese Zustände sind der `SendState` und der `WaitState`. In Abbildung 4.5 ist der Ablauf der Aktionen im jeweiligen Zustand dargestellt. Während sich das System im `SendState` befindet, wird ein komplettes Byte inklusive Start- und Stoppbit gesendet. Werden die Zustände `Start-Bit`, `Stop-Bit` oder einer der Zustände `Bit 0` bis `Bit 7` eingenommen, erfolgt eine Beschaltung der Sendeleitung des FPGA-Boards. Da der Ausgang nur zu jedem 100. Takt neu beschaltet werden soll, muss solange in einer Warteschleife verblieben werden. Diese ist durch die mit `W` benannten Zustände und die entsprechenden Pfeile angedeutet. Vor dem Senden des Startbit muss auf eine Synchronisation gewartet werden. Dieses hat zum Einen technische Gründe, da das Weiterschalten der Bit über einen ständig laufenden Zähler erfolgt, der beim Erreichen der 100 wieder von null beginnt. Zum Anderen bewirkt dieses Verhalten, dass das Stoppbit die vorgeschriebene Länge hat, falls im `WaitState` eine zu kurze Pause erzeugt wird.

Nach dem Senden eines Byte werden die Aktionen im `WaitState` ausgeführt. In diesem Zustand wird eine Pause mit zufälliger Länge zwischen zwei Byte eingelegt und das nächste zu sendende Byte berechnet. Zunächst erfolgt ein Verbleiben in einem Zustand, wo in jedem Takt nun beliebig viele Bit der Pseudozufallszahl geprüft werden, ob sie gesetzt sind. Wenn alle Bit gesetzt sind, wird der Zustand verlassen. Je nach gewünschter Wahrscheinlichkeit kann die Zahl der überprüften Bit durch Anpassen des Quellcodes variiert werden. Soll keine Pause zwischen zwei Byte eingelegt werden, kann der Vergleich eingesetzt werden, der immer wahr ist (Tautologie).

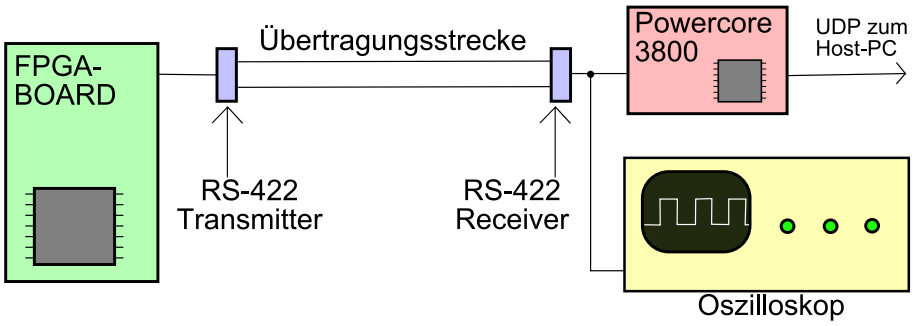


Abbildung 4.6: Aufbau des Übertragungstests

Kommt es zum Verlassen der Warteschleife innerhalb des `WaitState`, wird das nächste Byte bestimmt. Dieses kann wie zuvor beschrieben auf drei verschiedene Arten erfolgen.

Die verschiedenen Betriebsmodi

Mit den einzelnen Betriebsmodi lassen sich verschiedene Tests durchführen. Zunächst soll der Testaufbau betrachtet werden, der in Abbildung 4.6 dargestellt ist. Das Ausgangssignal des FPGA-Boards wird mit einem RS-422-Transmitter über die Datenleitung gesendet. Auf der Empfängerseite wird das Signal von einem RS-422-Receiver wieder gewandelt und dem Rabbit Powercore 3800 zugeführt. Die Auswertung erfolgt über die serielle Debug-Verbindung oder über UDP-Pakete. Des Weiteren wird das zurückgewandelte Signal mit einem Oszilloskop analysiert.

Senden eines festen Byte Das wiederholte Senden eines Byte dient zum grundlegenden Test der Übertragungsstrecke. Auf der Seite des Rabbit Powercore 3800 kann so geprüft werden, ob nach Konfiguration der seriellen Schnittstellen, wie in 3.2.8 beschrieben, entsprechendes Byte empfangen wird. Das Byte kann dann über die Debug-Konsole ausgegeben und mit der Vorgabe verglichen werden.

Auf folgende Art lässt sich leicht feststellen, ob der Testgenerator mit der richtigen Baudrate arbeitet. Es wird der Wert 0xAA gesendet, welcher der Zahl 10101010 im Dualsystem entspricht. Diese wird nun wie in Abschnitt 2.3 erläutert mit dem niederwertigen Bit zuerst gesendet. Die Pause wird auf den Wert null gesetzt, so dass zwischen zwei Byte nur ein Start- und ein Stoppbit liegt. Es ergibt sich daraus ein Bitmuster, bei dem immer abwechselnd ein 0-Bit und ein 1-Bit gesendet wird. Dieses Rechtecksignal kann nach Passieren der Übertragungsstrecke problemlos mit einem Oszilloskop betrachtet werden, wobei moderne Oszilloskope dabei gleich die Frequenz bestimmen können. Die Fotografieeiner solchen Auswertung ist in Abbildung 4.7 zu sehen.

Bei der Frequenzauswertung wird ein Wert von 249,6 kHz gemessen, da eine Periode aus zwei Zeichen besteht. Die gemessene Baudrate liegt somit bei ca. 499 kBd statt der gewählten 500 kBd. Ursachen für diese geringe Abweichung sind möglicherweise die Messungenauigkeit des verwendeten Oszilloskops sowie leichte Abweichungen bei dem Quarz des FPGA-Boards.

Wie in 3.2.7 erläutert, wird bei dem Powercore 3800 mit einer Baudrate von ca. 496 kBd gearbeitet. Im Gegensatz dazu arbeiten die Lasermesssysteme mit einer Baudrate von 500 kBd. Deshalb

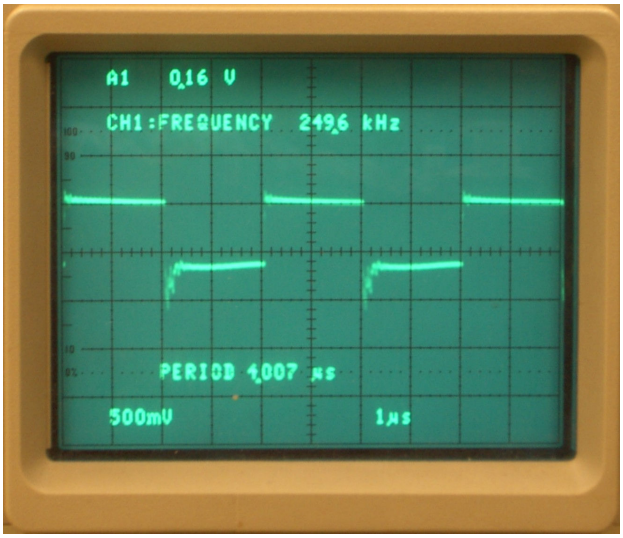


Abbildung 4.7: Oszilloskopauswertung des Signals nach Passieren der Übertragungsstrecke (Bitmuster 10101010)

ist die Möglichkeit von Vorteil, die Auswirkungen einer seriellen Kommunikation mit abweichender Baudrate zu untersuchen. Hierzu kann konfiguriert werden, dass nicht jeden 100. Takt ein neues Zeichen gesendet werden soll, sondern beispielsweise jeden 99. oder jeden 101. Takt, was einer Abweichung von ca. 1 % entspricht. Auf entsprechende Weise soll die Übertragung bei einer Abweichung bis zu 3 % auf korrekte Funktion überprüft werden, damit sichergestellt werden kann, dass die Abweichung von ca. 1 % zwischen den 496 kBd und den 500 kBd nicht an der kritischen Grenze liegt. Bei diesen Tests wurden keinerlei Übertragungsfehler beobachtet.

Senden eines hochzählenden Byte Nachdem sichergestellt ist, dass einzelne Byte übertragen werden können, wird in dem `WaitState` ein Zähler implementiert. Sobald der Zählerstand gesendet wird, erfolgt die Inkrementierung des Zählerstandes. Ist der höchstmögliche Zählerstand von 255 (0xFF) erreicht, erfolgt der Umbruch. Auf der Seite des Empfängers wird nun immer jeweils ein empfangenes Byte zwischengespeichert und mit dem nächsten Byte verglichen. Entspricht das Ergebnis dieses Vergleichs nicht der Erwartung, so kann dies zwei Ursachen haben:

- Es unterlaufen Fehler auf der Übertragungsstrecke.
- Aufgrund zu langsamer Verarbeitungsgeschwindigkeit gehen einzelne Byte verloren.

Die Fehler auf der Übertragungsstrecke können bewirken, dass ein falsches Byte empfangen wird. Ebenfalls ist denkbar, dass durch Störungen überhaupt kein Byte empfangen wird. Auf der Seite des Rabbit Powercore 3800 kann das Byte verloren gehen, wenn aufgrund hoher Systemauslastung die Interrupt-Service-Routine (ISR) nicht rechtzeitig ausgeführt wird und ein Byte verloren geht. Dieses soll dann durch Anpassen der Prioritätsstufe der ISR vermieden werden. Zusätzlich wird bei diesem Test künstlich Systemauslastung erzeugt, um zu untersuchen, ob der Datenempfang trotzdem einwandfrei funktioniert. Es ergibt sich, dass ab einer Prioritätsstufe von zwei die ISR in ihrer Ausführung nicht mehr behindert wird.

Senden von Telegrammen In diesem Testmodus sollen komplette Telegramme gesendet werden, die von der Struktur her den Telegrammen des SICK LMS 200 entsprechen. Die Länge jedes Telegramms soll variabel sein und zufällig ausfallen. Hierzu wird wieder vom Pseudozufallszahlengenerator Gebrauch gemacht. Die Telegrammlänge wird an der entsprechende Stelle in das Telegramm geschrieben, damit auf der Seite des Rabbit Powercore 3800 das Telegrammende erkannt werden kann. Die Nutzdaten werden mit Zufallswerten belegt. Auf eine Berechnung der CRC-Prüfsumme wird verzichtet und die zwei dafür vorgesehenen Byte werden ebenfalls zufällig gefüllt. An die Stelle des Adressbyte wird der Wert eines 8-Bit-Zählers gesetzt, der mit jedem Telegramm inkrementiert wird. Auf diese Weise kann überprüft werden, ob die zu entwickelnde Software in der Lage ist, den empfangenen Datenstrom korrekt in Telegramme zu unterteilen. Die Telegramme sollen bei diesem Test zusätzlich über die Ethernet-Schnittstelle weitergeleitet werden. Ein entsprechendes Programm auf PC-Seite empfängt die Daten und prüft, ob diese vollständig angekommen sind. Ein entscheidender Aspekt bei diesem Test ist die Fähigkeit des Programms auf dem Rabbit Powercore 3800, sich möglichst schnell in einen laufenden Datenstrom einzusynchronisieren und die Telegramme korrekt zu unterteilen.

Ebenfalls ist zu testen, ob die Performance ausreichend ist, um die Telegramme zu empfangen, aufzuteilen und weiterzuschicken. Die Größe der Telegramme kann angepasst werden, indem die höherwertigen Bit fest vorgegeben werden und nur die niederwertigen Bit zufällig belegt werden. Somit kann ebenfalls getestet werden, wie sich unterschiedliche Telegrammgrößen auf die Übertragungssicherheit auswirken. Es ergibt sich, dass bei vielen kleinen Telegrammen das Weiterleiten länger dauert als der Zeitraum, in dem ein Telegramm empfangen wird. Somit kommt es unabhängig von der Größe des verwendeten Puffers zu einem Pufferüberlauf. In der Praxis werden aber meist große Messdatentelegramme empfangen, so dass dieses Problem durch einen ausreichend dimensionierten Puffer abgefangen werden kann.

Mit diesen Tests ist nun sichergestellt, dass die grundlegende Funktionalität der Übertragungsstrecke sowie der Software vorliegt und die weitere Entwicklung mit angebundenen SICK-Lasermesssystemen stattfinden kann.

4.3 Auswahl von UDP zur Datenübertragung

In diesem Abschnitt geht es um das Senden von Messdaten vom Rabbit Powercore 3800 zum Hostrechner. Optional sollen Kontrolldaten vom Hostrechner an den Rabbit Powercore 3800 gesendet werden.

Der Rabbit Powercore 3800 besitzt eine Ethernet-Buchse und die dazugehörige Entwicklungsumgebung Dynamic C eine Bibliothek mit TCP/IP-Stack. TCP/IP ist durch das Internet inzwischen zum Standard auf Ethernet-Verbindungen geworden. Appletalk und Novells IPX-Protokoll werden kaum noch genutzt. Die SICK-Lasermesssysteme verschicken Telegramme mit einer Länge von bis zu 732 Byte. Außer den Telegrammen werden ACK- und NACK-Byte versendet.

Das *Transmission Control Protocol* (TCP) ist ein zuverlässiges, verbindungsorientiertes Transportprotokoll. Verlorengegangene Pakete werden automatisch erneut übertragen. Dadurch ist das Protokoll zustandsbehaftet und es ergibt sich ein höherer Verwaltungsaufwand. Die Verbindung vom Lasermesssystem zum Rabbit 3000 basiert hingegen auf einzelnen Telegrammen, so dass ein verbindungsorientiertes Protokoll überdimensioniert erscheint.

Das *User Datagram Protocol* (UDP) — als zustandsloses Protokoll — eignet sich besser für die Übertragung von Telegrammen und ACK/NACK-Meldungen. Dabei passt ein ganzes Telegramm in ein UDP-Paket, auf Fragmentierung kann verzichtet werden. Im Hostrechner stehen die Telegramme in einzelnen Datenblöcken zur Verfügung, da der Rabbit 3000 den Datenstrom von den Lasermesssystemen bereits in einzelne Telegramme zerlegt. Bei Verwendung von TCP würde es einen Datenstrom von Telegrammen geben, welcher im Hostrechner erneut synchronisiert werden müsste, um Telegrammanfänge zu ermitteln. Diese wurden aber bereits vom Rabbit 3000 bestimmt. Beim Versand über TCP müsste das Zerlegen in einzelne Telegramme ein erneutes Mal durchgeführt werden. Der Aufwand wäre damit unnötigerweise größer.

Bei UDP kommen im Gegensatz zu TCP die Daten nicht zwingend in richtiger Reihenfolge an. UDP-Pakete können verschiedene Transportwege nehmen und sich gegenseitig überholen. Bei einer direkten Verbindung zwischen dem Powercore 3800 und dem Hostrechner kann das Problem nicht auftreten. Alle Pakete werden in der richtigen Reihenfolge abgeschickt und können sich, auf dem einen Weg, den es gibt, nicht überholen. Außerdem können bei UDP einzelne Pakete verloren gehen. Dies ist unwahrscheinlich, da es sich um eine Vollduplex-Verbindung handelt. Empfangsbestätigungen werden bei UDP im Gegensatz zu TCP nicht verschickt. Auch bei der Verbindung zum Lasermesssystem werden keine Empfangsbestätigungen verschickt, so dass das hier nicht von Nachteil ist.

4.3.1 Entscheidung für UDP-Multicast

Bei UDP kann zwischen Anycast, Multicast, Unicast und Broadcast unterschieden werden. Wenn von UDP die Rede ist, wird normalerweise von Unicast ausgegangen. Es wird dabei ein Paket an einen bestimmten Empfänger geschickt.

Anycast ist aus Sicht des Senders von Unicast nicht zu unterscheiden. Der Sender glaubt, ein UDP-Paket an einen bestimmten Empfänger zu schicken. In Wirklichkeit wird aufgrund des Routings das Paket an einen von mehreren möglichen Rechnern weitergeleitet, nicht notwendigerweise immer an den gleichen. Anycast wird zum Loadbalancing verwendet und um die Antwortzeiten durch kürzere Wege zu reduzieren. Die Root-Server des Domain Name System (DNS) nutzen Anycast teilweise dafür.

Beim Broadcast wird ein Paket an eine Broadcastadresse geschickt, so erhalten es alle Rechner aus dem Subnetz. Der Sender kann aber nur Broadcastadressen aus seinen eigenen Subnetzen als solche erkennen, da keine Informationen über die Struktur anderer Subnetze vorliegen. Auf Broadcastadressen wird heutzutage im Internet häufiger verzichtet, da viele Subnetze aus Sicherheitsgründen nur aus einer IP-Adresse bestehen. Dort ist die IP-Adresse des Rechners mit der Broadcastadresse identisch. In eigenen Netzwerken wird noch mit Broadcastadressen gearbeitet. Einen Spezialfall stellt die Broadcastadresse 255.255.255.255 dar. Bei ihr findet grundsätzlich kein Routing statt, die Pakete werden bei Ethernet direkt in einen Ethernet-Broadcast umgesetzt.

Bei Multicast handelt es sich um eine Mehrpunktverbindung. Es wird ein Paket an eine Benutzergruppe geschickt. Beim Broadcast ist die Benutzergruppe fest durch ein Subnetz vorgegeben. Beim Multicast dagegen können einzelne Rechner dieser Benutzergruppe beitreten, nur dann erhalten sie die Pakete der Multicast-Gruppe. Im IPv4[3] ist für Multicast der IP-Adressenbereich

[3] Internet Protocol Version 4: Ist noch der Standard für IP, wird durch IPv6 eines Tages abgelöst

von 224.0.0.0 bis 239.255.255.255[4] vorgesehen. Der Sender kann also anhand der Zieladresse erkennen, ob es sich um eine Multicast-Adresse handelt. Für den eigentlichen Versand hat das zunächst keine Auswirkungen. Multicast bietet im Gegensatz zu Broadcasts eine das lokale Netzwerk übergreifende Mehrpunktverbindung. Das hat zur Folge, dass Multicast-Pakete geroutet werden müssen, um alle Teilnehmer der Benutzergruppe zu erreichen. Eine Verwaltung der Multicast-Gruppe und der dazugehörigen Information findet nicht in den Sendern, sondern ausschließlich in den Routern statt. Die Empfänger können einer Multicast-Gruppe beitreten, indem sie ihre Mitgliedschaft mit einem IGMP-Paket bei ihrem Router anmelden. Die Router sind dann untereinander für den Austausch der Multicast-Pakete zuständig. Hat kein Rechner Interesse an den Multicast-Paketen, so werden sie auch von keinem Router weitergeleitet. Der Sender hat also kein Wissen darüber, an wie viele Empfänger seine Pakete gesendet werden.

Zum Verschicken von UDP-Paketen müssen diese in IP-Pakete verpackt werden. Um diese über Ethernet zu verschicken, muss eine Umsetzung der IP-Adressen in MAC-Adressen erfolgen. Die IP-Pakete werden dann in Ethernet-Frames eingebettet und an die ermittelte MAC-Adresse verschickt. Die Broadcastadresse 255.255.255.255 wird direkt in einen Ethernet-Broadcast übersetzt. Die Multicast-IP-Adressen werden direkt in MAC-Adressen umgesetzt, indem bei IPv4 die untersten 23 Bit der IP-Adresse und die MAC-Adresse 01-00-5e-00-00-00 zusammengesetzt werden. Dass sich hierbei mehrere Multicast-IP-Adressen eine MAC-Adresse teilen ist beabsichtigt. Für alle anderen IP-Adressen, die weder 255.255.255.255 noch eine Multicast-Adresse sind, muss die MAC-Adresse noch ermittelt werden.

Für die Umsetzung der IP-Adressen in MAC-Adressen ist das *address resolution protocol* (ARP) zuständig. Der Sender fragt in einem *ARP request-broadcast* nach der MAC-Adresse für eine IP-Adresse. Der Empfänger reagiert auf den Broadcast mit einem ARP-*reply*. Nun kann der Sender sein IP-Paket an den Empfänger in einem Ethernet-Frame verschicken. Damit nicht für jedes IP-Paket ein ARP-*request* verschickt und auf die Antwort gewartet werden muss, speichert der Sender in seiner ARP-Tabelle die Zuordnungen von IP- und MAC-Adressen zwischen.

Bei der Anbindung von Lasermesssystemen hat Multicast gegenüber Unicast den Vorteil, dass nicht nur eine einzelne Anwendung die Möglichkeit hat, die Lasermesswerte zu erhalten, sondern beliebig viele — auch Computerübergreifend. Ein weiterer Vorteil ist, dass keine MAC-Adressen mit ARP aufgelöst werden müssen. Sollte das Netzwerkkabel nicht stecken oder aber der Hostrechner nicht richtig konfiguriert sein, so käme es im Powercore 3800 zu Timeouts, wenn Unicast verwendet würde. Ein Verzicht auf ARP-*requests* führt damit zu einem etwas schnelleren und sichereren System. Weitere Informationen zu UDP, Multicast und ARP sind in [Tan03, Abschnitt 5.6, S. 327-361] zu finden.

4.3.2 UDP-Client und -Server für Linuxrechner

Zum Testen von UDP-Verbindungen bietet es sich an, kleine Kommandozeilenprogramme zum Senden und Empfangen von UDP-Paketen zu schreiben. Beim Senden muss zwischen Unicast und Multicast nicht unterschieden werden. In Listing 4.1 ist ein Beispielcode in C zum Senden von UDP-Paketen für Linuxrechner angegeben.

[4] Bei IPv6 sind alle IP-Adressen, die mit ff:: beginnen für Multicast vorgesehen.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main(int argc, char *argv[]) {
  struct addrinfo hints, *res_hinfo;
  const char *host = "224.0.0.23";
  const char *msg = "[empty message]";
  int s, r;

  if (argc > 1) host = argv[1];
  if (argc > 2) msg = argv[2];

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_DGRAM;
  hints.ai_protocol = IPPROTO_UDP;

  if ((r = getaddrinfo(host, "2222", &hints, &res_hinfo))) {
    fprintf(stderr, "ERROR: getaddrinfo(): %s\n", gai_strerror(r));
    return 1;
  }

  if ((s = socket(res_hinfo->ai_family,
                  res_hinfo->ai_socktype,
                  res_hinfo->ai_protocol)) < 0) {
    fprintf(stderr, "ERROR: socket(): %s\n", strerror(errno));
    return 2;
  }

  if (sendto(s, msg, strlen(msg), 0, res_hinfo->ai_addr,
             res_hinfo->ai_addrlen) > 0)
    printf("Message sent\n");
  else fprintf(stderr, "Error while sending data\n");

  freeaddrinfo(res_hinfo);
  close(s);

  return 0;
}
```

Listing 4.1: Beispiel zum Senden von UDP-Paketen

Die Funktion `getaddrinfo()` dient dem Auflösen von Rechnernamen und Services. Sie erzeugt eine *socket address structure* `addrinfo`. Diese kann direkt in `bind()`- und `connect()`-Aufrufen verwendet werden. Sie ist eine Zusammenfassung für die Funktionen `getipnodebyname()`, `getipnodebyaddr()`, `getservbyname()`, und `getservbyport()`. Weitere Informationen sind in

der *man page* `getaddrinfo(3)` zu finden. In diesem Beispiel wird nur `ai_addr` und die dazugehörige Länge `ai_addrlen` aus der Struktur als Empfangsadresse zum Versenden des UDP-Pakets verwendet. Der Systemaufruf `socket()` liefert einen neu generierten Socket zurück. Ein Socket ist ein Kommunikationsendpunkt. Die Funktion `sendto()` schickt über einen Socket ein Paket an einen Empfänger. Dessen Adresse wurde zuvor mit `getaddrinfo()` ermittelt.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <errno.h>
4  #include <unistd.h>
5  #include <sys/socket.h>
6  #include <sys/types.h>
7  #include <netinet/in.h>
8  #include <netdb.h>
9  #include <assert.h>
10 #include <arpa/inet.h>
11 #include <sys/ioctl.h>
12 #include <stdlib.h>
13 #include <net/if.h>
14
15 int main() {
16   int fd = -1, on = 1;
17   struct sockaddr_in sa;
18   struct ip_mreqn mreq;
19   unsigned char *buf = NULL;
20   size_t buflen = 0;
21   fd_set rfds, rfdsset;
22
23   if ((fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
24     fprintf(stderr, "Error: socket(): %s\n", strerror(errno));
25     goto fail;
26   }
27
28   setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
29
30   memset(&sa, 0, sizeof(sa));
31   sa.sin_family = AF_INET;
32   sa.sin_port = htons(2222);
33   sa.sin_addr.s_addr = INADDR_ANY;
34
35   if (bind(fd, (struct sockaddr*) &sa, sizeof(sa)) < 0) {
36     fprintf(stderr, "Error: bind(): %s\n", strerror(errno));
37     goto fail;
38   }
39
40   memset(&mreq, 0, sizeof(mreq));
41   mreq.imr_multiaddr.s_addr = inet_addr("224.0.0.23");
42   mreq.imr_address.s_addr = INADDR_ANY;
43   mreq.imr_ifindex = if_nametoindex("eth0");
44
45   if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
46                  &mreq, sizeof(mreq)) < 0) {
```

```
47      fprintf(stderr, "Error: setsockopt(): %s\n", strerror(errno));
48      goto fail;
49    }
50
51    FD_ZERO(&rfdsset);
52    FD_SET(fd, &rfdsset);
53
54    for(;;) {
55      int l;
56      ssize_t rl;
57
58      rfds = rfdsset;
59
60      if (select(fd+1, &rfds, NULL, NULL, NULL) < 0) {
61        fprintf(stderr, "Error: select(): %s\n", strerror(errno));
62        goto fail;
63      }
64
65      if (ioctl(fd, FIONREAD, &l) < 0) {
66        fprintf(stderr, "Error: ioctl(): %s\n", strerror(errno));
67        goto fail;
68      }
69
70      if (buflen < (size_t) l)
71        if (!(buf = realloc(buf, buflen = l+1))) {
72          fprintf(stderr, "Error: out of memory\n");
73          goto fail;
74        }
75
76      if ((rl = recv(fd, buf, buflen, 0)) < 0) {
77        fprintf(stderr, "Error: recv(): %s\n", strerror(errno));
78        goto fail;
79      }
80
81      assert(l == rl);
82
83      /* in buf ist jetzt das Telegramm enthalten */
84      if (buf[0] == 0x02) printf("Telegramm empfangen\n");
85    }
86
87   fail:
88    if (fd >= 0) close(fd);
89
90    return 1;
91  }
```

Listing 4.2: UDP-Serverbeispiel mit Multicast-Empfang

In dem Beispiel der UDP-Serveranwendung in Listing 4.2 wird als erstes ein Socket in Zeile 23 erzeugt. Bevor der Socket an einen Port gebunden wird, muss die Option `SO_REUSEADDR` gesetzt werden, damit nicht nur eine Anwendung die Daten bekommen kann, sondern mehrere Anwendungen auf einem Host. Das setzen Option `SO_REUSEADDR` hat mehrere Konsequenzen.

Wird ein Programm beendet, ohne den Socket zu schließen, so wird der Port vom Kernel in den Zustand `TIME_WAIT` versetzt. Er kann sofort wieder verwendet werden, wenn vor dem Binden des Sockets an einen Port die Option `SO_REUSEADDR` gesetzt wird. Für Multicast-Umgebungen gibt es die Möglichkeit, dass mehrere Anwendungen den selben Port benutzen, indem alle die Option `SO_REUSEADDR` setzen.

In Zeile 35 werden alle lokalen IP-Adressen mit dem Port 2222 an den Socket gebunden. Die Multicast-IP-Adresse aus Zeile 41 wird in Zeile 45 dem Socket hinzugefügt, so dass sowohl Unicast als auch Multicast-UDP-Pakete auf dem Port 2222 empfangen werden. Unicast-Pakete kommen dann jedoch nur bei einer Anwendung pro Host an. Es lässt sich nicht sagen bei welcher.

Der Programmcode in der `for`-Schleife ist zum Abarbeiten von den empfangenen Telegrammen/ UDP-Paketen. Die Aufrufe `select`(), `ioctl`() und `realloc`() in den Zeilen 60, 65 und 71 sind zum ressourcenschonenden Warten und Speicher allozieren. Das UDP-Paket wird in Zeile 76 abgeholt und in `buf` geschrieben. Eine Verarbeitung der Daten ist danach möglich.

4.3.3 Rabbit-Powercore-3800-UDP-Konfiguration

Um den Ethernet-Anschluss des Powercore 3800 benutzen zu können, muss dieser konfiguriert werden. Eine Konfiguration kann durch Definitionen zur Kompilierzeit über die `#define`-Direktive erfolgen oder zur Laufzeit über spezielle Funktionen. In Listing 4.3 ist in den ersten Zeilen eine Konfiguration mit der `#define`-Direktive zu sehen. Es ist auch möglich, den Powercore 3800 so zu konfigurieren, dass er beispielsweise über DHCP eine IP-Adresse zugewiesen bekommt. Als autark arbeitendes Gerät, wie es in dieser Arbeit entwickelt wird, soll der Prototyp ohne Fremdkonfiguration lauffähig sein.

Mit `#define TCPCONFIG` kann eine Standardeinstellung gewählt werden. Die Zahl null deaktiviert alle Standardeinstellungen. Mit `#define USE_ETHERNET` wird die Anzahl der Ethernet-Schnittstellen angegeben; der Powercore 3800 besitzt nur eine. Mit `IFCONFIG_*` kann dieses Netzwerkgerät aktiviert werden. `MAX_UDP_SOCKET_BUFFERS` gibt die Anzahl an Puffern an, die für UDP-Sockets zur Verfügung stehen. Der Standardwert von null ist in diesem Falle nicht ausreichend und muss auf eins gesetzt werden.

Standardmäßig ist die Ethernet-MTU[5] im Rabbit 3000 auf 600 gesetzt. Damit Telegramme nicht auf mehrere Ethernet-Pakete aufgeteilt werden müssen, bietet es sich an, sie mit `ETH_MTU` auf 1500 zu erhöhen. Wenn DNS und TCP nicht benötigt werden, sollten die dazugehörigen Codeteile mittels `#define DISABLE_DNS 1` und `#define DISABLE_TCP 1` deaktiviert werden.

Um die Funktionen der TCP/IP-Bibliothek zu verwenden, muss diese mit `#use "dcrtcp.lib"` eingebunden werden.

Die Funktion `sock_init()` übernimmt die Initialisierung der Netzwerktreiber und der TCP/IP-Bibliothek. Außerdem wird die zur Kompilierzeit festgelegte Konfiguration ausgeführt. Diese Funktion muss vor allen anderen TCP/IP-Funktionen verwendet werden.

Nachdem die Sockets initialisiert wurden, kann mit `udp_open(udp_Socket *s, word lport, longword remip, word port, dataHandler_t datahandler)` ein UDP-Socket geöffnet werden. Der `lport` gibt die lokale Portnummer an, die an den Socket gebunden werden soll. `remip` und

[5] *Maximum Transmission Unit*, bei Ethernet maximal 1500

`port` sorgen für eine Einschränkung der UDP-Pakete, die auf diesem Socket empfangen werden sollen. Sollen alle UDP-Pakete an die lokale Portnummer von allen Rechnern empfangen werden, so muss `remip` auf -1 (gleichbedeutend mit 255.255.255.255) gesetzt werden. Der Wert aus `port` wird dann ignoriert. Soll nur von einer IP-Adresse empfangen werden, so ist diese einzutragen. Im `datahandler` kann eine Callback-Funktion registriert werden, die bei ankommenden UDP-Paketen aufgerufen wird. Weitere Hinweise zum `datahandler` folgen weiter unten.

Die Funktion `udp_extopen()` ist eine *extended version* von `udp_open()`. Zusätzlich zum lokalen Port des UDP-Sockets kann noch ein Interface angegeben werden um mehrere Interfaces zu unterscheiden. Der Powercore 3800 besitzt jedoch nur ein Interface, so dass dieses Feature nicht benötigt wird. Außerdem gibt es die Möglichkeit, einen eigenen Speicherbereich aus dem *extended memory* anzugeben, in dem die UDP-Pakete vom Netzwerktreiber gespeichert werden. Auf die Puffer von `MAX_UDP_SOCKET_BUFFERS` kann dann verzichtet werden. `udp_waitopen()` unterscheidet sich von `udp_extopen()` nur dadurch, dass für die Auflösung von `remip` ein Timeout angegeben werden kann.

Damit die Sockets auf dem Powercore 3800 richtig funktionieren, muss regelmäßig die Funktion `tcp_tick(void *s)` aufgerufen werden. Aufgaben dieser Funktion sind das Beantworten von ARP-Requests, das Verschicken von ICMP-Paketen, das Speichern von eingehenden Paketen und das erneute Versenden verloren gegangener TCP-Pakete. Wird als Parameter `s` kein `NULL`, sondern ein Zeiger auf einen bestimmten Socket übergeben, liefert der Rückgabewert Informationen über den Zustand dieses Socket. Es wird empfohlen, die Funktion zehn Mal in der Sekunde aufzurufen. In der Regel funktionieren Anwendungen aber auch wenn `tcp_tick()` häufiger oder seltener aufgerufen wird.

Die Funktion `udp_recv(udp_Socket *s, char *buffer, int len)` dient dazu, auf die UDP-Pakete im Empfangspuffer zuzugreifen. Sie kopiert die Daten aus dem Empfangspuffer, der im *extended memory* liegt, in den übergebenen Speicherbereich (`char *buffer`) im *root-memory*-Bereich. `udp_recvfrom()` liefert zusätzlich die IP-Adresse und den Port vom Absender des UDP-Pakets.

Zum Verschicken von UDP-Paketen ist die Funktion `udp_send()` geeignet. Sie verschickt das UDP-Paket an die IP-Adresse und Port, die bei `udp_open()` für `remip` und `port` angegeben wurden. Wurde dort keine bestimmte IP-Adresse/Port ausgewählt oder soll das Paket an eine andere IP-Adresse/Port gehen, so muss `udp_sento()` mit Ziel-IP-Adresse und -Port verwendet werden.

Die mit `MAX_UDP_SOCKET_BUFFERS` „allozierten“ Speicherbereiche liegen im *extended memory*. Auch die bei `udp_extopen()` zu übergebenen Speicherbereiche müssen im *extended memory* liegen. Die UDP-Pakete werden deshalb zuerst immer dort gespeichert. Der *extended memory* hat absolute 20-Bit-Adressen im Gegensatz zu den „normalen“ Zeigern im *root memory*, die 16 Bit breit sind.

Es gibt zwei Varianten, auf die UDP-Daten aus dem *extended memory* zugreifen. Mit `udp_recv()` bzw. `udp_recvfrom()` werden durch Pollen die Daten in den *root memory* kopiert. Alternativ kann bei `udp_open()` und dessen Varianten eine Callback-Funktion registriert werden. Diese Funktion wird mit Zeigern auf die Daten im *extended memory* aufgerufen. Da auf Daten im *extended memory* nicht direkt zugegriffen werden kann, müssen die Daten zuerst mit `xmem2root()` in den *root-memory*-Bereich kopiert werden. Alternativ können einzelne Daten mit den Funktionen `xgetint()`, `xgetlong()`, `xgetfloat()`, ... abgefragt werden. Die Funktion `udp_xsendto()`

kann die Daten aus dem *extended memory* direkt per UDP verschicken, es ist dann ein Performancegewinn vorhanden. Die Callback-Funktion wird aus der Funktion `tcp_tick()` ausgelöst und unterscheidet sich damit nicht so stark vom Pollen mittels `udp_recv()`.

```
1  #define TCPCONFIG 0
2  #define USE_ETHERNET 1
3  #define IFCONFIG_ETH0 \
4          IFS_IPADDR, aton("134.100.13.229"), \
5          IFS_NETMASK, aton("255.255.255.0"), \
6          IFS_ROUTER_SET, aton("134.100.13.250"), \
7          IFS_UP
8  #define MAX_UDP_SOCKET_BUFFERS 1
9  #define DISABLE_DNS 1
10 #define DISABLE_TCP 1
11 #define ETH_MTU 1500
12
13 #memmap xmem
14 #use "dcrtcp.lib"
15
16 #define LOCAL_PORT   2222
17 #define REMOTE_IP    -1
18 #define REMOTE_PORT  0
19
20 udp_Socket sock;
21
22 int send_packet(char *buf, int length,
23                 longword remip, word remport) {
24   if (udp_sendto(&sock, buf, length, remip, remport) < 0) {
25     printf("udp_sendto() failed!\n");
26     sock_close(&sock);
27     if(!udp_open(&sock, LOCAL_PORT, REMOTE_IP, REMOTE_PORT, NULL)) {
28       printf("udp_open() failed!\n");
29       exit(0);
30     }
31   }
32   tcp_tick(NULL);
33   return 1;
34 }
35
36 int receive_packet(void) {
37   static char buf[128];
38   static char ip[16];
39   int length;
40   longword remip;
41   word remport;
42 #GLOBAL_INIT
43   { memset(buf, 0, sizeof(buf)); }
44
45   if (0 > (length = udp_recvfrom(&sock, buf, sizeof(buf),
46                                  &remip, &remport)))
47     return 0;
48
```

```
49    printf("UDP ip=%s port=%d length=%u\n",
50           inet_ntoa(ip, remip), remport, length);
51    send_packet(buf, length, remip, remport);
52
53    return 1;
54  }
55
56  void main() {
57    if (sock_init()) {
58      printf("sock_init() failed!\n");
59      exit(0);
60    }
61    if(!udp_open(&sock, LOCAL_PORT, REMOTE_IP, REMOTE_PORT, NULL)) {
62      printf("udp_open() failed!\n");
63      exit(0);
64    }
65    for (;;) {
66      tcp_tick(NULL);
67      receive_packet();
68    }
69  }
```

Listing 4.3: Beispielcode für Powercore 3800 UDP-Client und -Server

4.4 Zusammenfassung

Es wurde der Prototyp mit seinen Hardware-Komponenten vorgestellt. Er setzt sich aus dem Powercore 3800 und einer selbst entwickelten Platine zusammen. Um die Funktionsfähigkeit der seriellen Schnittstellen zu überprüfen, wurde mit Hilfe von VHDL ein Testgenerator entwickelt, der ein SICK-Lasermesssystem simuliert. Anschließend wurde hiermit die Ethernet-Verbindung zwischen dem Powercore 3800 und dem Hostrechner getestet. Es steht damit der Hardwareteil und die Schnittstellen-Ansteuerung eines Eingebetteten Systems zur Verfügung.

5 Softwareentwicklung

Nachdem im vorangegangenen Kapitel die Entwicklung und das Testen des Hardwareprototypen beschrieben wurde, geht es in diesem Kapitel um die Entwicklung der Software für den Rabbit 3000 und für den Hostrechner.

Zunächst wird der Ablauf des Programms auf dem Eingebetteten System vorgestellt. Dabei wird ein Überblick über die implementierten Funktionen und deren Abarbeitung gegeben. Anschließend wird die Datenstruktur vorgestellt, in der die eingehenden Telegramme der Lasermesssysteme gespeichert werden. Die Interrupt-Service-Routine, die die Telegramme in diese Datenstruktur schreibt, wird im dritten Abschnitt erläutert. Im darauf folgenden Teil wird der endliche Automat vorgestellt, der die Steuerung der Lasermesssysteme durch Auswertung der empfangenen Telegramme übernimmt. Die Vorverarbeitung, die im Rabbit 3000 stattfindet, wird im fünften Abschnitt beschrieben. Implementiert ist eine Minimalabstandsprüfung und ein Algorithmus zur Generierung einer Reflektormarkenliste. Die UDP-Datenpakete, die zwischen dem Rabbit 3000 und dem Hostrechner ausgetauscht werden, sind im sechsten Abschnitt spezifiziert. Die Anpassung der Software im Steuerrechner ist Thema des letzten Abschnitts.

5.1 Programmablauf des Eingebetteten Systems

Der Ablauf des Programms innerhalb des Rabbit 3000 ist Gegenstand dieses Abschnitts. Es wird in groben Zügen auf die Abläufe der einzelnen Funktionen eingegangen.

Die implementierten Funktionen lassen sich drei verschiedenen Programmteilen zuordnen:

- Initialisierung beim Programmstart
- Hauptblock des Programms
- Funktionen innerhalb der Interrupt-Service-Routine

Der Ablauf des Programms und die Funktionen der einzelnen Programmteile sind in Abbildung 5.1 dargestellt. Die Programmteile werden zunächst kurz vorgestellt. In den darauf folgenden Abschnitten wird dann ausführlich auf die implementierten Funktionen eingegangen.

5.1.1 Initialisierung beim Programmstart

Wie der Grafik zu entnehmen ist, beginnt das System nach dem Einschalten der Stromversorgung mit einigen Konfigurationsoperationen. Diese Operationen dienen dazu, die Netzwerkverbindung sowie die seriellen Schnittstellen betriebsbereit zu machen. Die Register der seriellen Schnittstellen werden wie in Abschnitt 3.2.8 erläutert konfiguriert. Dieser Vorgang ist erforderlich, damit die Kommunikation mit den Lasermesssystemen erfolgen kann. Die Netzwerkschnittstelle wird

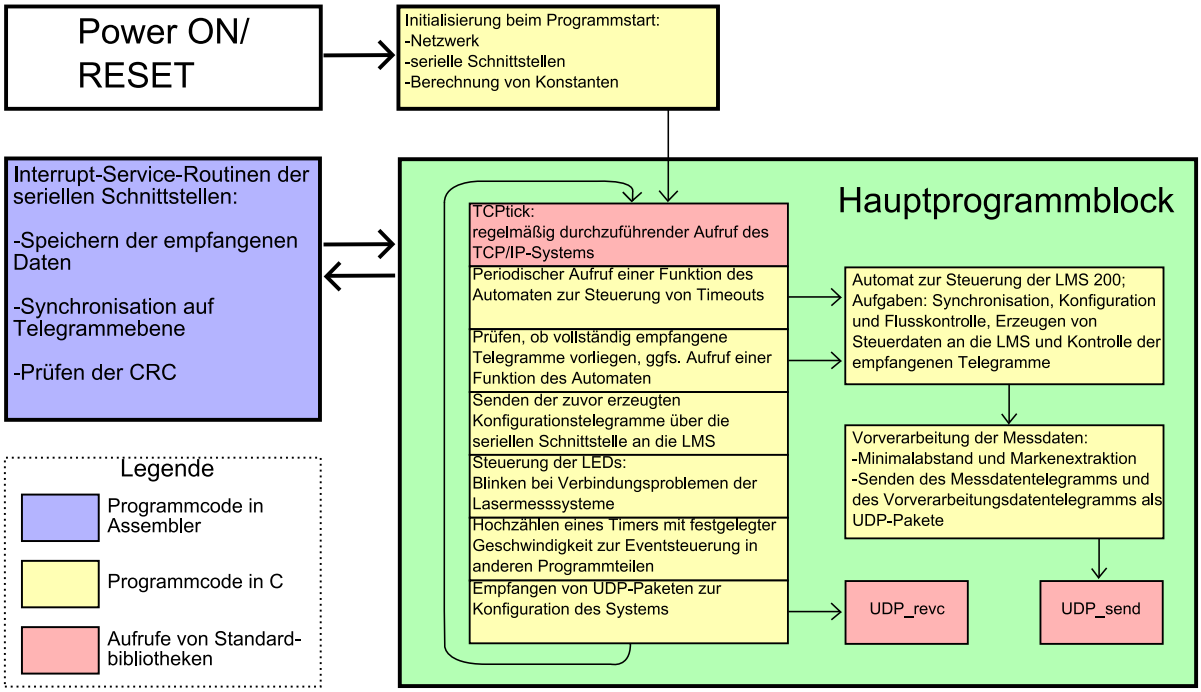


Abbildung 5.1: Programmablauf innerhalb des Rabbit 3000

über Aufrufe von vorgefertigten Funktionen initialisiert. Es wird unter anderem eine IP-Adresse vergeben, die das System im Netzwerk ansprechbar macht. Für die Rechenoperationen innerhalb der Vorverarbeitungsfunktionen sind diverse Konstanten zu berechnen. Im Programmablauf werden die Werte regelmäßig benötigt, so dass es aus Performancegründen sinnvoll ist, sie vorher zu berechnen.

5.1.2 Hauptblock des Programms

Der Hauptblock des Programms beinhaltet die Aktionen, die während der Laufzeit regelmäßig durchgeführt werden. Kern des Hauptblocks ist eine globale `for`-Schleife, die wiederholt durchlaufen wird. Aus dieser Programmschleife heraus werden alle Funktionen aufgerufen, die für den Betrieb erforderlich sind. Innerhalb der Schleife wird das in Kapitel 3.4 erläuterte Prinzip der Costates angewendet. Dadurch, dass Wartezeiten in einem Task dazu genutzt werden können, um Operationen in anderen Tasks auszuführen, können verschiedene Aufgaben auch ohne die Verwaltungsfunktion eines Betriebssystems parallel ausgeführt werden.

Die einzelnen Costates werden im Folgenden kurz vorgestellt:

TCPtick Zum Betrieb der Netzwerkschnittstelle muss in regelmäßigen Abständen diese Funktion aufgerufen werden. Es handelt sich um eine vorgefertigte Funktion aus dem mitgelieferten TCP/IP-Stack, die das TCP/IP-System des Rabbit Powercore 3800 steuert. Innerhalb dieser

Funktion wird der Netzwerkcontroller angesprochen, ARP- und Ping-Anfragen werden beantwortet und der Empfang von Telegrammen wird ermöglicht. Das Intervall zwischen zwei Aufrufen beträgt 10 ms, wobei eine Überschreitung dieser Zeit unkritisch ist.

Timeout-Aufruf der LMS-Automaten In regelmäßigen Zeitabständen müssen in den Automaten zur Steuerung der Lasermesssysteme Aufrufe durchgeführt werden, durch die überprüft wird, ob eine ausstehende Antwort der Lasermesssysteme eine maximal zulässige Zeitspanne überschreitet. In diesem Fall liegt ein Verbindungsproblem vor und es werden Maßnahmen zur Neusynchronisation veranlasst. Die Überprüfung erfolgt ungefähr alle 60 ms. Der Automat zur Steuerung der Lasermesssysteme wird ausführlich in Abschnitt 5.4 behandelt.

Prüfen auf empfangenes Telegramm In diesem Aufruf wird der Telegrammpuffer geprüft. Liegt ein vollständig empfangenes Telegramm vor, wird dieses verarbeitet. Bei der Verarbeitung erfolgt zunächst eine Auswertung im Automaten zur Steuerung der Lasermesssysteme. Abhängig von der Art des Telegramms und dem vorherigen Zustand erfolgt ein Zustandswechsel. Handelt es sich um ein Messdatentelegramm, erfolgt eine Vorverarbeitung der Messwerte. Schließlich erfolgt eine Weiterleitung des Telegramms durch einen Aufruf der vorgefertigten `UDP_send`-Routine.

Senden von Konfigurationstelegrammen an die LMS Wurden in dem Automaten zur Steuerung der LMS Telegramme zur Konfiguration der Systeme generiert und in die entsprechende Speicherstruktur geschrieben, erfolgt in diesem Costate das Versenden über die serielle Schnittstelle.

Steuerung der LEDs Wenn die Lasermesssysteme nicht synchronisiert sind, soll dieses durch Blinken der LEDs angezeigt werden, die normalerweise zur Anzeige von Distanzunterschreitungen dienen. Es erfolgt im Blinktakt eine Überprüfung der LMS-Automatenzustände und gegebenenfalls eine Steuerung der LEDs. Für den Fall, dass eine Synchronisation vorliegt, erfolgt das Ansteuern der LEDs hingegen in der Vorverarbeitungsfunktion.

Hochzählen eines Timers In regelmäßigen Abständen wird ein Timer hochgezählt. Dieser dient dazu, den Zeitpunkt eines Telegrammempfangs mit einem Zeitstempel zu versehen und innerhalb der Timeout-Funktion mit dem aktuellen Wert zu vergleichen. So kann bestimmt werden, ob ein ausstehendes Telegramm die nicht rechtzeitig eintrifft. Die Abweichung der Zählgeschwindigkeit durch die Costates soll vernachlässigt werden, da es für den Betrieb nicht relevant ist, ob Verbindungsprobleme einige Millisekunden später erkannt werden.

Empfang von Ethernet-Daten In dieser Costate-Anweisung wird überprüft, ob vom Hostrechner Daten an den Powercore gesendet wurden. Der Empfang erfolgt über den Aufruf der vorgefertigten Funktion `UDP_recv`. Abhängig von der Art des empfangenen Pakets wird die gewünschte Funktion ausgeführt. Die verschiedenen Funktionen sind in Abschnitt 5.6 erklärt. Beispielsweise kann der Reset des Systems veranlasst werden oder es können die Lasermesssysteme einzeln vorübergehend abgeschaltet werden.

5.1.3 Interrupt-Service-Routine der seriellen Schnittstellen

Alle Aktionen im Hauptblock werden unterbrochen, sobald ein Byte über die serielle Schnittstelle empfangen wird. Dieser Abbruch priorisiert das Empfangen der seriellen Daten vor allen anderen Aufgaben. Innerhalb der ISR wird der Datenstrom in Telegramme eingeteilt und in den Telegrammpuffer geschrieben. Die Funktionen der ISR werden in Abschnitt 5.3 ausführlich erläutert.

5.2 Datenstruktur

Die einzelnen von den Lasermesssystemen kommenden Byte müssen in einer Datenstruktur zwischengespeichert werden. Für jedes Lasermesssystem gibt es eine eigene Datenstruktur, da die einzelnen Byte unabhängig voneinander sind. Die meisten Interrupt-Service-Routinen speichern ankommende Byte in einem Ringpuffer. Es gibt zwei Offsets für diesen, der eine gibt die Position an, an die als nächstes geschrieben werden soll, der andere die Position, von der als nächstes gelesen werden soll. Der Ring hat normalerweise eine Größe von 2^n Byte. Das hat den Vorteil, dass die Offsets nur mit einer Maske und dem Bitoperator & (*and*)-verknüpft werden müssen, um einen Rundlauf zu erreichen.

Ringpuffer haben für den Anwendungszweck dieser Arbeit folgende Nachteile:

- Sie müssen mehrere Kilobyte groß sein, um mehrere Telegramme aufnehmen zu können. Das hat zur Folge, dass für die Offsets eine 8-Bit-Integer nicht mehr ausreicht. In einem 8-Bit-Mikrocontroller steigt dadurch der Aufwand für die Offsetberechnung.
- Ab und zu kommt es vor, dass ein Telegramm nicht in einem Stück im Speicher liegt. Das ist dann der Fall, wenn der Anfang am Ende des Ringpuffers liegt und der Rest des Telegramms am Anfang. Soll das Telegramm beispielsweise per UDP verschickt werden, so müssen die Daten in einem Stück im Speicher liegen.
- In Ringpuffern müssen die Daten in der Reihenfolge verarbeitet werden, in der sie hereingekommen sind (FIFO-Prinzip). Es ist nicht möglich, einzelne Telegramme länger aufzubewahren, ohne sie aus dem Ringpuffer heraus zu kopieren.

In Abbildung 5.2 ist die Datenstruktur skizziert, die im Rahmen dieser Arbeit entwickelt wurde. Jedes Telegramm liegt dabei in einem Stück vor, so dass einem Versand über UDP nichts im Wege steht. Die einzelnen Telegrammpuffer (in türkis dargestellt) haben eine Größe von 768 Byte. Sie fassen damit die größten Telegramme, die von SICK-Lasermesssystemen verschickt werden. Es handelt sich dabei um 732 Byte große Messdatentelegramme mit einer Winkelauflösung von 0,5 Grad. Es wurde ein Puffergröße von 768 statt 732 gewählt, weil 768 ganzzahlig durch 256 teilbar ist. Das hat den Vorteil, dass beim Prüfen der Länge im Telegrammheader ausschließlich das MSB des 16 Bit breiten Längenfeldes überprüft werden muss. Die Interrupt-Service-Routine, die das Telegramm byteweise einliest, kann so schneller entscheiden, ob das Telegramm zu groß für den Puffer ist. Es kann nämlich vorkommen, dass die Interrupt-Service-Routine beim Einsynchronisieren in den Telegrammdatenstrom in den Messdaten einen Headeranfang findet. Ohne Überprüfung würde es zu einem Pufferüberlauf kommen.

Die bis jetzt beschriebene Struktur löst zwei der oben genannten Probleme: Telegramme liegen nun in einem Stück vor und werden nicht am Ende des Ringpuffers umgebrochen. Außerdem kann

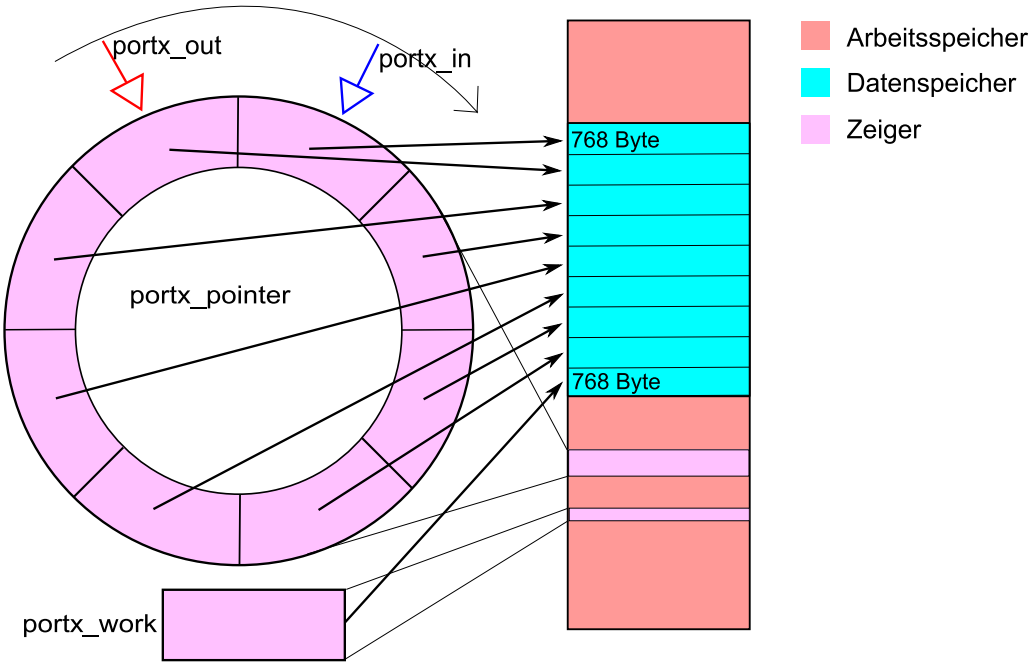


Abbildung 5.2: Datenstruktur für eingehende Telegramme

ein 8-Bit-Offset verwendet werden, um einen Speicherplatz zu indizieren. Die Speicheradresse lässt sich dann durch Multiplikation des Offsets mit der Pufferlänge von 768 berechnen.

Auch in dieser Datenstruktur besteht die Einschränkung, dass keine Telegramme längerfristig aufbewahrt werden können. Dieses Problem lässt sich lösen, indem der Offset nicht direkt angibt welcher Telegrammpuffer genommen werden soll, sondern einen Zeiger auf den Telegrammpuffer speichert. Durch einfaches Austauschen der Zeiger können so einzelne Telegrammpuffer verschoben werden, ohne dass die Daten kopiert werden müssen. In Abbildung 5.2 ist das Konstrukt mit den Zeigern auf der linken Seite dargestellt. Die Offsets für Schreiben und Lesen in die Datenstruktur sind `portx_in` und `portx_out`. Die Offsets verweisen auf Zeiger, die ihrerseits auf die eigentlichen Telegrammpuffer zeigen. Soll nun ein Telegramm länger aufbewahrt werden, so muss nur ein Zeiger aus dem Ring mit dem Zeiger aus `portx_work` ausgetauscht werden. Ab sofort ist das eine Telegramm aus dem Ring entfernt und wird nicht von neuen Telegrammen überschrieben. `portx_in` und `portx_out` sind beide vom Typ `char`, sie geben den Offset für den Zeigerring an. Der Zeigerring ist ein `char *portx_pointer[]`, also ein Zeiger auf ein Zeigerarray.

`portx_in` zeigt immer auf den als nächstes zu beschreibenden Puffer. Der Offset `portx_in` muss nur dann verändert werden, wenn ein Telegramm korrekt empfangen wurde. `portx_out` zeigt immer auf den zuletzt gelesenen Puffer. Das heißt, wenn ein weiteres Telegramm gelesen werden soll, muss der Offset weiter gesetzt werden. Daraus folgt, dass der Puffer genau dann leer ist, wenn `portx_in == portx_out+1` gilt, nämlich wenn als nächstes von dem Puffer gelesen werden soll, in den noch geschrieben wird. Maximal voll ist der Puffer, wenn `portx_in == portx_out` ist, denn dann soll in einen Bereich geschrieben werden, aus dem gerade gelesen wird.

Um ein Telegramm in die Datenstruktur aufzunehmen, muss als erstes überprüft werden ob der Puffer voll ist (`portx_in == portx_out`). Ist dies nicht der Fall, so können die Telegrammdaten Byte für Byte in den Puffer geschrieben werden, auf den `portx_in` zeigt. Nachdem das letzte Byte geschrieben ist, muss der Offset `portx_in` unter Verwendung der Bitmaske `BUFFER_MASK` inkrementiert werden.

Zum Lesen des nächsten Telegramms muss zuerst geprüft werden, ob überhaupt Telegramme im Puffer vorliegen. Das ist nur dann der Fall, wenn `portx_in != portx_out+1` ist. Da das Telegramm aus `portx_out` bereits gelesen wurde, muss `portx_out` inkrementiert werden. Hierbei ist zu beachten, dass `portx_out` immer mit der Bitmaske `BUFFER_MASK` zu verknüpfen ist. Die Telegrammdaten stehen nun in `portx_out` zur Verfügung.

Wenn ein Telegramm längere Zeit aufbewahrt werden soll, muss es aus dem Ring entfernt werden nachdem es versendet wurde. Zuerst muss wie beim Lesen geprüft werden, ob mindestens ein Telegramm im Puffer ist. Danach muss ebenfalls `portx_out` weiter gesetzt werden. Der Zeiger `portx_out`, der auf das bereits verschickte Telegramm zeigt wird mit `portx_work` ausgetauscht. Das Telegramm liegt jetzt in `portx_work`. In `portx_out` befindet sich jetzt das Telegramm, dass zuvor in `portx_work` ausgelagert war. Es wird beim nächsten Durchlauf automatisch überschrieben. Die Konstruktion mit `portx_work` hat die Einschränkung, dass nur ein Telegramm zur Zeit ausgelagert werden kann. Sollen mehrere zugleich ausgelagert werden, muss die Datenstruktur verändert werden. `portx_work` müsste dann ein Array von Zeigern sein.

5.3 Interrupt-Service-Routine

In diesem Abschnitt wird die Interrupt-Service-Routine (ISR) der Anwendungssoftware in dem Powercore 3800 vorgestellt. Zunächst werden Überlegungen zur maximal zulässigen Ausführungszeit der Interrupt-Service-Routine angestellt. Im darauf folgenden Teil werden die implementierten Funktionen erläutert, bei denen es sich um das Empfangen und Abspeichern von Daten, die Synchronisation auf Telegrammebene und das Prüfen der CRC-Prüfsumme handelt. Anschließend wird der Ablauf der ISR dargestellt und beschrieben, wie entsprechende Funktionen implementiert sind.

5.3.1 Ausführungszeit und Latenz

Die folgenden Ausführungen bauen auf den in Kapitel 3.2.5 beschriebenen Grundlagen der Interruptverarbeitung des Rabbit-3000-Mikroprozessors auf. Wird von den seriellen Schnittstellen ein Byte empfangen, so wird ein Interrupt ausgelöst und die ISR ausgeführt. Die Zeit zwischen Auftreten des Interrupts und dem Start der dazugehörigen ISR wird Interrupt-Latenz genannt. Je nach Art und Konfiguration der Komponente sind die Anforderungen an diese Latenzzeit mehr oder weniger kritisch.

Im Folgenden ist die maximal zulässige Latenz für eine serielle Schnittstelle bei 500 kBd errechnet: Da pro Byte das Startbit, 8 Datenbit und das Stoppbit übertragen werden, ergeben sich zehn übertragene Zeichen pro Byte. Es können also 50000 Byte pro Sekunde empfangen werden. Es besteht die Gefahr, dass ein empfangenes Byte verloren geht, wenn die ISR nicht abgearbeitet ist, bevor ein neues Byte eintrifft. Ein Byte muss also in 20 μs verarbeitet sein, das heißt,

Ausführungszeit einer ISR in Takten	Interrupt-Latenz in μs	maximal zul. Interrupt-Latenz in μs	Prozessor-auslastung in %
50	1,0	19,0	9,7
100	2,0	18,0	19,4
200	3,9	16,1	38,8
300	5,9	14,1	58,1
400	7,8	12,2	77,5
500	9,7	10,3	96,9
600	11,7	8,3	116,3

Tabelle 5.1: Interrupt-Latenz in Abhängigkeit von der Ausführungszeit der ISR

dass die maximale Latenz zuzüglich der Ausführungszeit der ISR 20 μs nicht überschreiten darf. Angenommen eine ISR benötigt zum Verarbeiten eines Byte 200 Taktzyklen, so ergibt sich bei einem Prozessortakt von 51,6 MHz eine Ausführungszeit von ca. 3,9 μs. In diesem Fall sollte sichergestellt werden, dass die maximale Latenz für die Ausführung der ISR nicht über 16 μs liegt.

Eine solch niedrige Interrupt-Latenz kann nur garantiert werden, wenn die Interrupts der seriellen Schnittstelle gegenüber anderen Interrupts bevorzugt behandelt werden. In diesem Fall wäre die Interrupt-Latenz einer seriellen Schnittstelle nahe null, da jeder andere Programmteil und jede andere ISR einfach abgebrochen werden würde.

Da zwei gleichberechtigte serielle Schnittstellen betrieben werden, müssen noch die Auswirkungen dieser Konfiguration auf die Interrupt-Latenz untersucht werden. Für weitere Berechnungen werden folgende Bedingungen angenommen: Die zwei seriellen Schnittstellen werden beide auf der gleichen Priorität betrieben, die größer ist als die Priorität aller anderen auftretenden Interrupts.

Es ergibt sich für das genannte Beispiel mit 200 Taktzyklen Ausführungszeit pro ISR als maximal im Betrieb auftretende Interrupt-Latenz 3,9 μs. Dies liegt daran, dass jede andere ISR einfach abgebrochen werden kann, nicht aber die ISR der zweiten seriellen Schnittstelle. Bis zur Beendigung dieser können die genannten 3,9 μs vergehen.

Bei anderen Ausführungszeiten der ISR ändern sich die Latenzzeiten sowie die maximal zulässige Latenz. In der Tabelle 5.1 sind diese Zeiten für einige Ausführungszeiten zusammengestellt. Zusätzlich ist noch die Systemauslastung errechnet. Diese ergibt sich aus der doppelten Ausführungszeit einer ISR geteilt durch 20 μs, beruhend auf der Annahme, dass pro 20 μs 2 Byte verarbeitet werden müssen.[1] Wie in der Tabelle zu erkennen ist, überschreitet die tatsächliche Interrupt-Latenz die maximal zulässige, wenn die ISR etwas über 500 Takte zur Ausführung benötigt. Ebenfalls würde dann die Prozessorauslastung rechnerisch auf über 100 % steigen.

In diesen Berechnungen ist noch nicht berücksichtigt, dass die seriellen Schnittstellen E und F einen 4-Byte-Empfangspuffer haben. Eine temporär erhöhte Interrupt-Latenz würde also keinen Datenverlust hervorrufen, jedoch würde schon eine etwas länger andauernde Erhöhung der

[1] in der Praxis wird nicht durchgehend gesendet

Latenz zum Datenverlust führen. Ebenfalls berücksichtigen diese Rechnungen noch nicht die Tatsache, dass pro Lasermesssystem im Durchschnitt nur 27,5 kByte/s an Daten eintreffen.

Es muss zwischen Funktionsumfang der ISR und der daraus resultierenden Prozessorauslastung abgewogen werden. Unter der Bedingung, dass ansonsten nicht viele rechenintensive Aufgaben ausgeführt werden und dass die Schnittstellen nicht permanent empfangen, erscheint eine Ausführungszeit bis ca. 300 Takte als angemessen für die ISR der seriellen Schnittstellen. In [R3000UM] ist eine ISR für serielle Schnittstellen beschrieben, die zum Empfangen und Abspeichern 117 Takte benötigt. Es bleibt bei der Entwicklung einer ISR also noch genug Potential für die Implementierung zusätzlicher Funktionen.

5.3.2 Funktionsumfang

Dieser Abschnitt befasst sich mit den implementierten Funktionen der ISR:

- Speichern von empfangenen Daten
- Synchronisation auf Telegrammebene
- Prüfen der CRC-Prüfsumme der Telegramme

Ziel der Implementierung zusätzlicher Funktionalität in der ISR ist es, die empfangenen Daten später telegrammweise in einem Stück im Speicher liegen zu haben, damit die Routine für das Versenden von UDP-Paketen die Daten problemlos verarbeiten kann. Ein zusätzliches Umkopieren der Daten soll vermieden werden, da dieses zusätzliche Rechenzeit verursachen würde. Es ist vom Gesamtaufwand her günstiger, die empfangenen Daten gleich an den Speicherort zu schreiben, an dem sie benötigt werden.

Wie im vorangegangenen Abschnitt 5.2 ausgeführt wurde, erfolgt die Speicherung der empfangenen Daten in einem Ringpuffer für die Telegramme. Ausgehend von der aktuellen Schreibposition und dem momentan zu beschreibenden Feld des Puffers muss die Speicheradresse ermittelt werden, in die das empfangene Byte abgelegt wird. Die Schreibposition wird bei jedem Schreibvorgang inkrementiert. Hierbei handelt es sich um einen Wert, der den aktuellen Versatz zur im Ringpuffer gespeicherten Basisadresse darstellt. Maximal kann dieser Wert die Größe eines Telegramms erreichen. Ist ein Telegramm komplett geschrieben, wird das nächste Feld im Ringpuffer zum Beschreiben ausgewählt.

Die Telegramme der Lasermesssysteme haben einen festgelegten Aufbau. Dieser ist ausführlich in Abschnitt 2.2.3 beschrieben. Auf der Seite des Empfängers ist zunächst das Startbyte und das Adressbyte auszuwerten und anschließend das LSB und das MSB der Telegrammlänge zu speichern. Anhand dieser Länge stellt die Empfängerseite fest, wie viele der nächsten empfangenen Byte zu dem aktuellen Telegramm gehören. Ist die entsprechende Anzahl an Byte empfangen, wird das Feld des Ringpuffers weitergesetzt und das nächste Telegramm in den dort referenzierten Speicherbereich geschrieben.

Die letzten beiden Byte eines Telegramms beinhalten eine CRC-Prüfsumme. Diese wird in der bisherigen Anbindung innerhalb des Hostrechners überprüft. Im Rahmen der Implementierung von Vorverarbeitungsfunktionen ist die Überprüfung der CRC-Prüfsumme in die ISR integriert. Falls es zu Übertragungsfehlern gekommen ist, können die defekten Telegramme erkannt und gelöscht werden. Es bestünde auch die Möglichkeit der Überprüfung der CRC außerhalb der

ISR, beispielsweise vor dem Versenden. Folgende Gesichtspunkte sprechen aber für die Implementierung dieser Funktionalität innerhalb der ISR:

- Verfügbarkeit von Algorithmen zur sukzessiven Berechnung der Prüfsumme
- Lastglättung, da sich der Rechenaufwand über die Empfangszeit des gesamten Telegramms verteilt
- erweiterte Möglichkeiten bei der Sicherung der Synchronisation (vgl. kommenden Abschnitt)

In [SickTL, S. 104f] ist ein Programmbeispiel in Assembler angegeben, wie die CRC-Prüfsumme der SICK LMS 200 sukzessive, also byteweise berechnet werden kann. Bei jeder Operation wird ein 16-Bit breites Zwischenergebnis generiert und in die nächste Operation mit einbezogen. Nach dem letzten empfangenen Datenbyte ist die CRC-Prüfsumme auf Empfängerseite fertig berechnet. Nun können die beiden noch folgenden CRC-Byte mit den zuvor berechneten verglichen werden. Kam es aufgrund eines Übertragungsfehlers zu einem veränderten oder ausgelassenen Byte, stimmen die errechnete und die empfangene Prüfsumme nicht überein. Der theoretisch denkbare Fall, dass aufgrund mehrerer Fehler die Prüfsumme trotz fehlerhafter Übertragung stimmt, kann nahezu ausgeschlossen werden. Eine weitere Möglichkeit bei der Auswertung der CRC besteht darin, die letzten beiden empfangenen CRC-Byte ebenfalls nach dem Algorithmus wie die vorherigen Byte zu verarbeiten. Wenn am Ende das Ergebnis gleich null ist, war die Übertragung fehlerfrei. In der ISR sollen jedoch die zwei Byte der Prüfsumme verglichen werden, da nur so eine frühe Erkennung von Übertragungsfehlern möglich ist.

Sicherung der Synchronisation

In bestimmten Situationen müssen spezielle Maßnahmen getroffen werden, damit die Datenintegrität gesichert bleibt und die Synchronisation schnellstmöglich wiederhergestellt wird:

- Synchronisation in eine laufende Übertragung (Systemstart oder Herstellen einer Verbindung zu einem LMS im Echtzeitmodus während der Laufzeit)
- Verlust oder Manipulation eines Byte durch Übertragungsfehler
- Byteverlust durch zu spätes Ausführen der ISR

Hierbei wird die Kenntnis über die Struktur der Telegramme der Lasermesssysteme ausgenutzt. Anhand folgender Kriterien kann innerhalb der ISR überprüft werden, ob die Synchronisation korrekt ist:

- Das Startbyte wird empfangen.
- Das Adressbyte ist korrekt.
- Das MSB der Länge ist kleiner als drei.
- Die CRC-Prüfsumme ist korrekt.

Zur Sicherheit müssen alle Bedingungen überprüft werden, damit eine fehlerhafte Synchronisation und damit falsche Messdatentelegramme nahezu ausgeschlossen werden können. Wenn in einen laufenden Datenstrom einsynchronisiert wird, wird in der Regel in den über 700 Nutzdatenbyte eines mit dem Inhalt 0x02 zu finden sein. Innerhalb der ISR wird dann fälschlicherweise davon ausgegangen, es handele sich um einen Telegrammanfang. Um diesen Fehler zu erkennen, wird zusätzlich das zweite Byte der Übertragung geprüft. Handelt es sich um das Adressbyte 0x80, wird weiterhin davon ausgegangen, dass die Synchronisation erfolgreich war. Im

Folgenden wird näherungsweise die Wahrscheinlichkeit berechnet, dass die Sequenz (0x02,0x80) in einer zufälligen Sequenz von etwa 700 Byte Nutzdaten zu finden ist. Diese Rechnung soll nicht einen exakten Wert liefern, sondern die Abschätzung der Größenordnung ermöglichen:

Die Berechnung erfolgt über die Berechnung der Wahrscheinlichkeit des Gegenereignisses, also dass diese Sequenz nicht vorkommt. Es gibt 699 Positionen, die dieses Muster aufweisen können. Bei Gleichverteilung der einzelnen Byte ist die Wahrscheinlichkeit, dass diese Sequenz vorliegt, $\frac{1}{256^2}$ bei jeder der Positionen. Die Wahrscheinlichkeit des Gegenereignisses ergibt sich zu:

$$\left(\frac{65535}{65536}\right)^{699} = 0,99$$

Etwa 1 % aller Telegramme enthalten diese Sequenz, führen also potentiell zu einer fehlerhaften Synchronisation. Es müssen daher weitere Maßnahmen getroffen werden.

Die Prüfung des MSB der Länge beruht auf folgender Annahme: Die größten Telegramme, die im Betrieb von den Lasermesssystemen empfangen werden, sind die Messdatentelegramme mit einer Länge von 732 Byte. Da das MSB bei der Berechnung der Gesamtlänge mit dem Faktor 256 multipliziert wird, darf bei einem korrekt empfangenen Telegramm der Wert zwei nicht überschritten werden.

Die Prüfung hat noch eine zusätzliche Aufgabe: Sollte eine Fehlsynchronisierung vorliegen oder kommt es aufgrund eines Übertragungsfehler beim MSB zu einem Wert größer als zwei, käme es zu einem Speicherüberlauf. Dieser ereignet sich, weil wegen der überhöhten Länge zu viele Byte dem Telegramm zugeordnet und linear im Speicher abgelegt werden. Es werden andere Daten überschrieben, was zum Absturz des Systems führen kann.

Sollte nach Prüfung des MSB der Länge dennoch eine Fehlsynchronisation vorliegen, bleibt diese ohne Auswirkungen auf die Systemstabilität, da für Telegramme bis 768 Byte Speicherplatz reserviert ist. Das Restrisiko einer Fehlsynchronisation sinkt durch die Prüfung nochmal um den Faktor $\frac{3}{256}$.

Nach erfolgreicher Verifikation der CRC-Prüfsumme kann von einem korrekt empfangenen Telegramm ausgegangen werden. Das Telegramm wird dann zur Weiterverarbeitung freigegeben und das folgende Telegramm in dem nächsten Feld des Ringpuffers gespeichert.

Schnelle Neusynchronisation

Sobald in einer der Überprüfungen ein Fehler festgestellt wird, werden bisher empfangene Daten des aktuellen Telegramms verworfen. Innerhalb der ISR wird auf das nächste potentielle Startbyte gewartet. Eine Erweiterung der Testfunktionen soll die Synchronisation noch verbessern. Wenn eine Prüfung fehlschlägt, wird anschließend getestet, ob es sich bei diesem Byte um das nächste Startbyte handeln könnte. Falls bei der Übertragung eines Telegramms durch Störungen ein oder zwei Byte verloren gehen, wird das Startbyte des nächsten Telegramms von der Routine als Teil der Prüfsumme betrachtet. Es kann mit hoher Wahrscheinlichkeit bereits beim Vergleich des ersten Teils der Prüfsumme erkannt werden, dass diese nicht korrekt ist. Wird direkt anschließend überprüft, ob es sich bereits um einen neuen Telegrammanfang handelt, bleibt das folgende Telegramm unversehrt. Außerdem kann bei der Einsynchronisierung in einen laufenden Datenstrom die Wahrscheinlichkeit verringert werden, dass durch eine ungünstige Konstellation der Daten ein Startbyte „übersprungen“ wird.

STX	ADR	LENGTH		CMD	DATA	STATUS	CRC	
1	1	2		1	LENGTH-2	1	2	
state0	state1	state2	state3	state4			state5	state6

Abbildung 5.3: Beziehung von Automatenzustand und Telegrammposition

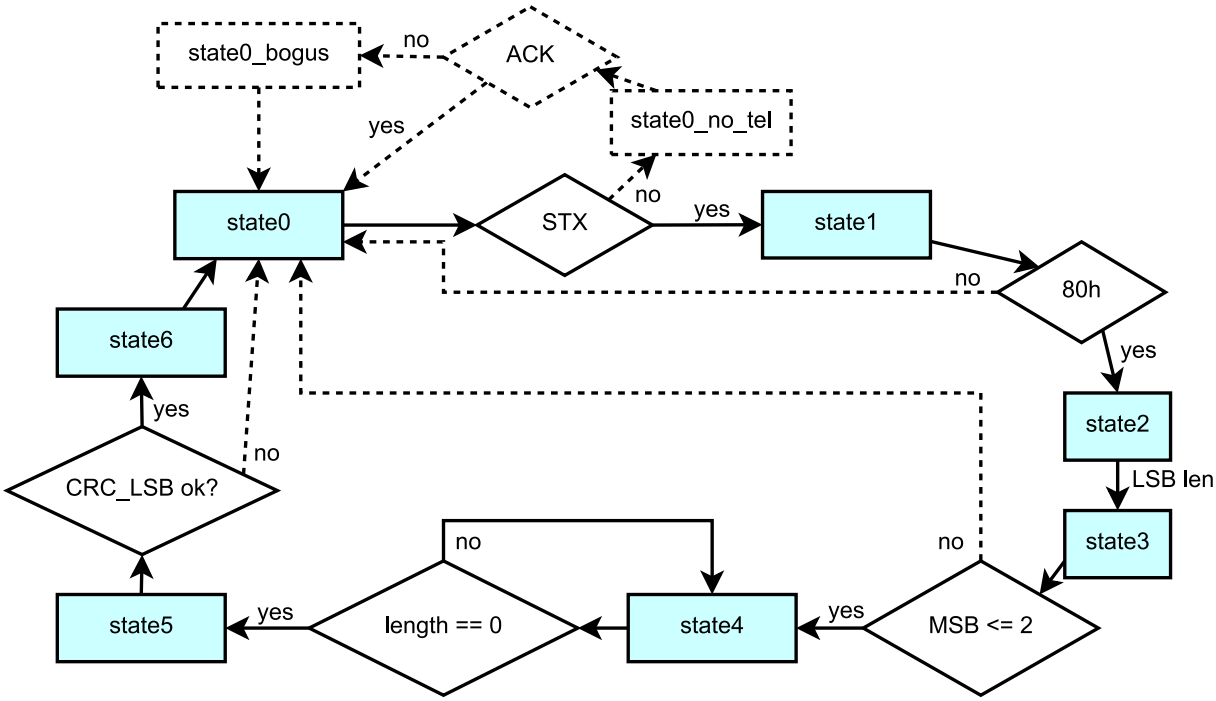


Abbildung 5.4: Ablaufdiagramm des ISR-Automaten

5.3.3 Implementierung

Um die beschriebene Funktionalität zu implementieren, wird innerhalb der ISR ein Automat konstruiert. Der Zustand, in dem sich der Automat jeweils befindet, ist abhängig von dem Byte, das als nächstes erwartet wird. Dieses Verhalten soll an zwei Grafiken näher erläutert werden.

In Abbildung 5.3 ist der Aufbau der Telegramme der SICK LMS 200 den Zuständen des Automaten zugeordnet. Der Automat besitzt sieben Zustände, die mit „state0“ bis „state6“ bezeichnet sind. Jeder dieser Zustände hat einen anderen Funktionsablauf. In den rot unterlegten Zuständen findet eine Prüfung der zuvor beschriebenen Bedingungen statt. Die gesamten Nutzdaten, die in der Darstellung des Telegramms gelb dargestellt sind, können innerhalb eines Zustandes abgearbeitet werden, da alle diese Daten auf die gleiche Art verarbeitet werden.

Der Ablauf der Telegrammverarbeitung im realisierten Automaten ist in Abbildung 5.4 dargestellt. Startzustand ist der „state0“. Dieser wird eingenommen, wenn auf den Anfang eines Telegramms gewartet wird, also auch nach aufgetretenen Fehlern. Beim empfangenen Byte wird geprüft, ob dieses ein Startbyte darstellt. Ist dieses nicht der Fall, erfolgt noch eine Prüfung, ob es sich um ein *Acknowledge* handelt. Dieses ist ein Byte, das die Lasermesssysteme schicken,

nachdem sie ein Telegramm empfangen haben. Das *Acknowledge* wird nicht weiter beachtet und es wird weiterhin auf den Anfang eines Telegramms gewartet. Handelt es sich weder um ein Startbyte noch um ein *Acknowledge*, werden die Daten verworfen und ein Zähler erhöht. Dieses erfolgt im Unterzustand „state0_bogus“, der im Gegensatz zu den Hauptzuständen nach dem Erreichen noch innerhalb der Ausführung der ISR wieder verlassen wird. Der Zähler dient der Statistik, so dass Auswertungen bezüglich der Empfangssicherheit gemacht werden können. Wurde das Startbyte empfangen, wird noch geprüft, ob der Telegrammpuffer vollgelaufen ist. Ist der Telegrammpuffer vollgelaufen, kann das aktuelle Telegramm nicht geschrieben werden, weil an diesem Speicherort im Moment noch ein altes Telegramm verarbeitet wird. In diesem Fall wird das aktuelle Telegramm verworfen und es wird ein eigens für dieses Ereignis implementierter Zähler hochgezählt. Dieser dient zur Auswertung des Programmablaufs. Somit kann im Rahmen der Entwicklungsarbeit die Vorverarbeitung so angepasst werden, dass es nicht zu diesem Ereignis kommt. Falls im Telegrammpuffer noch Speicherplatz für das Telegramm vorhanden ist, wird das Startbyte abgespeichert und es wird in den „state1“ gewechselt.

Im „state1“ wird auf das Adressbyte gewartet. Wenn dieses empfangen wird, erfolgt die Speicherung und der Wechsel in „state2“. Wird ein anderes Byte empfangen, so wird in den „state0“ zurückgekehrt, wobei die Aktionen dieses Zustandes ebenfalls ausgeführt werden, um eine schnelle Neusynchronisation zu ermöglichen.

In „state2“ wird das LSB der Längenangabe empfangen. Dieses wird lediglich abgespeichert und nicht ausgewertet, da zur Auswertung der Wert des MSB der Länge benötigt wird, der erst im folgenden Byte empfangen wird. Es folgt ein Wechsel in „state3“.

Hier erfolgt die Auswertung des MSB der Längenangabe. Entspricht dieses den genannten Bedingungen, so wird es abgespeichert und eine zusätzliche Variable mit der Länge der Nutzdaten errechnet. Falls sich aus dem MSB ein zu langes Telegramm ergeben würde, so wird von einem Fehler ausgegangen und in „state0“ gewechselt.

Wenn die Überprüfung erfolgreich war, wird in „state4“ mit dem Empfangen der Nutzdaten begonnen. Dabei werden die Daten solange ungeprüft im Telegrammpuffer abgespeichert und der Wert der Telegrammlänge dekrementiert, bis dieser Wert null erreicht hat. Dieser Zustand wird im Gegensatz zu allen anderen Zuständen bei dem Empfangen von Telegrammen mehrfach eingenommen. Anschließend wird in den „state5“ gewechselt.

Im „state5“ erfolgt die Prüfung des LSB der CRC-Prüfsumme. Bei Erfolg wird in „state6“ weitergearbeitet, bei Misserfolg in „state0“.

Das MSB der CRC-Prüfsumme des Telegramms wird in „state6“ kontrolliert. Wenn dieses korrekt ist, wird das aktuell empfangene Telegramm zur Weiterverarbeitung freigegeben, indem der im Telegrammpuffer zu schreibende Bereich gewechselt wird. Bei einer fehlerhaften Prüfsumme wird das aktuelle Telegramm verworfen. Unabhängig von der vorangegangenen Aktion wird in den „state0“ gewechselt.

Zur exakten Bestimmbarkeit der Ausführungszeit und aus Effizienzgründen ist die Routine komplett in Assembler geschrieben. Vom Grundaufbau entspricht die ISR der in Kapitel 3.2.8 vorgestellten ISR. Die Trennung zwischen Sende- und Empfangsinterrupts wurde auf die dort beschriebene Art vorgenommen.

Für die Implementierung des Automaten ist ein Konstrukt realisiert, das diese Funktionalität mit wenigen Operationen erreicht. Die Abläufe der einzelnen Zustände sind jeweils unter einer eigenen Speicheradresse abgelegt. Innerhalb der ISR erfolgt je nach aktuellem Zustand ein Sprung zu der entsprechenden Adresse. Der aktuelle Zustand ist dabei in einer 16-Bit-Variablen direkt als Sprungadresse gespeichert, so dass das Einnehmen eines Zustandes zu Beginn der ISR nur das Laden der Variablen und einen Sprung zu dieser Adresse erfordert. Diese Implementierung hat deutliche Vorteile gegenüber einer Struktur, die den Zuständen Nummern zuweist, diese Nummern beim Start der ISR prüft, und je nach Nummer einen Sprung zu dem Codeteil durchführt. Hier wären viele Vergleiche notwendig.

Beim Setzen eines neuen Zustandes wird die entsprechende Speicheradresse des neuen Zustandes in die Variable geschrieben. Die Programmierung dieses Vorganges erfolgt über sogenannte *lables*, also natürlichsprachliche Bezeichner, die zur Kompilierzeit durch die entsprechenden Adressen ersetzt werden.

Der kommentierte Quellcode der ISR ist im Verzeichnis `/quellcode/rabbit` auf der beiliegenden CD zu finden. Dort ist auch die Ausführungszeit der einzelnen Befehle mit aufgeführt. Die maximale Zeit, die zum Ausführen der ISR benötigt wird, beträgt 279 Takte. Also kann die gewünschte Funktionalität realisiert werden, ohne das festgelegte Limit zu überschreiten.

5.4 Synchronisationsautomat

Die Lasermesssysteme lassen sich mit Telegrammen steuern. Es können Parameter abgefragt und konfiguriert werden, sowie Messdaten vom Lasermesssystem empfangen werden. Die Steuerung dafür ist mit einem endlichen Automaten umgesetzt. Dieser endliche Automat empfängt Telegramme mit Hilfe der Interrupt-Service-Routine aus dem vorherigen Abschnitt und sendet Telegramme an das Lasermesssystem über einen Ausgabepuffer. Der Zustand des endlichen Automaten ist in `lms_state` in der Struktur `struct fsm` gespeichert. In Abbildung 5.5 ist der endliche Automat mit seinen Zuständen und den möglichen Transitionen abgebildet. Zum besseren Verständnis sind die Zustände eines Aufgabenbereichs mit runden Boxen farblich zusammengefasst. `IDLE` ist der Startzustand des endlichen Automaten. Die Zustände, die auf `wait` enden, sind Hilfszustände. Sie werden eingenommen, wenn eine Telegrammantwort vom Lasermesssystem erwartet wird. Kommt die Antwort nicht innerhalb der geforderten Zeit an, erfolgt ein Wechsel des Zustandes entlang der mit *timeout* beschrifteten Kanten.

Als Erstes muss überprüft werden, ob ein Lasermesssystem an der seriellen Schnittstelle vorhanden ist und mit welcher Baudrate es sendet. Die Lasermesssystem unterstützen vier unterschiedliche Baudraten: 9.600, 19.200, 38.400 und 500.000 Baud (*high-speed*-Modus). Unabhängig davon kann sich das Lasermesssystem im Echtzeitmodus befinden: Es sendet kontinuierlich Messdaten. Welche Art von Messdaten spielt für die Synchronisation dabei keine Rolle. Die Synchronisation durchläuft folgende Schritte:

1. Dem Lasermesssystem wird mitgeteilt, dass es keine kontinuierlichen Daten senden soll. Genaugenommen wird ein Telegramm *output measured value on request only* an das Lasermesssystem gesendet und eine Sekunde gewartet. (Zustände `500Kpre`, `9K6pre`, `19K2pre`, `38K4pre` und ihre Wartezustände)

2. Bevor weitere Kommunikation mit dem Lasermesssystem betrieben wird, löscht ein Aufruf der Funktion `porte_flush()` bzw. `portf_flush()` den Eingangspuffer, da sich ein unvollständiger Telegramm darin befinden kann.

3. Um zu überprüfen, ob das Lasermesssystem mit der aktuellen Baudrate des endlichen Automaten überein stimmt, wird der Status des Lasermesssystems abgefragt. Dazu wird ein *status request* Telegramm verschickt. (Zustände `500K`, `9K6`, `19K2`, `38K4` und ihre Wartezustände)

4. Wenn keine Statusantwort vom Lasermesssystem empfangen wurde, so muss mit einer anderen Baudrate von vorne begonnen werden. Ansonsten wird der Zustand `SYNC` eingenommen.

Nachdem die Schritte erfolgreich durchgeführt wurden, ist das Lasermesssystem erkannt worden, egal ob es gerade neu eingeschaltet wurde oder bereits lief.

Nach der Synchronisation folgt die Konfiguration (hellblauer Bereich) des Lasermesssystems, zunächst wird die Baudrate auf 500 kBd hochgesetzt (Zustand `SYNC`). Dazu wird ein *switch-operation-mode*-Telegramm verschickt. Die Antwort auf die Geschwindigkeitsanforderung wird vor der Geschwindigkeitsanpassung gesendet, deshalb wird danach mit einem *status request* (Zustand `CHECK`) überprüft wird ob, das Umschalten der Baudrate erfolgreich war. Das Lasermesssystem befindet sich im Zustand `WORK`. Ab diesem Zeitpunkt steht das Lasermesssystem im *high-speed*-Modus (500 kBd) zur Verfügung. An dieser Stelle wird geprüft, ob die Variable `run` aus der Struktur der seriellen Schnittstelle, den endlichen Automaten als nächstes dazu veranlasst in den Echtzeitmodus mit kontinuierlichen Messdaten zu wechseln. Dazu wird ein *all-the-measured-values-in-a-scan-continuously*-Telegramm verschickt. Der Konfigurationsteil des endlichen Automaten ist damit beendet. Soll nicht gleich in den Echtzeitmodus gewechselt werden, so bleibt der Automat in diesem Zustand, bis `run` einen Wert ungleich Null hat.

Im Zustand `DATA` und dem dazugehörigen Wartezustand `DATAwait` (gelber Bereich) findet eine Überprüfung statt, ob Messdatentelegramme vom Lasermesssystem empfangen werden konnten. Ist dies nicht der Fall, so muss in den `IDLE`-Zustand gewechselt werden und das Lasermesssystem erneut synchronisiert werden. Der Automat kann dazu veranlasst werden, den Echtzeitmodus von sich aus zu verlassen. Dazu muss die Variable `run` auf null gesetzt werden. Dieses Konstrukt ist dazu da, falls ein Lasermesssystem manuell umkonfiguriert oder betrieben werden soll. Es besteht nämlich die Möglichkeit, direkt an das Lasermesssystem Telegramme zu senden.

Im hellroten Bereich sind die Zustände zum Verlassen des Echtzeitmodus dargestellt. Die Zustände `KILL`, `KILLwait` und `KILLpre` werden lediglich deswegen benötigt, da das Antwortpaket in der Interrupt-Service-Routine verloren gehen kann. Das Lasermesssystem stoppt das Versenden von Echtzeitmessdaten in dem Augenblick, in dem es das *output-measured-value-on-request-only*-Telegramm erhält. Zu dem Zeitpunkt kann bereits ein Teil eines Messdatentelegramm versendet worden sein, so dass die Daten des Antwortpakets für einen Teil des Messdatentelegramms gehalten werden und damit verloren gehen. Auch ein Löschen des Puffers kann zu keiner 100-prozentigen Sicherheit führen. Kommt keine Antwort, so kann mit dem Zustand `KILL` zunächst überprüft werden, ob das Lasermesssystem den Echtzeitmodus verlassen hat und normal funktioniert. Ist dies nicht der Fall, so wird in `KILLpre` erneut ein Telegramm zum Beenden des Echtzeitmodus verschickt.

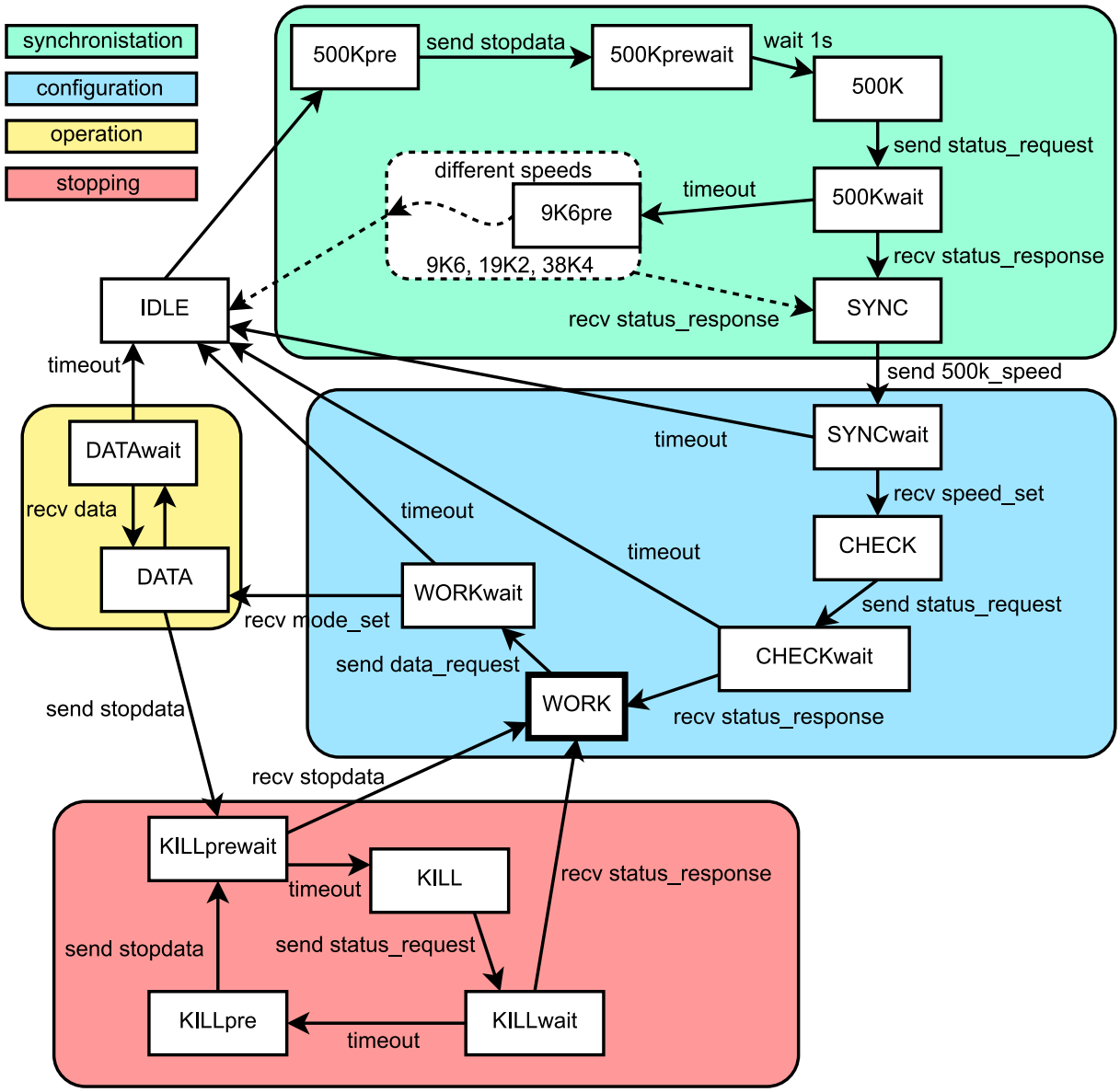


Abbildung 5.5: Automat der Lasermesssystem-Ansteuerung

Der endliche Automat benötigt einige Funktionen, um die an den Zustandsübergängen angegebenen Aufgaben zu lösen. Am Wichtigsten ist der Empfang von Telegrammen. Die Interrupt-Service-Routine legt die empfangenen Telegramme in der am Anfang dieses Kapitels beschriebenen Datenstruktur (siehe Abschnitt 5.2) ab. Die regelmäßig aufgerufene Funktion `cofunc int do_packet(fsm *port)` verarbeitet für den Port `port` die Telegramme, die im Eingangspuffer liegen. Die entsprechenden Zustandsübergänge werden ausgeführt.

Das Versenden von Telegrammen geschieht über den Ausgabepuffer. Dieser ist 256 Byte lang und fasst damit alle Telegrammtypen. Alle Millisekunde wird ein Byte von diesem Puffer an das Lasermesssystem gesendet, bis der Puffer leer ist. Der Abstand zwischen zwei aufeinander folgenden Byte muss mindestens 55 Mikrosekunden und maximal sechs Millisekunden sein. Der oft benötigte Timeout wird über die Funktion `cofunc int do_timeout(fsm *port)` umgesetzt. Diese Funktion prüft für die einzelnen Zustände, ob es zu einem Timeout gekommen ist und welches der dazugehörige Folgezustand ist. Um die Baudrate zu ändern, gibt es die Funktionen `porte_set_speed()` und `portf_set_speed()` für die beiden seriellen Schnittstellen. Sie schreiben in die entsprechenden Timer-Register die passenden Teiler.

5.5 Vorverarbeitung der Lasermessdaten

In diesem Abschnitt werden die Vorverarbeitungsfunktionen erläutert, die außerhalb der Interrupt-Service-Routine (ISR) realisiert sind. In Kapitel 5.3 ist bereits die Einteilung in Telegramme und die Kontrolle der CRC-Prüfsumme beschrieben. Die im Folgenden betrachteten Funktionen setzen jeweils ein komplett empfangenes Telegramm eines Lasermesssystems voraus. Im Rahmen der Vorverarbeitung außerhalb der ISR sollen die empfangenen Messdaten ausgewertet und ein eigenes Telegramm mit den Ergebnissen der Operationen zusätzlich zum Messdatentelegramm des Lasermesssystems an den Hostrechner gesendet werden.

Folgende Funktionen sind zu realisieren:

- Bestimmung des Minimalabstands
- Markenextraktion

5.5.1 Bestimmung des Minimalabstands

Bei der Bestimmung des Minimalabstands wird die Entfernung zum nächstgelegenen Gegenstand errechnet. Dieser Abstand soll ausgehend vom Robotermittelpunkt ermittelt werden. Für den späteren Betrieb wird lediglich eine Lösung implementiert, bei der nur die Unterschreitung von zwei verschiedenen Radien um den Robotermittelpunkt geprüft wird. Diese Entscheidung beruht darauf, dass die Rechenleistung der Rabbit-3000-CPU nicht ausreicht, um die exakte Bestimmung des geringsten Abstands für jedes Messdatentelegramm durchzuführen.

Nutzwert von zwei Überwachungsbereichen

Auf dem Serviceroboter ist im `mobiled` ein Funktionsaufruf zur Bestimmung des Minimalabstands implementiert. Der Aufruf dieser Funktion erfolgt mit dem Ziel der Kollisionsvermeidung.

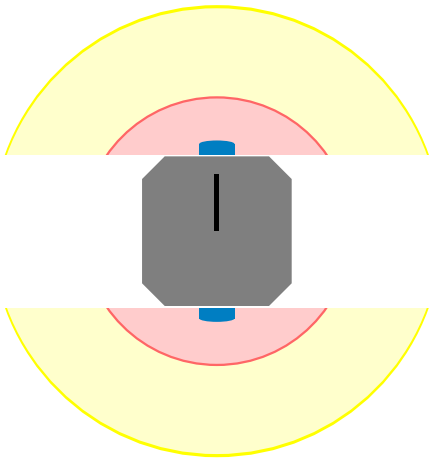


Abbildung 5.6: Die zwei Überwachungsbereiche um den Robotermittelpunkt

Für jedes der beiden Lasermesssysteme kann dieser Aufruf unabhängig voneinander erfolgen. Wenn nun bereits Informationen aufgrund der Vorverarbeitungsfunktion vorliegen, inwiefern der Minimalabstand zwei Radien unterschreitet, kann für folgende Fälle eine weitere Berechnung des genauen Minimalabstands in dem Hostrechner entfallen:

- Es wird keiner der Radien unterschritten - das nächstgelegene Hindernis ist so weit entfernt, dass es nicht gesondert berechnet werden muss.
- Beide Radien werden unterschritten - in diesem Fall darf in entsprechende Richtung keinesfalls weiter gefahren werden.

Bei den genannten Konstellationen wird von der Minimalabstandsfunktion stets ein Standardwert ausgegeben, unabhängig von dem tatsächlichen Minimalabstand zum nächsten Hinderniss. Dieses Verhalten muss mit den weiteren Programmteilen des Serviceroboters abgestimmt werden.

Für den Fall, dass nur der größere der beiden Radien unterschritten wird, muss weiterhin der exakte Wert vom Steuerrechner des Serviceroboters berechnet werden. Die beiden Radien sind mit einem geeigneten Wert zu belegen, so dass der Bewegungsvorgang rechtzeitig gebremst bzw. gestoppt werden kann. Eine grafische Darstellung der zwei Überwachungsbereiche zeigt Abbildung 5.6

Implementierung der Bereichsprüfung

Bei der Implementierung der Prüfung zweier Radien auf Unterschreitung wird aus Performancegründen ein Algorithmus implementiert, der jeden Messwert mit zwei zuvor berechneten Schwellwerten vergleicht. Da der Abstand zwischen dem Robotermittelpunkt und dem gemessenen Punkt von Interesse ist, variieren die einzelnen Schwellwerte je nach Messwinkel. Es werden Listen mit Abstandswerten errechnet, so dass für jeden Messwert abhängig vom Messwinkel die entsprechenden Vergleichswerte aus den Listen herangezogen werden. In Abbildung 5.7 ist die den folgenden Berechnungen zu Grunde liegende Geometrie dargestellt.

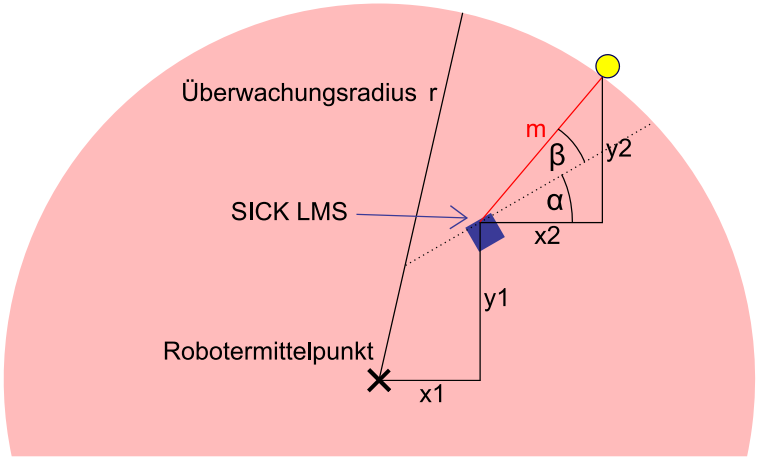


Abbildung 5.7: Geometrische Betrachtung der Bereichsüberwachung

Die Werte x_1, y_1 und α sind fest vorgegeben. Über diese Größen wird festgelegt, an welcher Stelle ein Lasermesssystem auf dem Serviceroboter montiert ist. y_1 bezieht sich hierbei auf den Abstand zum Robotermittelpunkt in Fahrtrichtung. x_1 ist der Abstand senkrecht zur Fahrtrichtung. Der Winkel α zeigt den Montagewinkel. Dieser ist so festgelegt, dass bei einem Winkel von 0° das Lasermesssystem direkt in Fahrtrichtung messen würde, also alle Bereiche bis ±90° ausgehend von der Fahrtrichtung[2]. Beim Serviceroboter des Arbeitsbereichs TAMS sind diese Werte wie folgt belegt:

LMS in Fahrtrichtung:

- $x_1 = 0$ mm
- $y_1 = 336$ mm
- $\alpha = 0°$

LMS an der Roboterrückseite:

- $x_1 = 0$ mm
- $y_1 = -336$ mm
- $\alpha = 180°$

Bei der Berechnung ist die Berücksichtigung des Versatzes senkrecht zur Fahrtrichtung und die Möglichkeit, beliebige Winkel zu verwenden, aufgrund der aktuellen Installation der Lasermesssysteme nicht zwingend notwendig. Da die Datenstruktur im `mobiled` dieses aber explizit vorsieht, soll bei der Bestimmung der Grenzwerte die Möglichkeit der Verwendung beliebiger Werte gegeben sein. Zur Verdeutlichung dieser einzelnen Größen ist in Abbildung 5.7 eine Installationsgeometrie gezeigt, die nicht die Installation der Lasermesssysteme auf dem Serviceroboter widerspiegelt.

Der Winkel β ist der aktuelle Messwinkel des Lasermesssystems. Wie in Kapitel 2.2 erläutert, messen diese Systeme von oben betrachtet gegen den Uhrzeigersinn in Halbgradschritten. Die erste Messung soll einem Winkel von 0° entsprechen, die 361. demnach einem Winkel von 180°.

[2] Ausgehend vom LMS und nicht vom Robotermittelpunkt

Bei gegebenem Abstand r vom Robotermittelpunkt ist für jeden Messwinkel β die Entfernung m zu errechnen, bei der dieser Abstand erreicht wird. Mit dem Satz des Pythagoras ergibt sich folgender Zusammenhang:

$$r^2 = (x_1 + x_2)^2 + (y_1 + y_2)^2 \tag{5.1}$$

x_2 und y_2 berechnen sich folgendermaßen:
$x_2 = \cos(\alpha + \beta) \cdot m$ Mit $\gamma = \alpha + \beta \Rightarrow x_2 = \cos\gamma \cdot m$
$y_2 = \sin\gamma \cdot m$
Damit ergibt sich (5.1) zu:

$$r^2 = (x_1 + \cos\gamma \cdot m)^2 + (y_1 + \sin\gamma \cdot m)^2 \tag{5.2}$$

$$r^2 = x_1^2 + 2 \cdot x_1 \cdot (\cos\gamma) \cdot m + \cos^2\gamma \cdot m^2 + y_1^2 + 2 \cdot y_1 \cdot (\sin\gamma) \cdot m + \sin^2\gamma \cdot m^2 \tag{5.3}$$

$$r^2 = x_1^2 + y_1^2 + (2 \cdot x_1 \cdot \cos\gamma + 2 \cdot y_1 \cdot \sin\gamma) \cdot m + \left(\sin^2\gamma + \cos^2\gamma\right) \cdot m^2 \tag{5.4}$$

Mit $\sin^2\gamma + \cos^2\gamma = 1$ ergibt sich:

$$r^2 = x_1^2 + y_1^2 + (2 \cdot x_1 \cdot \cos\gamma + 2 \cdot y_1 \cdot \sin\gamma) \cdot m + m^2 \tag{5.5}$$

Sei $p = 2 \cdot x_1 \cdot \cos\gamma + 2 \cdot y_1 \cdot \sin\gamma$ und $q = x_1^2 + y_1^2 - r^2$

$$0 = m^2 + p \cdot m + q \tag{5.6}$$

Sowohl p als auch q sind aus gegebenen Größen berechenbar. Als Lösungen für m ergeben sich über die allgemeine Lösung für die normierte quadratische Gleichung:

$$m_{1/2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q} \tag{5.7}$$

Es wird stets davon ausgegangen, dass sich das Lasermesssystem innerhalb des Überwachungsradius befindet. Andere Konfigurationen ergeben keinen Sinn. Es kann auf diese Art mathematisch eine positive und eine negative Lösung geben. Die negative Lösung liefert den Abstand zwischen dem Lasermesssystem und dem Überwachungsradius auf entgegengesetzter Seite hinter dem Lasermesssystem. Diese negative Entfernung ist nur eine theoretische Lösung und soll unberücksichtigt bleiben. Da $\sqrt{p^2/4 - q}$ keine negativen Werte annimmt, ist die größere der beiden Lösungen die gesuchte positive:

$$m = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q} \tag{5.8}$$

Die Auswertung erfolgt für jedes Lasermesssystem mit zwei verschiedenen Radien, so dass pro Lasermesssystem zwei Listen mit je 361 Werten entstehen. Bei der Berechnung wird mit Fließkommazahlen gearbeitet. Das Ergebnis wird nach einem Cast als 16-Bit-Integerwert in Millimeter

gespeichert. So ist ein direkter Vergleich mit den Messdaten möglich. Der Rundungsfehler kann gegenüber der Messungenauigkeit der Lasermesssysteme vernachlässigt werden.

Eine Berechnung der Listen erfolgt stets zu Systemstart. Im Betrieb kann eine Neuberechnung veranlasst werden, wenn ein entsprechendes UDP-Telegramm mit aktuellen Werten für die Positionen der Lasermesssysteme und die zu überwachenden Radien an das System geschickt wird. Wird ein Messdatentelegramm empfangen, so startet die Prüfung auf Bereichsverletzungen. Wenn noch keiner der Bereiche unterschritten wurde, wird nur der weiter entfernte Bereich geprüft. Erfolgt eine Unterschreitung dieses Bereiches, wird bei demselben Messwert die Unterschreitung des näheren Bereichs ebenfalls geprüft. Bei allen folgenden Messwerten des Telegramms wird nur noch die Unterschreitung des näheren Bereichs geprüft. Liegt ein Gegenstand innerhalb dieses Bereiches, so wird bei allen übrigen Werten des aktuellen Messdatensatzes keinerlei Prüfung mehr vorgenommen.

Das Ergebnis wird als sogenannter „Alarm_Level“ ausgegeben. Die Bedeutung dieses Byte ist folgende:

- Alarm_Level=0 — Es liegt keine Bereichsverletzung vor.
- Alarm_Level=1 — Nur der äußere Bereich ist unterschritten.
- Alarm_Level=2 — Beide Bereiche sind unterschritten.

Wenn keine Bereichsverletzungen gemeldet werden, können sich trotzdem Gegenstände innerhalb der Bereichsradien in Bereichen befinden, die von den Lasermesssystemen nicht erfasst werden können. Diese Bereiche befinden sich beim Serviceroboter an den Seiten. Innerhalb der Vorverarbeitung wird zudem keine Prüfung vorgenommen, ob die angegebenen Positionen der Messsysteme überhaupt einen sinnvollen Kollisionsschutz ergeben. Entsprechende Vorkehrungen sind in der Software des Hostrechners zu treffen.

Visualisierung der Überwachungsbereiche

Die drei verschiedenen Zustände pro Lasermesssystem sollen über Leuchtdioden im selbstgebauten Gehäuse für den Rabbit Powercore 3800 visualisiert werden. Dazu werden je Lasermesssystem drei Leuchtdioden (rot, gelb, grün) installiert und über einen freien Parallelport und einen Treiberchip angebunden. Für jedes Lasermesssystem kann nun der Abstand des nächstgelegenen Gegenstands zum Robotermittelpunkt in drei Stufen angezeigt werden. Die Funktion der Anzeige erfolgt ähnlich einer Einparkhilfe in modernen Autos. In Abbildung 4.3 auf Seite 101 ist eine Fotografie dieser Anzeige zu sehen. In vorliegender Konstellation zeigen die leuchtenden grünen LEDs an, dass bei beiden Lasermesssystemen keine Bereichsverletzungen vorliegen. Die Anzeige dient hauptsächlich zu Demonstrationszwecken. Zum schnellen Test des Systems auf Funktionsfähigkeit kann ein Gegenstand in den Nahbereich des Serviceroboters gebracht werden und überprüft werden, ob die Anzeige auf diese Bereichsverletzung reagiert.

Bestimmung des exakten Abstandswertes

Wie bereits erwähnt, verfügt der Rabbit 3000 nicht über genug Rechenleistung, um für jedes Messdatentelegramm den Minimalabstand zum Robotermittelpunkt zu berechnen. Die Gründe dafür sollen im Folgenden erläutert werden. Zunächst muss auf die Ausführungen in Kapitel 6.2

vorgegriffen werden. Dort wird aufgezeigt, dass die Auslastung der Rabbit-3000-CPU auch schon mit der Auswertung der zwei Radien in die Nähe eines Bereiches kommt, wo der Verlust von Messdaten droht. Eine Bestimmung des Minimalabstands ausgehend von den Lasermesssystemen wäre ohne weiteres möglich, da hier nur die einzelnen Messwerte verglichen werden müssten. Die Umrechnung zur Distanz ausgehend vom Robotermittelpunkt erfordert hingegen einen hohen Rechenaufwand.

Es sollen nun die Rechenoperationen aufgezeigt werden, die für die Bestimmung des exakten Minimalabstands auszuführen sind. Ausgangspunkt ist die oben hergeleitete Gleichung 5.2:

$$r^2 = (x_1 + \cos\gamma \cdot m)^2 + (y_1 + \sin\gamma \cdot m)^2$$

In diesem Fall ist r zu bestimmen und ergibt sich zu:

$$r = \sqrt{(x_1 + \cos\gamma \cdot m)^2 + (y_1 + \sin\gamma \cdot m)^2} \tag{5.9}$$

In einem Messdatensatz muss für jeden der 361 Messwerte dieser Wert berechnet werden, wobei der niedrigste dieser Werte der gesuchte ist. Pro Rechnung sind vier Additionen (eine dient zur Berechnung von γ), zwei trigonometrische Funktionen (Sinus und Cosinus), vier Multiplikationen (zwei davon sind die Quadrierungen) und eine Quadratwurzel zu berechnen. Diese Operationen müssen mit Fließkommazahlen erfolgen, da nur so mit den trigonometrischen Funktionen gearbeitet werden kann. Wie in Kapitel 3.2.3 erläutert ist, werden bei der Rabbit-3000-CPU Fließkommaoperationen hardwareseitig nicht unterstützt und müssen softwareseitig mit geringer Geschwindigkeit emuliert werden. Beim Betrieb mit dieser Bestimmung des Minimalabstands kommt es zwangsweise zum Überlaufen des Telegrammpuffers, da die Vorverarbeitung zu viel Rechenzeit beansprucht.

Bei dem Algorithmus besteht noch Optimierungspotential, wenn die Berechnung speziell auf die im Serviceroboter gegebene Geometrie der Lasermesssysteme angepasst wird. Bei der Berechnung von Sinus und Cosinus werden nur bestimmte Winkel abgefragt, da der Winkel β immer in Halbgradschritten ansteigt und zu diesem Winkel entweder 0° oder 180° addiert werden. Die benötigten Werte können dann in einer Tabelle vorab gespeichert werden. Der Term x_1 kann entfallen, da dieser stets null ist. Es ergibt sich folgende Gleichung:

$$r = \sqrt{(\cos\gamma \cdot m)^2 + (y_1 + \sin\gamma \cdot m)^2} \tag{5.10}$$

$$r = \sqrt{\cos^2\gamma \cdot m^2 + y_1^2 + 2 \cdot y_1 \cdot \sin\gamma \cdot m + \sin^2\gamma \cdot m^2} \tag{5.11}$$

$$r = \sqrt{m^2 + 2 \cdot y_1 \cdot \sin\gamma \cdot m + y_1^2} \tag{5.12}$$

Für das Lasermesssystem in Fahrtrichtung ergibt sich:

$$r = \sqrt{m^2 + 2 \cdot 336\ \text{mm} \cdot \sin\beta \cdot m + 112896\ \text{mm}^2} \tag{5.13}$$

Für das Lasermesssystem an der Roboterrückseite ergibt sich:

$$r = \sqrt{m^2 + 2 \cdot -336\ \text{mm} \cdot \sin(\beta + 180°) \cdot m + 112896\ \text{mm}^2} \tag{5.14}$$

$$r = \sqrt{m^2 + 2 \cdot -336\ \text{mm} \cdot -\sin\beta \cdot m + 112896\ \text{mm}^2} \tag{5.15}$$

$$r = \sqrt{m^2 + 2 \cdot 336\ \text{mm} \cdot \sin\beta \cdot m + 112896\ \text{mm}^2} \tag{5.16}$$

Für beide Lasermesssysteme kann also mit der gleichen Formel gearbeitet werden. Die Tabelle zur Bestimmung der Sinuswerte kann dahingehend erweitert werden, dass der Faktor $2 \cdot y_1$, mit dem entsprechenden Sinuswert multipliziert, abgespeichert wird. Dieses spart im Betrieb zusätzlich Rechenzeit. Der Floatwert für y_1^2 wird ebenfalls beim Programmstart berechnet. Es bleiben also noch zwei Multiplikationen, zwei Additionen und eine Quadratwurzel.

Die Anzahl der Berechnungen kann außerdem durch Änderung des Ablaufs der Auswertung reduziert werden. An den Außenbereichen des Lasermesssystems wird mit der Berechnung begonnen und es werden zur Mitte hin in jedem Schritt zwei Messwerte abgearbeitet. Wenn ein Messwert den bis dahin niedrigsten Abstand ergibt, wird sowohl der Messwert als auch der dazugehörige Minimalabstand gespeichert. In den darauf folgenden Schritten erfolgt zunächst nur ein Vergleich der Messwerte. Ist einer der folgenden Messwerte größer als der gespeicherte Wert, aus dem der bisher niedrigste Abstand berechnet wurde, so braucht die Berechnung für den Wert gar nicht erst zu erfolgen. Der Minimalabstand ist bei festem m abhängig von dem Messwinkel und bei 90° am größten, da der Term $\sin\beta$ dann maximal ist. Deshalb kann im beschriebenen Fall das Ergebnis der Operation keinen neuen Minimalwert liefern und die Berechnung muss nicht ausgeführt werden.

Wenn eine gewisse Ungenauigkeit bei der Berechnung des Minimalabstands in Kauf genommen werden kann, ist nochmals eine Senkung der Zahl der Operationen möglich, indem von dem abgespeicherten Messwert der gewünschte Abstand subtrahiert wird. Eine Neuberechnung erfolgt dann nur, wenn sich der aktuelle Messwert signifikant von dem bisherigen Wert unterscheidet.

Die Implementierung dieser Verarbeitungsstrategie erweist sich als teilweise praktikabel. Es ist möglich, für eines der beiden Lasermesssysteme jeden zweiten Messdatensatz auszuwerten, ohne dass es im Normalbetrieb aufgrund der daraus resultierenden Systemlast zu Telegrammverlusten kommt. Sobald jedoch künstlich ein ungünstiges Szenario erzeugt wird, gehen Telegramme aufgrund zu hoher Systemauslastung verloren. Dieses Szenario muss eine Raumgeometrie beinhalten, die von Außenbereichen der Lasermesssysteme zum Zentrum einen stets sinkenden Distanzmesswert erzeugt (etwa ein großer Karton mit einer Ecke direkt vor dem Lasermesssystem aufgestellt). Da nicht ausgeschlossen werden kann, dass im Betrieb ähnliche Konstellationen vorkommen, wird auf die Implementierung der Minimalabstandsbestimmung verzichtet. Die Hauptfunktion des Systems, die Übertragung der Messdatentelegramme, soll in keinem Fall gefährdet sein.

5.5.2 Reflektormarkenextraktion

Die Navigation des Serviceroboters erfolgt in der aktuellen Implementierung auf Basis der erkannten Reflektormarken. In der bisherigen Implementierung wird innerhalb des `mobiled` eine Liste mit Winkeln und Entfernungen der erkannten Marken erzeugt. Zur Reduzierung der Auslastung des Steuerrechners kann diese Aufgabe im Rabbit Powercore 3800 ausgeführt werden. Die Lasermesssysteme senden in der aktuellen Konfiguration zu jedem Messwert die Reflexionsstärke als 3-Bit-Wert. Prinzipiell könnte jeder Messwert mit einer Reflexionsstärke größer als null direkt in diese Liste eingetragen werden. Es kommt jedoch oft dazu, dass eine Reflektormarke in mehreren benachbarten Messungen erkannt wird. In diesem Fall sollen der tatsächliche Winkel und die Entfernung dieser Marke näherungsweise berechnet werden. In der zu generierenden Liste soll die Marke dann nur einmal aufgeführt sein.

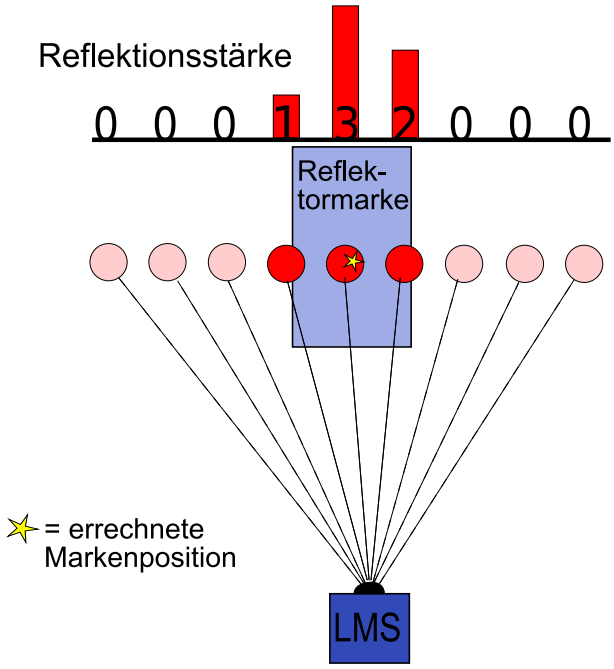


Abbildung 5.8: Bestimmung der Markenposition bei mehrfacher Erkennung einer Marke

Implementierung der Markenextraktion

Treffen folgende Bedingungen zu, so wird davon ausgegangen, dass Messwerte mit einer Reflexionsstärke größer als null zu derselben Marke gehören:

- Es handelt sich um benachbarte Messungen.
- Der Distanzwert unterscheidet sich um nicht mehr als 10 cm.

Eine solche Konstellation ist grafisch in Abbildung 5.8 dargestellt. Beim Auftreten dieser Konstellation soll der Winkel dieser Marke und die Entfernung gemittelt werden. Es kann dabei die Reflexionsstärke mit einbezogen werden, so dass die Messwerte mit starker Reflexion ein größeres Gewicht bekommen. Obwohl kein linearer Zusammenhang zwischen tatsächlicher Reflexionsstärke und dem 3-Bit Wert besteht, wird im Folgenden ein solcher unterstellt, da nur so eine sinnvolle Einbeziehung der Reflexionsstärke möglich ist.

Bei der Auswertung werden die Messwerte von 0° bis 180° der Reihe nach durchlaufen. Wenn der erste Messwert einer Marke erkannt wird, werden folgende Schritte durchgeführt:

- Speicherung des aktuellen Winkels in halben Grad in der Variablen für den Basiswinkel
- Speicherung der Distanz der Marke in der Variablen für die Basisdistanz
- Speicherung der Reflexionsstärke in der Variablen für die Gesamtreflexionsstärke
- Initialisierung der Variablen für die gewichteten Winkelversätze mit null
- Initialisierung der Variablen für die gewichteten Distanzversätze mit null

Für weitere Messwerte, die nach oben genannten Bedingungen der gleichen Marke zuzuordnen sind, kommt es zu folgenden Operationen:

- zur Variablen für die gewichteten Winkelversätze wird folgender Wert addiert:

$$\text{aktuelleReflexionsstärke} \cdot (\text{aktuellerWinkel} - \text{Basiswinkel})$$

- zur Variablen für die gewichteten Distanzversätze wird folgender Wert addiert:

$$\text{aktuelleReflexionsstärke} \cdot (\text{aktuelleDistanz} - \text{Basisdistanz})$$

- Addition der aktuellen Reflexionsstärke zur Variablen für die Gesamtreflexionsstärke

Ist die letzte Messung einer Marke abgeschlossen, können die Werte gemittelt werden. Die Variable für die Distanzversätze wird durch die Gesamtreflexionsstärke geteilt und zur Basisdistanz addiert. Bei der Berechnung des Winkels soll die Auflösung erhöht werden, so dass der gemittelte Winkel auf $\frac{1}{120}$ Grad genau in einem 16-Bit-Wert ausgegeben wird. Dazu erfolgt vor der Mittelwertbestimmung eine Multiplikation mit 60, da zuvor eine Auflösung von 0,5 Grad vorliegt. Dieser Wert wird gewählt, damit kein Überlauf des 16-Bit-Wertes zu befürchten ist und für möglichst viele Werte der Gesamtreflexionsstärke eine Division ohne Rest möglich ist ($60 = 2 \cdot 2 \cdot 3 \cdot 5$).

Diese Methode der Mittelwertbestimmung über die gewichteten Differenzwerte hat folgenden Vorteil gegenüber der Addition aller gewichteten Werte: Da nur die Differenzen summiert werden, die aufgrund der Vorprüfung zwangsläufig gering sind, droht kein Überlauf der 16-Bit-Variablen. Es ist somit nicht notwendig, die Berechnungen auf Fließkommabasis durchzuführen, was zu einer inakzeptablen Rechenzeit führen würde.

5.5.3 Kombination von Reflektormarkenextraktion und Abstandsüberwachung

Zur effizienten Implementierung der Markenextraktion und der Überprüfung auf Unterschreitung zweier Überwachungsradien soll die Abarbeitung beider Funktionen verschachtelt erfolgen. Für jeden Messwert werden zunächst die Operationen zur Abstandsbestimmung durchgeführt, bevor die Verarbeitung zur Generierung der Markenliste startet. Auf diese Weise braucht jeder Messwert nur einmal geladen zu werden. Dies hat den Vorteil, dass die Berechnung der Zeiger nur einmal erforderlich wird. Außerdem braucht die Distanz nur einmal berechnet zu werden. Diese ergibt sich aus 13 Bit, die über zwei Byte verteilt sind. Der Vorteil durch diese Kombination ist erheblich. Entsprechende Messungen sind in Kapitel 6.4 dargestellt.

5.6 Datenpakete

In diesem Abschnitt werden die Formate der einzelnen Pakete vorgestellt die zwischen dem Rabbit 3000 und einem oder mehreren Hostrechnern ausgetauscht werden. Hier nicht beschriebene Pakete sollten von Anwendungsentwicklern grundsätzlich ignoriert werden, da es sich dabei um zusätzliche Erweiterungen handelt, die nicht immer zur Verfügung stehen und sich ändern können.

5.6.1 Vom Rabbit 3000 zum Hostrechner

Alle Daten, die vom Rabbit 3000 zum Hostrechner verschickt werden, gehen an die Multicast-IP-Adresse 224.0.0.23 Port 2222. Der Rabbit 3000 braucht keine Netzwerkkonfiguration, um seine Daten zu versenden. Die Datenpakete sind so klein, dass sie stets in ein UDP-Paket passen. Auf Anwenderseite kann damit jedes UDP-Paket einzeln abgearbeitet werden.

Es gibt verschiedene Sorten von UDP-Paketen, die der Rabbit 3000 verschickt:

- Telegramme: Sie fangen mit dem Byte 0x02 an, gefolgt von einer Adresse (acht Bit breit) und der Länge (16 Bit breit, *little-endian*) des Telegramms.
- ACK/NACK: SICK-Lasermesssysteme bestätigen den Empfang jedes Telegramms mit einem ACK- oder NACK-Byte. Je nach Konfiguration des Rabbit 3000 werden diese einzelnen Byte 0x06 für ACK und 0x15 für NACK übertragen oder nicht. Normalerweise sind sie deaktiviert, da nicht zwischen den beiden Lasermesssystemen unterschieden werden kann.
- Debug-Meldungen: Der Status des Rabbit 3000 kann mit bestimmten UDP-Paketen abgefragt werden. In den Debug-Meldungen kommt natürlichsprachlicher ASCII-Text zurück. Diese Pakete werden nur bei Bedarf versendet und sollten im Normalbetrieb ignoriert werden.

Die Telegramme können anhand der Adresse im Headerfeld in zwei Gruppen eingeteilt werden. Ist die Adresse kleiner als 0x80, so handelt es sich um Telegramme, die vom Rabbit 3000 an ein Lasermesssystem verschickt wurden. Ist die Adresse 0x00, so ging das Telegramm an das Lasermesssystem am seriellen Port E. Ist die Adresse 0x01, so wurde das Telegramm über Port F versendet. Telegramme dieser Sorte dienen ausschließlich dem Debuggen, um zu sehen, wie erfolgreich die Lasermesssysteme angesteuert werden. Eine Auswertung im Hostrechner ist nicht empfehlenswert. Die Adresse eines Telegramms ist größer oder gleich 0x80, wenn es sich um ein Telegramm vom Lasermesssystem handelt bzw. wenn es Daten enthält, die vom Lasermesssystem kommen. Dazu gehören vom Rabbit 3000 vorverarbeitete Daten. Das niederwertigste Bit der Adresse gibt dabei an, ob es sich um den seriellen Port E (Adressen 0x80, 0x90, ...) oder um den seriellen Port F (Adressen 0x81, 0x91, ...) handelt. Die Lasermesssysteme antworten zwar immer mit der Adresse 0x80, jedoch wird bei eingehenden Paketen an dem Port F das Adressfeld um eins inkrementiert. Durch die Änderung der Adresse im Header stimmt die CRC-Prüfsumme des Telegramms nicht mehr. Je nach Konfiguration des Rabbit 3000 findet eine Anpassung der CRC-Prüfsumme statt oder nicht. Es wird empfohlen auf die Prüfung der CRC-Prüfsumme zu verzichten, da eingehende Telegramme vom Lasermesssystem bereits im Rabbit 3000 geprüft werden und UDP-Pakete im IP-Header eine Prüfsumme haben.

Adressen 0x80 und 0x81

Die Telegramme mit den Adressen 0x80 und 0x81 wurden so vom Lasermesssystem empfangen. Einzig die Adresse und ggf. die CRC-Prüfsumme wurden angepasst. Die Bedeutung der einzelnen Telegramme lässt sich in [SickTL] nachlesen. An dieser Stelle wird nur das Telegrammformat mit den Messdaten der Lasermesssysteme vorgestellt.

Das UDP-Paket/Telegramm fängt mit dem STX-Byte 0x02 an. Es folgt die Adresse 0x80 oder 0x81. Ein Messdatentelegramm hat eine Länge von 732 Byte. Im Header ist jedoch die Länge ohne Header und Prüfsumme angegeben, so dass eine Länge von 726 Byte im Header (*little-endian*) folgt. Die beiden Längenbyte sind damit 0xD6 und 0x02. Nach dem vier Byte langen

Header folgt das Kommando, es ist 0xB0 (Antwort auf Messwertanforderung). Die beiden darauf folgenden Byte geben die Messauflösung (Millimeter- oder Zentimetermodus) und die Anzahl der Messwerte an. Für Winkelschritte mit 0,5 Grad und 361 Messwerten ergibt sich 0x4169 (*little-endian*: 0x69 0x41). Es folgen 361 mal zwei Byte im *little-endian*-Format. Bei einer Konfiguration mit einem Messbereich von acht Metern geben die untersten 13 Bit die Entfernung in Zentimetern bzw. Millimetern an. Es ist dabei zu berücksichtigen, dass der maximal gültige Hexwert 0x1ff7 ist. Größere Messwerte werden zur Fehlerkodierung genutzt, im Fehlerfall sollte der Messwert für diesen Laserstrahl ignoriert werden. Die restlichen drei Bit sind aktuell für die Reflexionsstärke konfiguriert. Wenn keine Reflektormarke in dem Winkel zu sehen ist, wird null zurückgeliefert. Ansonsten ist eine Reflektormarke wenigstens zum Teil sichtbar. Details zu einer möglichen Implementierung sind im Anhang A zu finden.

Adressen 0x90 und 0x91

Im Rabbit 3000 werden die Messwerte der Lasermesssysteme bereits vorverarbeitet. Es werden Reflektormarken extrahiert und Bereichsverletzungen erkannt. Die gewonnen Informationen aus der Vorverarbeitung werden in Telegrammen mit den Adressen 0x90 bzw. 0x91 an den Hostrechner übertragen. Das fünfte Byte (Kommando) ist bei diesen Telegrammen 0xB3. Für jede gefundene Reflektormarke folgen vier Byte. Der Winkel wird als 16-Bit-Integer in 120-tel Grad angegeben. Die Entfernung wird in Millimetern als 16-Bit-Integer angegeben. Zuerst kommt die Entfernung in *little-endian*-Reihenfolge, dann der Winkel. Die Anzahl der gefundenen Marken ist der im Header gespeicherten Telegrammlänge zu entnehmen, sie ist $\frac{\text{Länge}-2}{4}$. Das eine Byte ist das Kommando Byte, das Andere folgt nach den Marken und gibt eine Kollisionswarnung an. Ist es Null, so befinden sich keine Hindernisse im Radius von einem Meter um den Robotermittelpunkt von diesem Lasermesssystem. Ein Wert von zwei gibt an, dass sich ein Hindernis innerhalb des 60-Zentimeter-Radius befindet. Ist der Wert eins, so befindet sich ein Hindernis zwischen 60 und 100 Zentimeter Entfernung. Details zur Kollisionswarnung sind in 5.5.1 zu finden.

Ein Telegramm kann wie folgt aussehen:

STX	Adr	Länge	CMD	1. Dist	1. Winkel	2. Dist	2. Winkel	Kol.
0x02	0x90	0x0A 0x00	0xB3	0x46 0x09	0x18 0x15	0x23 0x0F	0xB1 0x3B	0x01
		10 (2 Marken)		2374 mm	5400=45°	3875 mm	15281=127,34°	gelb

5.6.2 Vom Hostrechner zum Rabbit 3000

Der Rabbit 3000 lässt sich ausschließlich über die Ethernet-Buchse vom Powercore 3800 ansteuern. Er verarbeitet UDP-Pakete, die an seinen Port 2222 gehen. Die aktuell konfigurierte IP-Adresse ist 192.168.0.2 für das private Subnetz vom Hostrechner und dem Mikroprozessor.

Rabbit 3000 neustarten

Empfängt der Rabbit 3000 ein UDP-Paket, dass mit den fünf Byte `reset` anfängt, wird ein Softreset durchgeführt. Die Dynamic-C-Funktion `void forceSoftReset()` wird dazu aufgerufen. Sie springt zum Anfang des BIOS, um dort mit der Codeausführung fortzufahren. Nach

einem Softreset gehen kurz die LEDs aus und fangen dann an zu blinken, bis die Lasermesssysteme wieder synchronisiert und konfiguriert sind.

Statusinformationen abfragen

Wird ein UDP-Paket gesendet, bei dem das erste Byte `'i'` (0x69) ist, so liefert der Rabbit 3000 an die Multicast-Adresse ein UDP-Paket mit natürlichsprachlichem Text. Die Antwort könnte zum Beispiel wie folgt aussehen:

```
E:: run:1 btrash:73 ttrash:0 fill:0 F:: run:1 btrash:134 ttrash:0 fill:1
```

Die erste Hälfte an Statusinformationen bezieht sich auf Port E (`E::`), die zweite auf Port F (`F::`). `run` gibt an, ob der Rabbit 3000 das Lasermesssystem in den Echtzeitmodus bringen soll. Ein Wert von eins sorgt dafür, dass der Rabbit 3000 versucht, das Lasermesssystem in den Echtzeitmodus zu versetzen. Wird dem Rabbit 3000 jedoch mitgeteilt, dass er das Lasermesssystem stoppen soll, so ist `run` Null. `btrash` steht für die Anzahl der Byte, die zum Synchronisieren in den Telegrammdatenstrom verworfen wurden. Sollte wider Erwarten während einer Verbindung ein Byte verloren gehen und der Rabbit 3000 die Synchronisation verlieren, so steigt der Wert. Änderung am Wert finden ausschließlich durch die Interrupt-Service-Routine statt, die die einzelnen Byte abholt. Die Anzahl der verworfenen Telegramme ist in `ttrash` zu finden. Ist der Puffer voll oder stimmt die CRC-Prüfsumme eines Telegramms nicht, so wird dieser Zähler um eins inkrementiert. Der aktuelle Füllstand des Puffers ist in `fill` angegeben und sollte Null oder Eins betragen.

Direkte Kommunikation mit den Lasermesssystemen

Es besteht für den Hostrechner die Möglichkeit, eigene Telegramme an ein Lasermesssystem zu schicken. Der Rabbit-3000-Mikroprozessor reicht das Telegramm bzw. die Daten dann eins zu eins weiter. Die Adresse im Telegrammheader muss in diesem Fall 0x00 sein. Außerdem wird eine gültige CRC-Prüfsumme benötigt. Dem Telegramm müssen zwei extra Byte `"de"` oder `"df"` im UDP-Paket vorangestellt werden, damit der Rabbit 3000 weiß, dass er Daten durchreichen soll (erstes Byte) und an wen er sie durchreichen soll (zweites Byte). Der Rabbit 3000 reicht vom UDP-Paket ab dem dritten Byte alle Byte mit einer Pause von einer Millisekunde zwischen den Byte an das entsprechende Lasermesssystem weiter. Telegramme, die vom Lasermesssystem kommen, werden wie gehabt per Multicast mit veränderter Adresse weitergeleitet. Es besteht hiermit die Möglichkeit, das EEPROM von den Lasermesssystemen umzuprogrammieren oder andere Modi zu nutzen.

	STX	Adr	Länge	CMD	CRC
0x64 0x65	0x02	0x00	0x0A 0x00	0xXX ...	0x01 0x02
`de`=Port E	Header			Daten	Prüfsumme

Starten und Stoppen des Echtzeitmodus

Normalerweise sollen die Lasermesssysteme im Echtzeitmodus betrieben werden. Anwendungen brauchen die Entfernungsmessdaten, um ihre Position in einer Karte zu ermitteln. Es kann jedoch

vorkommen, dass die Lasermesssysteme anderweitig benutzt werden sollen oder umkonfiguriert werden müssen. In diesem Fall ist es notwendig den Echtzeitmodus zu beenden und nicht automatisch wieder zu aktivieren. Deshalb gibt es für jedes Lasermesssystem, genaugenommen für jeden seriellen Port, eine Zustandsvariable `run`, die angibt, ob das dazugehörige Lasermesssystem im Echtzeitmodus betrieben werden soll. Diese Variable ist nach dem Booten des Rabbit 3000 auf 1 gesetzt, so dass sofort damit begonnen wird, die Lasermesssysteme in den Echtzeitmodus zu bringen. Diese Variable kann über die Statusinformation (siehe oben) abgefragt werden. Im Betrieb lässt sich der Wert von `run` mit UDP-Paketen von außen ändern, so dass der Rabbit 3000 den Echtzeitmodus Stoppen und Starten kann. Das erste Byte im UDP-Paket muss `k` (*kill*) oder `s` (*start*) lauten. Das zweite Byte gibt den Port an und ist damit `e` oder `f`.

Paket	Bedeutung
`ke`	Echtzeitmodus an Port E stoppen
`kf`	Echtzeitmodus an Port F stoppen
`se`	Echtzeitmodus an Port E starten
`sf`	Echtzeitmodus an Port F starten

Position der Lasermesssysteme am Roboter ändern

Der Rabbit 3000 hat Kenntnis von der Position der einzelnen Lasermesssysteme, um seine Kollisionsbereiche zu überwachen. Die Bereiche werden vom Robotermittelpunkt aus angegeben, so dass eine Positionsänderung der Lasermesssysteme andere Schwellwerte benötigt. Die Position eines Lasermesssystems wird in Millimetern bzw. in Grad angegeben. Das Lasermesssystem am Port E hat momentan einen x-Wert von Null, einen y-Wert von 336 und einen Winkel von Null Grad fest einprogrammiert. Das Lasermesssystem vom Port F hat die Position (0|-336) mit einem Winkel von 180 Grad. Änderungen können immer nur temporär am Rabbit 3000 durchgeführt werden. Soll eine neue Position dauerhaft gesetzt werden, so muss der Programmcode neu kompiliert und übertragen werden. Das Paket zum temporären Ändern der Lasermesssystem-Positionen ist genau 13 Byte lang, zum Beispiel:

0x70	0x00 0x00	0x50 0x01	0x00 0x00	0x00 0x00	0xB0 0xFE	0xB4 0x00
CMD	$E_x = 0$	$E_y = 336$	$E_{angle} = 0$	$F_x = 0$	$F_y = -336$	$F_{angle} = 180$

Bereichsradien für den Kollisionsschutz ändern

Der Roboter hat zwei Kollisionsschutzradien. Genauere Information, worum es sich dabei handelt, sind in Abschnitt 5.5.1 zu finden. Der äußere Radius liegt bei einem Meter, der Innere bei 60 Zentimetern. Zur Laufzeit lassen sich diese ändern. Das UDP-Paket muss fünf Byte lang sein und mit einem `w` beginnen. Die beiden darauf folgenden 16-Bit-Integer sind äußerer und innerer Radius in Millimetern im *little-endian*-Format:

0x77	0xE8 0x03	0x58 0x02
`w`	1000 mm	600 mm

```
class CLASERFEEDER : protected CTHREAD {
// ...
public:
  CLASERFEEDER(CGENBASE *genBase, const bool simulate);
  virtual ~CLASERFEEDER(void);

  int GetClosestObstacleDistance(const int idx, double &distance);
  int GetLaserScanScanner(const int idx, CRADIALSCAN &scan);
  int GetLaserScanPlatform(const int idx, CRADIALSCAN &scan);
  int GetLaserScanPlatformMatched(const int idx,
                                  CRADIALSCANMATCHED &scan);
  int GetNumScanners(void);
  int GetScannerPosition(meter_t &x, meter_t &y,
                         radian_t &a, const int idx);
}
```

Listing 5.1: `public`-Auszug aus `laserFeeder.h`

5.7 Neue LaserFeeder-Implementierung auf dem Hostrechner

Im Hostrechner des mobilen Serviceroboters muss nach der neuen Anbindung mit dem Powercore 3800 die Software im `mobiled` angepasst werden. Der `mobiled` ist die zentrale Instanz, in der alle für die Steuerung des mobilen Serviceroboters notwendigen Funktionen integriert sind. Bei der bisherigen Anbindung (beschrieben in Abschnitt 2.5.2) kapselt ein CLASERFEEDER-Objekt den Zugriff auf die Lasermessdaten. Dieses CLASERFEEDER-Objekt ist ein Thread, das die Messdaten empfängt und speichert. Zusätzlich wird für jeden Messdatensatz die Lokalisation zur Aktualisierung ihrer internen Zustände aufgerufen. Andere Threads können von diesem Objekt die Messdaten nebenläufig abholen.

Unterhalb des CLASERFEEDER-Objektes verbergen sich CLASER-, CSICKLASER-, _RAWLASER-, _CSICKRAWLASER- und CRS422-Objekte. Bei der Implementierung der neuen Softwareschnittstelle wurde aufgrund der hohen Anzahl von Klassen und der nicht ganz übersichtlichen Struktur entschieden, der Einfachheit halber eine komplett neue Implementierung des CLASERFEEDER-Objektes zu schreiben. Als Vorteil hat sich dabei erwiesen, dass ein Thread genügt, so dass sich verschiedene Threads nicht synchronisieren müssen. Außerdem kommen die Lasermessdaten von beiden Lasermesssystemen nicht getrennt, sondern über einen UDP-Port in den Computer. Eine Aufspaltung in zwei CLASER-Threads würde nur eine Verdopplung der Verarbeitung bedeuten.

5.7.1 Schnittstellenbeschreibung nach außen

In der Header-Datei `laserFeeder.h` (siehe Listing 5.1) ist die Schnittstelle des neuen CLASERFEEDER-Objektes beschrieben. Es handelt sich hierbei um alle frei zugänglichen *public*-Methoden der vorherigen Schnittstelle. Nach „außen“ ergibt sich damit kein Unterschied, nur die interne Struktur hat sich geändert. Die Parametertypen der Methoden `GetLaserScanScanner()`,

```
class CRADIALSCAN {
  CRADIALSCAN (void);
  virtual ~CRADIALSCAN (void);
  CRADIALSCAN &operator= (const CRADIALSCAN &right);

  // scan data
  int _maxScans;
  int _numScans;
  float *_scanAngle;
  float *_scanDist;

  // mark data
  int _maxMarks;
  int _numMarks;
  float *_markAngle;
  float *_markDist;
};

class CRADIALSCANMATCHED : public CRADIALSCAN {
  CRADIALSCANMATCHED (void);
  virtual ~CRADIALSCANMATCHED (void);
  CRADIALSCANMATCHED &operator= (const CRADIALSCAN &right);
  CRADIALSCANMATCHED &operator= (const CRADIALSCANMATCHED &right);

  int *_markIdx;   // mark IDs from localisation
};
```

Listing 5.2: Auszug aus `radialscan.h`

```
class CLASERFEEDER : protected CTHREAD {
// ...
protected:
  CGENBASE *_genBase;
  int _numLaser;

  int fd;              // udp socket()
  unsigned char *buf;  // buffer for udp packets
  size_t buflen;

  double *_x0, *_y0, *_a0;
  unsigned *_alarm_level;
  unsigned *_platform_valid;
  CRADIALSCAN **_scanner;
  CRADIALSCANMATCHED **_platform;
  time_t *scantime;
  time_t *marktime;

  virtual void Fxn(void);
  void marks2matched(unsigned char *const buf, size_t l, int idx);
  void raw2radial(unsigned char *const buf, int idx);
  void updatePlatform(const int idx);
}
```

Listing 5.3: `protected`-Auszug aus `laserFeeder.h`

`GetLaserScanPlatform()` und `GetLaserScanPlatformMatched()` sorgen dafür, dass die Klassen CRADIALSCAN und CRADIALSCANMATCHED übernommen werden müssen. Der Unterschied zwischen den Methoden `GetLaserScanScanner()` und `GetLaserScanPlatform()` liegt darin, dass die Daten im ersten Fall in Lasermesssystemkoordinaten und im zweiten Fall in Roboter- bzw. Plattformkoordinaten übergeben werden. Die Schnittstellenbeschreibung dazu befindet sich in Listing 5.2. Die Typen `meter_t` und `radian_t` müssen wegen der Methode `GetScannerPosition()` ebenfalls erhalten bleiben. Sie sind in Header-Datei `units.h` als `typedef double meter_t` und `typedef double radian_t` definiert.

5.7.2 Private Daten und Methoden des CLASERFEEDER

Das CLASERFEEDER-Objekt hat zu den nach außen sichtbaren Methoden noch private Daten und Methoden. Der Zeiger `CGENBASE *_genBase` wird benötigt um die Lokalisation mit neuen Lasermessdaten zu versorgen. Liegt ein neuer Messdatensatz vor, so muss `_genBase->UpdateLaser()` aufgerufen werden. Die Anzahl der Lasermesssysteme wird in `int _numLaser` gespeichert. Sie ist dazu da, bei den *public*-Methoden die Lasermesssystem-ID `idx` zu überprüfen. Der Dateideskriptor für den UDP-Socket wird in `int fd` gespeichert. Die UDP-Daten landen im dem Puffer `unsigned char *buf` mit der Länge `size_t buflen`.

Pro Lasermesssystem gibt es die Variablen `double _x0`, `double _y0`, `double`, `_a0`, `unsigned _alarm_level`, `unsigned _platform_valid`, `time_t scantime`, `time_t marktime`, `CRADIALSCAN _scanner` und `CRADIALSCANMATCHED _platform`, die als Zeiger auf Arrays im CLASERFEEDER

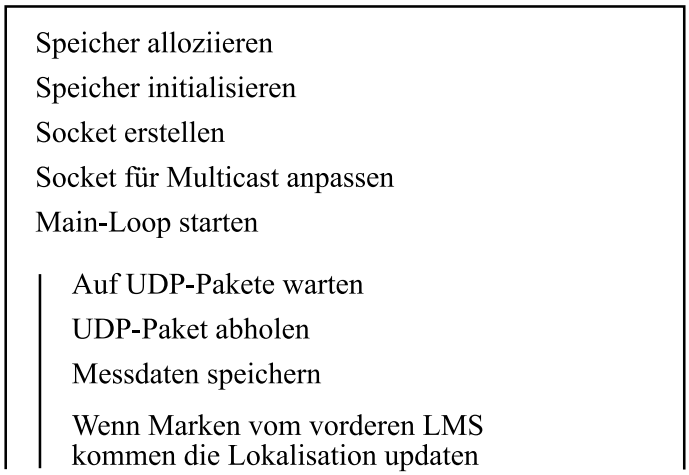


Abbildung 5.9: LaserFeeder-Thread der neuen Anbindung

hinterlegt werden. Auch hier ist `_numLaser` dazu da, die Arrayindizes zu überprüfen. Die Position des Lasermesssystems im Verhältnis zum Robotermittelpunkt wird in `_x0`, `_y0` und `_a0` gespeichert. Die Angaben sind in Metern bzw. im Bogenmaß. In `_alarm_level` wird der zuletzt übertragene Wert des Kollisionsschutzes des dazugehörigen Lasermesssystems gespeichert. `_scanner` nimmt die letzten Messdaten eines Lasermesssystems auf. `_scanner` ist ein CRADIALSCAN- und kein CRADIALSCANMATCHED-Objekt, weil das Array, welches die IDs der Marken speichert, von der Lokalisation kommt und diese nur in Plattformkoordinaten arbeitet. Diese werden in `_platform` zusammen mit den Entfernungsmessdaten in Roboterkoordinaten gespeichert. Die Entfernungsmessdaten in Roboterkoordinaten werden erst bei Bedarf berechnet, so dass in `_platform_valid` angegeben wird, ob die letzten Messdaten bereits umgerechnet wurden.

Die Funktion `virtual void Fxn(void)` wird von CTHREAD geerbt und beinhaltet den Code, der nach der Initialisierung ausgeführt werden soll. Dort ist der Main-Loop einzubinden. Die beiden Funktionen `void marks2matched(unsigned char *const buf, size_t l, int idx)` und `void raw2radial(unsigned char *const buf, int idx)` sind dazu da, die beiden verschiedenen Sorten von ankommenden Paketen mit den Lasermessdaten und den extrahierten Marken zu parsen und in die dafür vorgesehen Objekte `CRADIALSCAN` und `CRADIALSCANMATCHED` zu speichern. Die Funktion `void updatePlatform(const int idx)` wird intern verwendet, um die Daten von den Lasermesssystemkoordinaten bei Bedarf in Plattformkoordinaten zu transformieren. Werden die Plattformkoordinaten nicht benötigt, so kann auf eine Umrechnung verzichtet werden.

5.7.3 Threadinitialisierung

Die Initialisierung des CLASERFEEDER-Threads ist bei der neuen Implementierung kürzer, da wesentlich weniger gemacht werden muss. Die Lasermesssysteme müssen nicht mehr zurückgesetzt werden. Auf das Synchronisieren und Konfigurieren der Lasermesssystem wird auch verzichtet, da der Rabbit 3000 diese Aufgabe übernommen hat. Die Lasermesssysteme befinden sich bereits im Echtzeitmodus.

Zur Initialisierung müssen die privaten Zeiger aus dem CLASERFEEDER auf allozierten Speicher gesetzt werden und der Empfang von Multicast-UDP-Paketen aktiviert werden. Dazu muss mit `socket()` ein Verbindungsendpunkt erstellt werden. Diesem muss für Multicast die Option `SO_REUSEADDR` mit der Funktion `setsockopt()` zugewiesen werden. Mit `bind()` wird dieser Verbindungsendpunkt dann an einen Port gebunden um eingehende Pakete zu erhalten. Der Empfang von Multicast-Paketen erfolgt erst nach einem Beitreten zur Multicast-Gruppe 224.0.0.23 durch einen weiteren `setsockopt()`-Aufruf.

Am Ende der Initialisierung muss die geerbte Funktion `Start()` vom CTHREAD aufgerufen werden. Erst jetzt wird der neue Thread erstellt. Dieser neue Thread ruft anschließend die Funktion `Fxn()` vom CTHREAD auf, die in CLASERFEEDER überschrieben wurde. Der darin enthaltene Code wird dann nebenläufig zu den anderen Threads ausgeführt.

5.7.4 Datenempfang und -verarbeitung

Die Funktion `CLASERFEEDER::Fxn()` vom CLASERFEEDER-Thread beinhaltet den Code, der nach dem Erzeugen des Threads ausgeführt werden soll. Der Main-Loop besteht aus einer `while()`-Schleife, die so lange läuft, wie `_stopRequested` *false* ist. Im Main-Loop wird nur auf UDP-Pakete gewartet, die dann verarbeitet/gespeichert werden.

Zum Warten auf neue UDP-Pakete bietet es sich an, den Dateideskriptor vom Socket in ein `select()`-Aufruf mit aufzunehmen. Dieses `select()` blockiert so lange, bis Daten anliegen oder ein Fehler auftritt. Ein Fehler könnte beispielsweise das Schließen eines Sockets sein. Nach dem mit einem `ioctl()`-Aufruf geprüft wurde, wie viele Byte zur Verfügung stehen, kann bei Bedarf die Puffer mit einem `realloc()`-Aufruf vergrößert werden. Danach wird mit `recv()` vom Socket das nächste UDP-Paket abgeholt.

Es wird lediglich geprüft, ob das erste Byte des UDP-Paketes STX (0x02) ist. Auf ein Prüfen der CRC-Prüfsumme wird bewusst verzichtet, da der Rabbit 3000 dies bereits erledigt hat. Wie in Abschnitt 5.6.1 auf Seite 141 bereits erwähnt, müssen bei Telegrammen vom Rabbit 3000 zum Hostrechner die CRC-Prüfsummen nicht mehr korrekt sein, da die Adresse in den Telegrammen für das zweite Lasermesssystem geändert wurde.

Handelt es sich bei dem empfangenen Telegramm um Entfernungsmessdaten, wie sie die Lasermesssysteme verschicken, so wird die interne Funktion `raw2radial()` aufgerufen. Sie zerlegt das Paket in einzelne Entfernungsmesswerte und speichert sie in dem dazugehörigen CRADIAL-SCAN-Objekt. Auf ein Speichern der Marken bzw. der Reflexionswerte wird verzichtet, da der Rabbit 3000 hierfür gesonderte Telegramme verschickt. Als letztes wird der Zeitstempel, der zu den Messdaten gehört, aktualisiert. Ist ein Telegramm mit extrahierten Marken angekommen, so wird die interne Funktion `marks2matched()` aufgerufen. Sie speichert die gefundenen Marken sowohl in Scanner- als auch in Plattformkoordinaten. Eine Umrechnung in Plattformkoordinaten ist immer notwendig, da die Lokalisation nur mit diesen arbeitet. Auch hier wird der Zeitstempel aktualisiert.

Wenn ein Telegramm mit extrahierten Marken vom vorderen Lasermesssystem kommt, ist das ein guter Zeitpunkt, um die Lokalisation mit den neuen Marken zu aktualisieren. Zunächst wird die Gesamtanzahl an Marken von beiden Lasermesssystemen ermittelt. Danach kann das `matched`-Array in passender Größe erzeugt werden und mit minus eins initialisiert werden. Die

Marken selbst werden in einem CVEC-Objekt übergeben. Dieser Vektor hat zweimal so viele Komponenten wie es Marken gibt, da er abwechselnd mit der Entfernung in Millimetern und dem Winkel im Bogenmaß gefüllt wird. Anschließend wird die Lokalisation über das CGENBASE-Objekt mit dem CVEC-Objekt und dem `matched`-Array aufgerufen (`UpdateLaser()`). Nach dem Aufruf sind im `matched`-Array die IDs bei den Marken gesetzt, die in der Karte gefunden wurden. Die IDs aus dem Array werden im letzten Schritt im CRADIALSCANMATCHED in dem `_markIdx`-Array zu den Marken gespeichert, damit sie anderen Anwendungen zur Verfügung stehen.

5.7.5 Nebenläufige Abfragen am CLASERFEEDER-Objekt

Die nach Außen sichtbaren *public*-Methoden werden intern nicht verwendet. Sie sind ausschließlich für andere Threads gedacht, um Daten vom CLASERFEEDER-Objekt abzufragen. Die Methoden haben keine Seiteneffekte auf den Zustand des CLASERFEEDER-Objektes.

Die Methode `int GetNumScanners(void)` liefert die Anzahl der Lasermesssysteme zurück. Sie sollte damit immer zwei zurückgeben. Die Lasermesssysteme sind von null beginnend durchnummeriert. Bei allen anderen Methoden muss die entsprechende Nummer im Parameter `idx` übergeben werden. Die Methode `int GetScannerPosition(meter_t &x, meter_t &y, radian_t &a, const int idx)` liefert beispielsweise für das Lasermesssystem `idx` in Roboterkoordinaten die Position, an der es am Roboter montiert ist. Diese Information kann von Anwendungen benötigt werden, die die Laserstrahlpositionen zum Anzeigen oder Weiterverarbeiten benötigen.

Die Robotersteuerung muss sicherstellen, dass der Serviceroboter nicht gegen Hindernisse fährt. Nähert er sich einem Hindernis, so bietet es sich an, dass er etwas langsamer fährt. Die Motoransteuerung interessiert sich deshalb für den Abstand zum dichtesten Objekt in Fahrtrichtung. Dieser wird von `int GetClosestObstacleDistance(const int idx, double &distance)` zur Verfügung gestellt. Die Entfernung wird intern nicht immer berechnet. Liefert der Rabbit 3000 von seinen Bereichsradien keine Verletzung (`_alarm_value` ist Null), so wird einfach ein Abstand von einem Meter zurückgeliefert. Es wird auf die Berechnung der Entfernung verzichtet, da der Rabbit 3000 das bereits erledigt hat. Wenn beide Bereichsradien verletzt werden, sprich eine Kollision direkt bevorsteht (`_alarm_value` ist Zwei), dann wird eine Entfernung von 30 Zentimeter zurück geliefert. Ansonsten werden alle Messwerte in Roboterkoordinaten transformiert und der kleinste Wert zurückgegeben. Liegen keine aktuellen Messwerte vor, so wird von einer Verletzung beider Bereichsradien ausgegangen.

Die Lasermessdaten in unveränderter Form können über `int GetLaserScanScanner(const int idx, CRADIALSCAN &scan)` abgefragt werden. Sind die Daten im CLASERFEEDER-Objekt älter als ein bis zwei Sekunden, so wird ein leerer Messdatensatz zurückgeliefert. In der alten Anbindung gab es kein Alter für Messdatensätze.

Werden die Messdaten im Koordinatensystem des Serviceroboters, den Plattformkoordinaten, benötigt, so können sie über `int GetLaserScanPlatform(const int idx, CRADIALSCAN &scan)` oder `int GetLaserScanPlatformMatched(const int idx, CRADIALSCANMATCHED &scan)` abgefragt werden. Die zweite Methode liefert wie die erste die Entfernung- und Reflexionsmesswerte, zusätzlich aber noch ein Array mit den IDs der Marken, die von der Lokalisation erkannt wurden (*matched*). Bevor die Messdaten in das übergebene Objekt kopiert werden, wird ebenfalls das Alter der Messdaten überprüft. In Roboterkoordinaten stehen die Messdaten zunächst nicht zur

Verfügung. Ein interner Aufruf von `updatePlatform(idx)` sorgt dafür, dass das CRADIAL-SCANMATCHED-Objekt aktualisiert wird bevor die Messwerte bereitgestellt werden.

5.8 Zusammenfassung

In diesem Kapitel wurden die Entwicklung der Software auf dem entwickelten System und die Anpassungen der Software auf dem Hostrechner des Serviceroboters dargestellt. Die Software auf dem Rabbit 3000 kann die Lasermesssysteme steuern, die CRC-Prüfsumme der Telegramme auswerten und die Messdaten telegrammweise als UDP-Paket weiterleiten. Zusätzlich sind Vorverarbeitungsfunktionen implementiert, so dass zwei Bereichsradien um den Robotermittelpunkt auf Unterschreitung geprüft werden und eine Liste mit den Positionen von Reflektormarken erzeugt wird.

Die Software auf dem Hostrechner wurde dahingehend angepasst, dass die Messdaten über die Netzwerkschnittstelle empfangen werden und die Ergebnisse der schon im Rabbit 3000 durchgeführten Vorverarbeitung genutzt werden.

6 Leistungsanalyse

In diesem Kapitel wird die Leistungsfähigkeit der neuen Anbindung mit dem Rabbit Powercore 3800 analysiert. Zuerst wird gezeigt, dass es zu keinem Messdatenverlust kommt. Ein wesentliches Ziel der Neuentwicklung ist damit erreicht. Im zweiten Abschnitt geht es um die Prozessorauslastung des Hostrechners durch den Betrieb der Lasermesssysteme. Es werden Messergebnisse von der alten und neuen Anbindung miteinander verglichen. Im Lasermesssystem, dem Rabbit 3000 und im Hostrechner treten Latenzen bei der Verarbeitung von Messwerten auf. Im dritten Abschnitt werden diesbezüglich sowohl theoretische Berechnungen als auch praktische Messungen durchgeführt. Im letzten Abschnitt geht es um die Systemauslastung des Rabbit Powercore 3800 durch Vorverarbeitung. Die Ausführungszeiten der einzelnen Vorverarbeitungsfunktionen werden ermittelt.

6.1 Keine Messdatenverluste

Eine der wichtigsten Anforderung an das Eingebettete System ist, dass keine Messdaten der Lasermesssysteme beim Transport verloren gehen. Die Daten kommen über zwei RS-422-Schnittstellen in den Rabbit Powercore 3800 mit einer Baudrate von 500 kBd. Dort werden sie von der Interrupt-Service-Routine (ISR) abgearbeitet und in einem Puffer gespeichert. Aus diesem heraus werden die Telegramme per UDP verschickt und gegebenenfalls für die Weiterverarbeitung extrahiert. Im Hostrechner werden die Daten im Netzwerkpuffer abgelegt und bleiben dort zur Weiterverarbeitung liegen. Für jeden einzelnen Schritt muss überprüft werden, ob und auf welche Weise Messdaten verloren gehen können.

Auf jeder Datenleitung kann es durch äußere Einflüsse zu Störungen kommen. Auf der etwa einen Meter langen RS-422-Verbindung von den Lasermesssystemen zu dem Mikroprozessor ist das praktisch auszuschließen. Bei RS-422 wird eine Spannungsdifferenz an ein verdrilltes Kabel angelegt. Diese Art der Übertragung ist sehr robust.

Lasermesssysteme haben eine maximale Datenrate von je 50 kByte/s. Die ISR benötigt für den Empfang eines Byte maximal 279 Takte. Daraus folgt, dass der Prozessor bei einer Taktung von 51,6 MHz $\frac{51.600.000}{279} > 184.946$ Byte pro Sekunde empfangen kann. Die beiden seriellen Ports am Rabbit Powercore 3800 besitzen jeweils einen vier Byte großen FIFO-Puffer, welcher in dieser Anwendung nicht benötigt wird. Im Echtzeitmodus der Lasermesssysteme kommen jeweils 732 Byte 37,5 mal in der Sekunde an. Daraus ergibt sich eine Datenrate von $2 \cdot 732$ Byte $\cdot$ $37,5\ \mathrm{s}^{-1} = 54.900$ Byte/s für beide Lasermesssysteme zusammen. Von den $51,6 \cdot 10^6$ Takten pro Sekunden, die der Rabbit 3000 ausführt, werden $54.900 \cdot 279 = 15.317.100$ für die ISR verbraucht, das sind knapp 30 % der Prozessorleistung. Zwischen Lasermesssystem und Rabbit 3000 gehen somit keine Daten verloren.

Das Verschicken von Telegrammen aus dem Puffer des Rabbit Powercore 3800 erfolgt im Gegensatz zum Empfang innerhalb der ISR ohne Priorität. Ein Überlaufen des Puffers ist möglich, wenn die Telegramme nicht schnell genug versendet werden. Der Puffer hat eine Größe für acht Telegramme je Lasermesssystem. Messungen im normalen Betrieb haben ergeben, dass der Füllstand zwischen null und einem Telegramm schwankt.

Telegramme können jetzt nur noch auf der Ethernet-Verbindung zwischen dem Powercore 3800 und dem Hostrechner verloren gehen. Hierbei handelt es sich um eine 10 MBit Vollduplex-Verbindung mit einem gekreuzten Kabel. Da in dieser Situation keine Kollision auf der Twisted-Pair-Verkabelung auftreten kann, müsste der Linuxcomputer einzelne UDP-Pakete verlieren. Der TCP/IP-Stack unter Linux ist jedoch für hohe Durchsätze entworfen, so dass mit ziemlicher Sicherheit gesagt werden kann, dass eine Datenrate von unter 55 kByte/s keine Probleme bereitet.

Messungen über einen Zeitraum von fünf Stunden haben ergeben, dass kein einziges Byte verloren gegangen ist und keine fehlerhafte CRC-Prüfsumme bei Telegrammen aufgetaucht ist.

6.2 Benchmark-Ergebnisse

Zum Vergleichen der bisherigen Anbindung mit der Moxa-Schnittstellenkarte und der neuen Anbindung via Ethernet muss die Systemauslastung des Linuxcomputers ermittelt werden. In diesem Fall beschränkt sich die Auswertung auf die Prozessorlast. Ein Vergleich der Speichernutzung lohnt sich nicht, weil alte und neue Anbindung nur sehr wenig Speicher benötigen. Es werden nur die jeweils aktuellen Messdaten vorgehalten, die sich auf wenige KByte beschränken.

Zur Messung der Prozessorauslastung sind unter Linux Kommandozeilenprogramme wie `top` nur bedingt geeignet, da sie nur die momentane Auslastung eines Systems anzeigen. Das Programm `top` zeigt mit einem Aktualisierungsintervall von drei Sekunden die durchschnittliche Auslastung für Prozessor und Arbeitsspeicher an. Längere Durchschnittszeiten sind zwar theoretisch möglich, lassen sich aber nicht gut speichern.

In `/proc/stat` werden genauere Werte für die einzelnen Prozessoren in einem Linuxsystem ausgegeben. Das *proc*-Dateisystem ist ein Pseudo-Dateisystem, das eine Schnittstelle zu Datenstrukturen im Linuxkernel bereitstellt. Die erste Zeile in `/proc/stat` lautet beispielsweise `cpu  1922 0 1024 19817`. Die Einheit sind Jiffy, die in diesem Fall 1/100 Sekunde lang sind. Die erste Zahl gibt die Zeit an, die der Prozessor für Programme im *user mode* gebraucht hat. Der zweite Wert ist die Zeit für Programme mit geringer Priorität, d.h. welche die mit einem *nice level* größer als null laufen. Der dritte Wert ist die Zeit, die der Prozessor im *system mode* verbracht hat, d.h. mit dem Ausführen von Kernel-Code beschäftigt war. Der vierte Wert ist die Zeit im *idle mode*, das ist die Zeit, die der Prozessor im Leerlauf war. Bei neueren Kernelversion (ab Linux 2.6) gibt es noch weiter Angaben wie *iowait*, *irq* und *softirq*. Die angegebenen Werte beziehen sich auf die vom Systemstart an verbrachte Zeit in den einzelnen Modi. Weitere Informationen zum *proc*-Dateisystem sind in der *man page* `PROC(5)` auf nahezu jedem Linuxsystem zu finden.

Wird das SuSE-Linux-System auf dem Roboter gebootet und keine zusätzliche Software manuell gestartet, kann eine Messung der Systemauslastung im „Ruhezustand“ gemacht werden. Innerhalb einer Minute ändern sich die Werte in `/proc/stat` dabei wie folgt: `23 0 2 5977`. Das

System lief 6002 Jiffies, davon 23 im *user mode* und zwei im *system mode*. Das sind 0,385 % im *user* und 0,033 % *system mode*.

Soll die Prozessorauslastung der unterschiedlichen Anbindung miteinander verglichen werden, so bietet es sich an, möglichst viel der Robotersteuerung aus der Prozessorauslastung heraus zu rechnen. Startet der `mobiled` ohne angeschlossene Lasermesssysteme, so kommen keine Messdaten im Hostrechner an, die gespeichert und verarbeitet werden müssen. Die Prozessorauslastung, die jetzt im Computer vorhanden ist, muss deshalb von den Messergebnissen der jeweiligen Anbindungen abgezogen werden um bessere Vergleichswerte zu erhalten. Im „Leerlauf“ des `mobiled`, d.h. ohne Lasermesssysteme und ohne Bewegung des Roboters, liegt die Prozessorlast bei 23,150 % im *user mode* und bei 2,766 % im *system mode*.

6.2.1 Bisherige Anbindung: Moxa-Karte

Werden die Lasermesssysteme über die Moxa-Karte angeschlossen, so steigt die Prozessorlast auf 75,654 % (davon 23,379 % *system* und 52,275 % *user*). Nach Abzug der Messwerten für die „Leerlauf“-Last des `mobiled` ergibt sich ein Prozessorauslastung von 20,613 % im *system mode* für die Moxa-Kerneltreiber und eine Auslastung von 29,125 % im *user mode* für die Anbindung der Lasermesssysteme.

Werden die Lasermesssysteme über die Moxa-Karte angebunden, so entstehen Telegrammverlustraten von 36,2 % bzw. 40,8 % (siehe 2.5.3). Die Telegramme werden nur für die Lokalisation verarbeitet, wenn es von beiden Lasermesssystemen jeweils einen Datensatz gibt, der nicht älter als 40 ms ist. Die Telegramme kommen von jedem Lasermesssystem in einem Abstand von 26,64 ms an. Der Versatz zwischen den beiden Lasermesssystemen kann also maximal 13,32 ms sein, so dass in die letzten 40 ms von jedem Lasermesssystem zeitlich zwei Messdatensätze passen können, aber nicht müssen. In Abbildung 6.1 sind zwei 40 ms Zeitfenster und die Telegramme auf einer Zeitskala abgebildet.

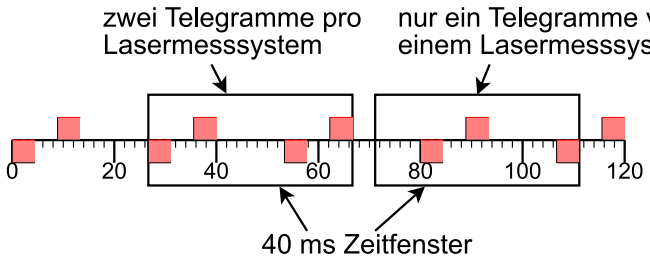


Abbildung 6.1: Zeitfenster für Telegrammverarbeitung

Je nachdem, wie die Telegrammverluste bei den beiden Lasermesssystemen zusammenfallen, können im günstigsten Fall 59,2 % ankommen. Das ist genau dann der Fall, wenn die Telegrammverluste zur „gleichen“ Zeit auftreten. Im ungünstigsten Fall liegen sie so, dass zuerst das eine Lasermesssystem keine Daten mehr liefert und danach das andere. Die Verlustraten würden sich dann zu 77 % addieren. Ein Zeitdiagramm dazu findet sich in Abbildung 6.2. Im oberen Zeitdiagramm fallen die Verluste zusammen. Im unteren liegen sie teilweise versetzt, so dass nicht alle Telegramme in die Lokalisation einfließen. Die Verlustraten der Telegramme beruhen

auf den Verlusten der einzelnen Byte in den Telegrammen. Da diese von beiden Lasermesssystemen durch den Kerneltreiber stark voneinander abhängig sind, ist von dem günstigeren Fall von 59,2 % kompletter Telegramme auszugehen.

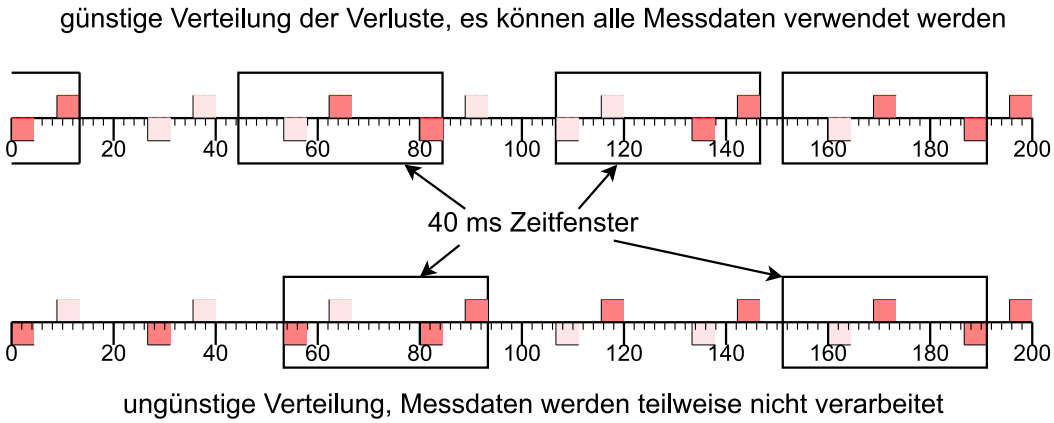


Abbildung 6.2: Verteilung der Telegrammverluste

Wenn statt 59,2 % der Telegramme alle 100 % ankommen, ist mit einer höheren Prozessorauslastung zu rechnen. Die meisten Codeteile des `mobiled` benötigen proportional viel Rechenzeit zur Anzahl der Telegramme. Der Code ist überwiegend blockierend und wartet auf ganze Telegramme. Einzig der Codeteil, in dem nach neuen Telegrammanfängen im seriellen Datenstrom gesucht und der CRC geprüft wird, ist nicht proportional vom Rechenaufwand her. In diesen Rechnungen wird der Einfachheit halber dieser Rechenaufwand auch als proportional betrachtet, da er im Verhältnis zu der Lokalisation mit einem Kalman-Filter und der Umrechnung der Koordinaten in Roboterkoordinaten kaum ins Gewicht fallen dürfte.

Der Code aus dem Moxa-Kernelmodul verbraucht genauso soviel Rechenzeit, unabhängig davon, wie viele Telegramme verloren gehen. Die *system mode* Prozessorlast steigt ausschließlich um die fehlenden 0,135 %, die durch die Byteverlustrate entstehen. Der Einfachheit halber wird die Prozessorlast im *system mode* deshalb als konstant angesehen.

Aus den oben genannten 29,125 % Prozessorlast im *user mode* für 59,2 % der empfangenen Telegramme wird dann 49,198 % für 100 % der Telegramme. Die Prozessorlast im *system mode* bleibt bei 20,613 % unverändert.

6.2.2 Neue Anbindung: Powercore 3800

Bei der neuen Anbindung über die Ethernet-Schnittstelle kommen die Messdaten in UDP-Paketen in den Linuxcomputer. Es ist davon auszugehen, dass praktisch keine Prozessorauslastung im *system mode* vorhanden ist, da die Behandlung von UDP-Paketen hoch optimiert ist und die Datenrate relativ gering ist. Der Code beschränkt sich auf die Abarbeitung von UDP-Paketen und das Aufrufen der Lokalisation. Da nur die benötigten Daten für die Lokalisation berechnet werden und alle anderen nur bei Bedarf, ist hier mit einer geringeren Prozessorauslastung zu rechnen, als bei der alten Anbindung mit der Moxa-Karte. Die Auslastung liegt

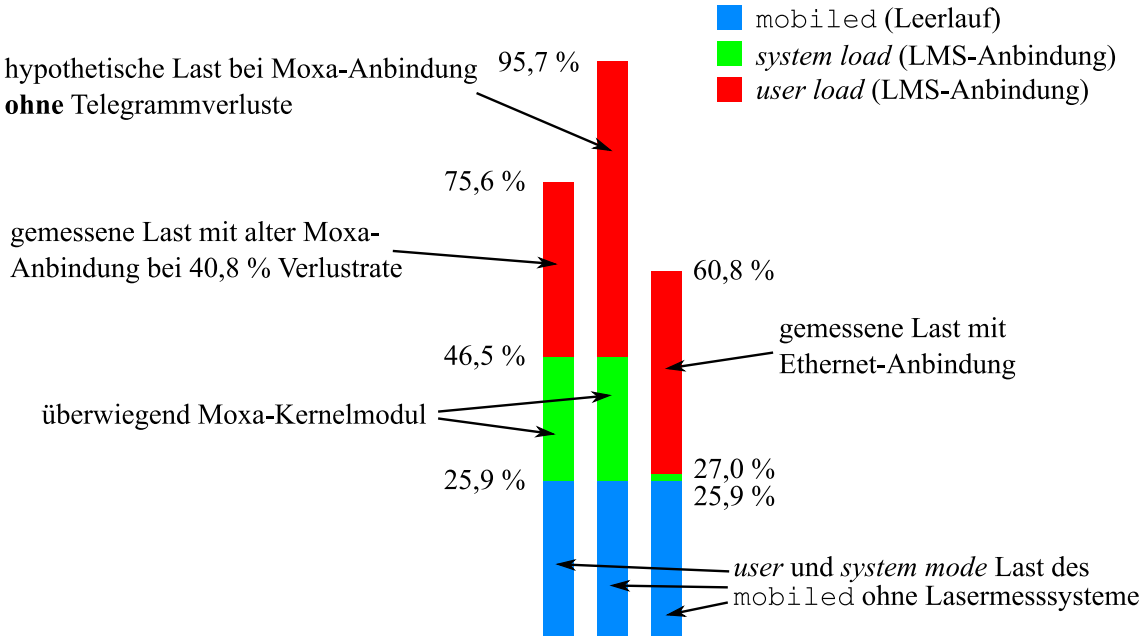


Abbildung 6.3: Systemauslastungsvergleich zwischen Moxa-Karte und Rabbit Powercore 3800

insgesamt bei 60,80 %, genaugenommen 56,92 % im *user mode* und 3,88 % im *system mode*. Unter Berücksichtigung der „Leerlauf"-Leistung des `mobiled` ergibt sich 33,77 % im *user mode* und 1,12 % im *system mode*. Die neue Anbindung verbraucht also 34,89 % der Prozessorauslastung für den Empfang der Lasermessdaten und die Lokalisation.

In Abbildung 6.3 ist auf der linken Seite die Prozessorauslastung für die alte Anbindung mit Moxa-Karte dargestellt. Der blaue Teil der Prozessorauslastung entsteht bei laufendem `mobiled` wenn keine Daten über die serielle Schnittstelle der Lasermesssysteme kommen. Die Prozessorlast im *user* und *system mode* ist dabei zusammengefasst angezeigt und liegt bei 25,9 %. Der `mobiled` befindet sich sozusagen im „Leerlauf". Sobald die Lasermesssysteme angeschlossen werden und Daten senden, steigt die Prozessorlast an. Der grüner Balken ist der Teil der *system load*, um die die Prozessorauslastung dabei steigt. Die im *user mode* höhere Last wird im roten Balken dargestellt.

Die Säule in der Mitte von Abbildung 6.3 zeigt ebenfalls die Prozessorauslastung mit der alten Anbindung über die Moxa-Schnittstellenkarte. Die Prozessorlast im *user mode* wurde hier jedoch so hochgerechnet, als ob keine Telegramme verloren gegangen wären. Die Auslastung im *system mode* bleibt aus oben genannten Gründen bei der Hochrechnung unverändert.

Die rechte Säule spiegelt die gemessene Prozessorauslastung mit der neuen Anbindung über Ethernet wider. Die neue Anbindung lässt die Last im *system mode* für 1,1 % und im *user mode* um 33,8 % steigen.

6.2.3 Fazit

Beim Vergleichen von alter und neuer Anbindung ist zunächst einmal die stark gesunkene Prozessorlast im *system mode* festzuhalten. Als Ursache lässt sich sofort das Moxa-Kernelmodul ausmachen, welches den Hauptanteil der Prozessorlast verursacht, vgl. 2.5.3.

Beim reinen Vergleich der Last im *user mode* von alter und neuer Anbindung ist festzustellen, dass diese von 29,1 % auf 33,8 % gestiegen ist. Unter Berücksichtigung der verloren gegangenen Telegramme ist sie dagegen von 49,2 % auf 33,8 % gesunken. Folgende Gründe sind dafür verantwortlich:

- Die Markenextraktion für die Lokalisation findet bereits im Rabbit 3000 statt.
- Es werden nur die Messdaten in Roboterkoordinaten umgerechnet, die auch benötigt werden. Bei der Anbindung über die Moxa-Karte wurden stets alle Messdaten umgerechnet.
- Es wird nur ein Thread verwendet. Vorher mussten die drei Threads sich synchronisieren.
- Die Abfragen der Messdaten von anderen Threads können parallel zum Schreiben der Messdaten stattfinden. Die Objekte werden zum Schreiben der neuen Daten nicht gelockt. Wird währenddessen gelesen, so kann es vorkommen, dass einzelne Lasermessergebnisse von dem vorhergehenden Messdurchlauf sind. Da aber sowieso keinerlei Garantie dafür übernommen wird, dass es sich um einen neuen Datensatz handelt, spielt das keine Rolle. Die Lokalisation wird aus dem gleichen Thread aufgerufen der auch die Messdaten speichert, so dass gewährleistet ist, dass sie jeden Messdatensatz genau einmal bekommt.

6.3 Verzögerung der Messwerte

Durch die neue Anbindung mit dem Powercore 3800 befindet sich ein zusätzliches Gerät zwischen den Lasermesssystemen und dem `mobiled`. Es gibt zwei Möglichkeiten, die Verzögerung zu ermitteln, die bei der Datenübertragung zwischen den Lasermesssystemen und dem `mobiled` entstehen. Die eine Möglichkeit ist das Berechnen der Verzögerung, die andere das Messen der Verzögerung von Telegrammen.

Das LMS 200 benötigt zum Messen mit einer Winkelauflösung von 0,5° zwei Spiegelrotationen, eine dauert 13,32 ms. Während dieser Zeit sammelt es die Messwerte im Datenpuffer. Am Ende der zweiten Spiegelrotation werden die Gesamtergebnisse an den Ausgabepuffer übergeben. Bis hierher sind 26,64 ms vergangen. Pro Spiegelrotation werden maximal 508 Byte ausgegeben (siehe [SickTL, S. 20, 123]). Daraus ergibt sich eine theoretische Datenrate von $\frac{508\ \text{Byte}}{0{,}01332\ \text{s}} \approx 38.138 \frac{\text{Byte}}{\text{s}}$. Ein Datentelegramm ist 732 Byte lang. Das LMS 200 benötigt damit $\frac{732}{38.138} \approx 19,19$ ms für das Verschicken des ganzen Telegramms. Je nachdem, wann ein Objekt im Sichtfeld des Lasermesssystems auftaucht, liegt die vergangene Zeit zwischen 19,19 ms (reine Versendezeit des Telegramms) und $26,64\ \text{ms} + 19,19\ \text{ms} = 45,83$ ms (zwei Spiegelrotationen plus Versendezeit). Genauere Informationen, wann das Versenden eines Telegramms beginnt, ist der Dokumentation nicht zu entnehmen.

Der Rabbit 3000 muss normalerweise einen Füllstand von maximal einem Telegramm haben, damit der Puffer nicht überlaufen kann. Sekündlich erscheinende Debug-Ausgaben im Rabbit 3000 Code liefern als Füllstand null oder ein Telegramm. Da alle 26,64 ms ein neues Telegramm eintrifft, kann es offenbar nicht länger als 26,64 ms im Rabbit 3000 verweilen. Die Übertragungszeit

auf der Ethernet-Verbindung kann mit ein paar Millisekunden angenommen werden, da es sich um eine lokale Verbindung handelt.

Die Zeitspanne zwischen dem Eintreten eines Objekts in das Sichtfeld des Lasermesssystems und dem Eintreffen der Messdaten im Hostrechner liegt damit zwischen ca. 20 ms und $45,83$ ms + $26,64$ ms + 3 ms = $75,47$ ms.

Alternativ zur reinen Berechnung der Verzögerungszeit können Messungen durchgeführt werden. Es ist die Zeit zwischen der Verletzung des Sichtfeldes eines Lasermesssystems und der Zeit, zu der das Telegramm im Linuxcomputer ankommt, zu bestimmen. Zur Messung wurde ein Gegenstand durch das Sichtfeld des Lasermesssystems auf einen Kontakt fallen gelassen. Der Gegenstand wurde in einer Höhe von 88 Zentimetern losgelassen und trat nach 67 Zentimeter Fallstrecke in das Sichtfeld des Lasermesssystems ein und löste nach weiteren 21 Zentimetern einen Kontakt aus. Die Formel für den freien Fall lautet:

$$h = h_0 - \frac{1}{2}gt^2 \tag{6.1}$$

Hierbei ist h die Höhe, h_0 die Ausgangshöhe von 0,88 m, g die Erdbeschleunigung von $9,81$ m/s^2 und t die Fallzeit. Aufgelöst nach t folgt:

$$t = \sqrt{\frac{2 \cdot (h_0 - h)}{g}} \tag{6.2}$$

Gesucht ist die Zeit zwischen dem Eintritt des Objekts ins Sichtfeld des Lasermesssystems und dem Auslösen des Kontakts. Dazu wird die Zeitdifferenz zwischen Gesamtfallzeit ($h = 0$ m) und Fallzeit bis zum Sichtfeld des Lasermesssystems ($h = 0,21$ m) gebildet.

$$\sqrt{\frac{2 \cdot (0,88 \text{ m} - 0 \text{ m})}{9,81 \ \frac{\text{m}}{\text{s}^2}}} - \sqrt{\frac{2 \cdot (0,88 \text{ m} - 0,21 \text{ m})}{9,81 \ \frac{\text{m}}{\text{s}^2}}} \approx 0,4236 \text{ s} - 0,3696 \text{ s} = 54,0 \text{ ms}. \tag{6.3}$$

Gemessen wurde die Zeit zwischen dem Auslösen des Kontaktes und dem Empfang des UDP-Paketes. In fast allen Fällen kam das UDP-Paket im Hostrechner an, bevor der Kontakt ausgelöst wurde. Die Messwerte sind deswegen fast immer negative Zeiten. Es kam einmal vor, dass der Kontakt unter 0,5 ms vor dem Empfang des das dazugehörige UDP-Paket ausgelöst wurde. Da aber mit einer Genauigkeit von Millisekunden gerechnet wird, sind in diesem Falle alle Messwerte kleiner gleich null. Die 67 Messwerte liegen in dem Bereich von -37 ms bis 0 ms.

Der Durchschnittswert ist

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i = \frac{1}{67} \cdot (-1158) \approx \ = -17,3 \text{ ms} \tag{6.4}$$

mit einer Standardabweichung (oder durchschnittliche Fehler der Einzelmessung) von

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N} (x_i - \overline{x})^2} = \sqrt{\frac{1}{66} \cdot 4705,61} \approx 8,44 \text{ ms}. \tag{6.5}$$

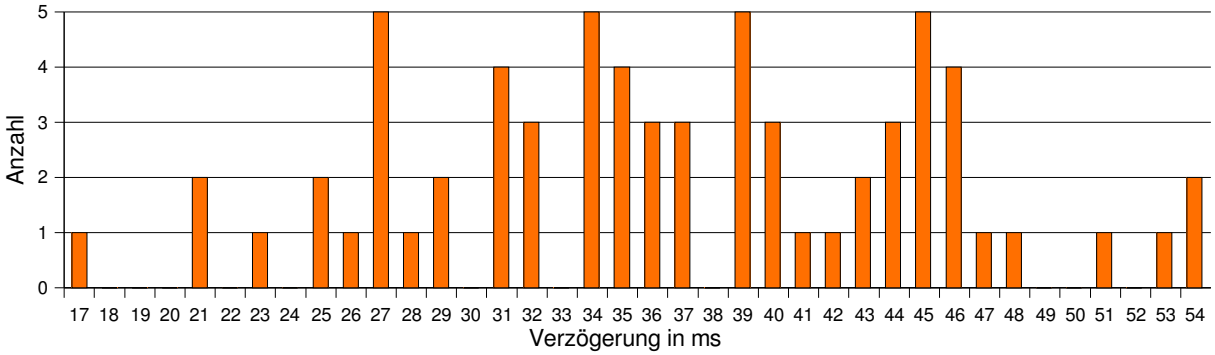


Abbildung 6.4: Zu sehen sind die Verzögerungszeiten bereinigt um die Fallzeitdifferenz von 54 ms

Die durchschnittliche Verzögerung von den Telegrammen ist $54\text{ ms} + (-17,3\text{ ms}) = 36,7\text{ ms}$ und liegt damit in der oben theoretisch ermittelten Zeitspanne von 20 bis 75 ms. Der Messwert -37 ms ergibt eine Verzögerung von 17 ms. Diese Verzögerung ist theoretisch nicht möglich. Folgende Fehlerquellen wurden bei dieser Messung noch nicht berücksichtigt:

- Wird das Objekt auch nur ein paar Zentimeter weiter oben oder unten losgelassen, so entstehen bereits Schwankungen um mehrere Millisekunden.
- Als Auslösekontakt wurde eine PS/2-Tastatur verwendet. Diese sendet pro Tastendruck einen Scancode über eine synchrone serielle Verbindung an den Computer. Dabei treten Verzögerungen auf, die hier nicht berücksichtigt wurden.
- Der Scheduler vom Linux-Betriebssystem wechselt nicht vorhersagbar zwischen den auszuführenden Tasks. Es entstehen zusätzliche unvorhersagbare Verzögerungszeiten im Bereich einiger Millisekunden.

In Abbildung 6.4 ist das Histogramm der einzelnen Messungen zu sehen. Die Werte wurden um die berechneten 54 ms korrigiert, so dass die Verzögerungszeiten zu sehen sind.

Die Standardabweichung von 8,44 ms hängt von der Spiegelrotationsgeschwindigkeit (13,32 ms), von der Scheduling-Latenz (10 ms) und von der Messgenauigkeit ab.

Eine Verzögerung von 36,7 ms ist für den Anwendungsfall ausreichend. Der Kollisionsschutz des Roboters ist bei derart kurzen Erkennungszeiten vollständig gegeben. Durch das Lasermesssystem werden bereits Verzögerungen von durchschnittlich 32,51 ms (19,19 ms bis 45,83 ms) verursacht. Die Verzögerung durch die Vorverarbeitung und das Versenden über Ethernet benötigt nach diesen Messungen nicht einmal 5 ms. Ziel dieser Messungen ist nicht die Ermittlung eines genauen Wertes, sondern eine Überprüfung der Größenordnung.

6.4 Auslastung durch Vorverarbeitung von Messdaten

Die Messdatenvorverarbeitung benötigt Rechenzeit. Es kann nur soviel Vorverarbeitung im Rabbit 3000 implementiert werden, wie CPU-Ressourcen zur Verfügung stehen. Der Empfang der Telegramme von den Lasermesssystem und der Versand als UDP-Paket müssen auf dem Rabbit 3000 gewährleistet sein.

Jedes Lasermesssystem sendet alle 26,64 ms ein Telegramm, in dieser Zeit sind damit zwei zu verarbeiten. Pro Telegramm stehen dann nur 13,32 ms zur Verfügung, damit es zu keinem Stau kommt. Die Bedienzeit muss nämlich kürzer sein als die Ankunftsabstände der Telegramme. Temporäre Verzögerungen können dabei mit einem Puffer ausgeglichen werden.

Weiter oben in diesem Kapitel (Abschnitt 6.1) wird eine Prozessorauslastung von knapp 30 % für den reinen Empfang der Telegramme von den beiden Lasermesssystem ermittelt. Da die Telegramme in Schüben eintreffen, ist die Prozessorauslastung zeitweilig bei 41,2 %, nämlich genau dann, wenn von beiden Lasermesssystemen Telegrammdaten ankommen. Die Prozessorauslastung wurde hierbei rechnerisch aufgrund des Durchsatzes von $2 \cdot 38.138$ Byte/s und einem Rechenaufwand von 279 Takten pro empfangenen Byte ermittelt.

Zum Verschicken von UDP-Paketen wird auf die Dynamic-C-Bibliothek zurückgegriffen. Da die Anzahl der Takte sich nicht direkt ermitteln lässt, muss der Ressourcenverbrauch durch Messungen bestimmt werden. Hier bieten sich zwei Messmöglichkeiten an. Der Rabbit 3000 sendet ununterbrochen mit einem Testprogramm UDP-Pakete mit 732 Byte Daten an den Hostrechner, der den Durchsatz und damit die Dauer für den Versand eines einzelnen Paketes berechnen kann. Messungen an der Ethernet-Schnittstelle zeigen, dass in 13,81 Sekunden 14336 UDP-Pakete eingetroffen sind. Pro UDP-Paket hat der Rabbit 3000 damit 0,963 Millisekunden benötigt. Im Rabbit 3000 kann mit dem Timer B eine Zeitmessung durchgeführt werden. Die so ermittelte Zeit zum Versenden eines UDP-Pakets beträgt 0,937 Millisekunden. Für jedes Lasermesssystem muss alle 26,64 Millisekunden ein UDP-Paket verschickt werden. Es entsteht eine Prozessorauslastung von knapp 4 % pro Lasermesssystem.

Ohne Messdatenvorverarbeitung wird der Prozessor bereits mit knapp 38 % für den Empfang (30 %) und das Versenden ($2 \cdot 4$ %) ausgelastet. Eine Vorverarbeitung von Messdaten ist damit nur eingeschränkt möglich. Weitere Zeitmessungen wurden für die Vorverarbeitung während des Betriebs gemacht. Das bedeutet, dass die Prozessorauslastung durch die ISR für das Empfangen von Telegrammen mit in die Zeitmessung eingeht.

Längere Zeiten als mit dem Timer B lassen sich mit der Echtzeituhr messen. Die Echtzeituhr liefert die aktuelle Zeit in Sekunden. Es stehen 48 Bit zur Verfügung, von denen 15 Bit für die Nachkommastelle verwendet werden. Die Genauigkeit liegt damit bei ungefähr 30 Mikrosekunden (1/32768 Sekunde).

Wird in der Vorverarbeitung nur die Kollisionswarnung aktiviert, so dauert die Berechnung des sogenannten Alarm-Levels pro Telegramm sechs bis sieben Millisekunden. Dieser Wert ist abhängig davon, was für einen zeitlichen Versatz die beiden Lasermesssysteme mit dem Verschicken ihrer Telegramme haben. Die ISR unterbricht die Berechnung je nach Lasermesssystem dann unterschiedlich stark.

Für die Ermittlung der Marken in der Vorverarbeitung werden pro Paket knapp fünf Millisekunden benötigt. Bei dieser Messung spielte es keine Rolle, ob keine oder ungefähr zehn Marken im Sichtfeld des Lasermesssystems waren. Mit vielen Marken im Sichtfeld lässt sich die Zeit aber künstlich erhöhen.

Zeitmessungen mit Berechnung des Alarm-Levels für die Kollisionswarnung und Ermittlung der Marken ergeben Zeiten von sechs bis zwölf Millisekunden pro Paket. Sechs Millisekunden werden benötigt, wenn das Lasermesssystem abgedeckt ist und damit keine Marken sehen kann und sofort einen Alarm-Level von 2 hat. Zwölf Millisekunden werden mit Hilfe von Reflektorband

erzeugt, das so vor das Lasermesssystem gehalten wird, dass viele „Marken“ im Sichtfeld sind. Die Kombination der Algorithmen zur Marken- und des Alarm-Level-Berechnung arbeitet wesentlich schneller als eine Nacheinanderausführung der Einzelalgorithmen. Erklären lässt sich das mit der Art, wie die Entfernung aus dem Telegramm berechnet wird. Die Entfernung ist über zwei Byte verteilt und muss mit einer Maske ausgeschnitten werden. Danach muss der ermittelte Entfernungswert auf Gültig geprüft werden, da einige Entfernungswerte zum Codieren von Fehlern verwendet werden. Für Markenberechnung und Alarm-Level muss die Operationen nur einmal je Messwert durchgeführt werden.

Für ein Telegramm darf die Verarbeitungszeit im Durchschnitt die 13,32 Millisekunden nicht überschreiten. Sollte das aber der Fall sein, so ist die Ankunftsrate der Telegramme höher als die Bedienrate. Es kommt dann zum Pufferüberlauf in der ISR. Um einen möglichst gleichmäßigen Durchsatz an Telegrammen zu erreichen, ist es notwendig, die Telegrammverarbeitungszeit inklusive Empfang, Vorverarbeitung und Versenden unter 13 Millisekunden zu halten. Der Rabbit 3000 ist mit der bereits implementierten Vorverarbeitung also voll ausgelastet.

6.5 Zusammenfassung

In diesem Kapitel wurde gezeigt, dass bei der neuen Anbindung keine Messdatentelegramme verloren gehen. Bei der bisherigen Anbindung trat ein Verlust von 36,2 % bzw. 40,8 % der Telegramme auf. Die Prozessorauslastung ist trotz der höheren Anzahl an Telegrammen insgesamt gesunken (von 75,6 % auf 60,8 %). Aufgrund der höheren Anzahl an zu verarbeitenden Telegrammen ist die Verarbeitungszeit zwar gestiegen (von 29,1 % auf 33,8 %), jedoch wird durch die Wahl der Netzwerkschnittstelle die Systemlast stark reduziert (von 20,6 % auf 1,1 %). Die Verzögerung durch die Integration des Eingebetteten Systems liegt laut Messungen bei weniger als 5 ms. Das Lasermesssystem benötigt alleine 26,64 ms für einen Messdatensatz, so dass die zusätzliche Verzögerungszeit zu vernachlässigen ist. Durch die Vorverarbeitung zur Reflektormarkenextraktion und zur Kollisionsvermeidung ist der Rabbit-3000-Mikroprozessor momentan nahezu ausgelastet.

Die neue Anbindung der SICK-Lasermesssysteme hat damit zu starken Verbesserungen geführt. Die freigewordene Prozessorzeit wird bereits im Rahmen anderer Projekte in Anspruch genommen.

7 Ausblick und weitere Entwicklungsmöglichkeiten

In diesem abschließenden Kapitel sollen weitere Entwicklungsmöglichkeiten aufgezeigt werden. Hierbei werden sowohl Weiterentwicklungen des Systems mit unveränderter Hardware betrachtet als auch Realisierungen auf anderer Hardware. Bei der Einschätzung, ob eine Weiterentwicklung sinnvoll ist, muss auch bedacht werden, dass es sich bei dem System primär um einen Schnittstellenumsetzer mit einigen Vorverarbeitungsfunktionen handelt. Diese Funktionalität ist in dem erreichten Stand bereits realisiert. Weitere Algorithmen der Vorverarbeitung erfordern deutlich mehr Rechenleistung. Wird berücksichtigt, dass zusätzlich auf dem Serviceroboter ein Pentium-4-Rechner installiert ist, erscheint es wenig sinnvoll, weitere höhere Vorverarbeitungsfunktionen auf der Seite des Rabbit Powercore 3800 zu implementieren, da die Performance deutlich geringer als auf dem Pentium-4-Rechner sein wird. Zudem darf unter keinen Umständen die Systemstabilität des Powercore 3800 und die zuverlässige Weiterleitung der Daten gefährdet werden. Dennoch sollen einige Weiterentwicklungsmöglichkeiten, die realisiert werden könnten, erörtert werden.

7.1 Integration in weitere Systeme

Die Leistung des Serviceroboters konnte mit dem Einsatz der neuen Anbindung deutlich verbessert werden. Ansatz für weitere Entwicklungsmöglichkeiten könnte die Integration des Systems in weitere Anwendungsgebiete der Lasermesssysteme darstellen. Zunächst wären geeignete Projekte auszuwählen. Die bisherige Anbindung müsste analysiert und die Vor- und Nachteile der Anwendung des entwickelten Systems müssten abgewogen werden. Eventuell wären Anpassungen an andere Versionen und Betriebsmodi der Lasermesssysteme notwendig. Ebenfalls könnten andere einfache Vorverarbeitungsfunktionen implementiert werden, die auf den jeweiligen Anwendungstyp zugeschnitten wären.

Ein mögliches Einsatzgebiet der entwickelten Anbindung ist das Erzeugen von dreidimensionalen Abbildungen der Konturen eines Raumes oder einer Szene mittels beweglich montierter Messsysteme. Denkbar wäre eine solche Erweiterung auch für den Serviceroboter des Arbeitsbereichs TAMS. Der Rabbit Powercore 3800 könnte hierbei ebenfalls für die Steuerung der Komponenten zur Bewegung der Messsysteme dienen. Der Antrieb ließe sich beispielsweise mit Modellbauservos realisieren, die an den pulsbreitenmodulierten Ausgängen des Powercore 3800 betrieben werden könnten. Es könnten ebenfalls optische Encoder zur Bestimmung des tatsächlichen Neigungswinkels verwendet werden. Der bekannte Neigungswinkel der Messsysteme könnte dann in den Messdatentelegrammen ebenfalls mitgespeichert werden, so dass die spätere Rekonstruktion der Szene ermöglicht wird.

Eine weitere Integrationsmöglichkeit besteht auch bei Anwendungsgebieten, in denen die Lasermesssysteme stationär betrieben werden. Der Einsatzzweck kann beispielsweise in der Überwachung eines Objekts liegen. Die Daten wären dann über Netzwerk abrufbar. Denkbar ist auch die kabellose Übertragung der Daten über einen WLAN-Access-Point mit entsprechender Funktionalität. Auf diese Art könnten bei diesen Anwendungen der Verkabelungsaufwand und damit die Kosten stark gesenkt werden.

7.2 Netzwerksicherheit

Bei der aktuellen Implementierung des Systems zur Ansteuerung zweier SICK LMS 200 erfolgt die Datenübertragung zum Hostrechner unverschlüsselt in Form von UDP-Paketen. Für den Fall, dass sich sowohl das System als auch der Hostrechner in einem vertrauenswürdigem Subnetz befinden, stellt dies kein Problem dar. Sobald die Kommunikation subnetzübergreifend erfolgt, müssen folgende Punkte bedacht werden:

- Messdaten können mitgelesen werden.
- Gefälschte Messdatentelegramme können eingeschleust werden.
- Ein Netzwerkangriff auf das System kann durchgeführt werden.

Werden anhand der Messdaten Bewegungen gesteuert, so ist die Integrität der Daten von besonderer Bedeutung, da fehlerhafte Daten zu Kollisionen führen könnten. Wenn ein Angreifer ein UDP-Paket mit falschen Netzwerkdaten und gefälschter Absenderadresse generiert und einschleust, könnte dieses im Hostrechnersystem nicht von echten Messwerten unterschieden werden. Aus diesem Grund besteht eine weitere Entwicklungsmöglichkeit darin, eine Integritätssicherung der Pakete zwischen dem System und dem Hostrechner zu implementieren. Hierbei wäre ein geeignetes kryptographisches Verfahren zu wählen und zu untersuchen, inwiefern die Performance des Rabbit Powercore 3800 hierfür ausreichend ist.

Eine weitere Entwicklungsmöglichkeit, die in das Gebiet Netzwerksicherheit fällt, ergibt sich aus folgendem Sachverhalt: Bei Tests reagiert der Rabbit Powercore 3800 auf Überflutung mit zufälligen Netzwerkdaten mit einem Schließen oder mit einem Absturz des Sockets. Die Ursache hierfür ist nicht einsichtig, da der Socket in vorgefertigten Bibliotheken implementiert ist. Das Hauptprogramm läuft hingegen weiter, jedoch mit der Einschränkung, dass das System nicht mehr auf UDP-Telegramme zur Konfiguration reagiert. Ausgangspunkt für weitere Entwicklungen wäre die Implementierung einer Funktion, die in regelmäßigen Abständen die Funktion des Sockets überprüft. Dies könnte beispielsweise durch sogenannte Loopback-Pakete erreicht werden, also Daten, die das System an sich selbst sendet. Wird hierbei eine Fehlfunktion erkannt, müsste eine geeignete Maßnahme getroffen werden, also beispielsweise ein erneutes Öffnen des Sockets oder ein kompletter Reset des Systems.

7.3 Optimierung der Vorverarbeitungsfunktionen

Die Vorverarbeitung bietet ebenfalls noch Raum für Verbesserungen. Zur Steigerung der Performance wäre es möglich, alle implementierten Funktionen in Assembler zu programmieren.

Die Operationen könnten dann entweder Teil der Interrupt-Service-Routine sein oder als eigenständige Funktionen in Assembler geschrieben werden. So könnte die Systemauslastung deutlich sinken und die Möglichkeit entstehen, weitere Funktionen zu implementieren.

Da die Implementierung der Berechnung des tatsächlichen Minimalabstands zu rechenaufwändig ist, werden in der aktuellen Implementierung lediglich zwei Bereichsradien auf Unterschreitung geprüft. Im folgenden Abschnitt wird eine Möglichkeit aufgezeigt, wie der Algorithmus für den tatsächlichen Minimalabstand dennoch an den Rabbit Powercore 3800 angepasst werden könnte.

Die Implementierung des MinimalDistance-Algorithmus scheitert an den zeitaufwändigen Fließkommaoperationen. Es wird nun versucht, diese durch Integeroperationen zu ersetzen. In Kapitel 5.5 ist folgende Gleichung für die Vorverarbeitung hergeleitet:

$$r = \sqrt{m^2 + 2 \cdot y_1 \cdot \sin\gamma \cdot m + y_1^2}$$

Diese Formel lässt sich mittels quadratischer Ergänzung umformen zu:

$$r = \sqrt{\left((m + y_1 \cdot \sin\gamma)^2 - \sin^2\gamma \cdot y_1^2\right) + y_1^2} \tag{7.1}$$

$$r = \sqrt{(m + y_1 \cdot \sin\gamma)^2 + y_1^2 \left(1 - \sin^2\gamma\right)} \tag{7.2}$$

$$r = \sqrt{(m + y_1 \cdot \sin\gamma)^2 + y_1^2 \cdot \cos^2\gamma} \tag{7.3}$$

Der Term $y_1 \cdot \sin\gamma$ kann in der Konfigurationsphase als Fließkommawert berechnet werden und dann als 16-Bit Integerwert mit der Einheit mm in einer Liste für alle auftretenden Winkel gespeichert werden. Hierbei entsteht ein Rundungsfehler, der aber deutlich kleiner als die Messungenauigkeit ist. Analog erfolgt die Berechnung und Speicherung von $y_1^2 \cdot \cos^2\gamma$, bis auf die Abweichung, dass wegen der Quadrierung ein 32-Bit Integerwert zur Speicherung verwendet werden muss.

Bei der Berechnung im Betrieb werden eine 16-Bit Integer-Addition, eine 16-Bit Multiplikation (Quadrierung) mit 32-Bit Ergebnis, eine 32-Bit Addition und eine Quadratwurzeloperation durchgeführt. Die Additionen sowie die Multiplikation sind durch entsprechende Assemblerbefehle in kurzer Ausführungszeit möglich. Für die Quadratwurzel wird hingegen eine Funktion aus der mitgelieferten Bibliothek verwendet, die nur mit Fließkommazahlen arbeitet und eine Ausführungszeit von 1000 Takten hat. Deshalb ist die Berechnung für jeden Messwert nicht durchführbar. Es könnte jedoch stets r^2 statt r verglichen werden, da bei positiven Werten aus $r_1^2 < r_2^2$ stets auch $r_1 < r_2$ folgt. Der Vergleich würde dann zwischen 32-Bit Integerwerten erfolgen, wobei der Vergleich der höherwertigen 16 Bit in vielen Fällen ausreichend sein wird. Steht der kleinste Wert fest, so muss noch pro Messdatentelegramm für einen Wert die Quadratwurzel berechnet werden. Die Überprüfung der Bereichsradien könnte komplett entfallen und die Ansteuerung der LEDs würde abhängig vom ermittelten Wert erfolgen.

Die Implementierung und der Test dieses Algorithmus wäre eine Möglichkeit, das bestehende System mit nützlicher Funktionalität zu erweitern. Es müsste geprüft werden, inwiefern die Performance ausreicht, um einen stabilen Betrieb zu gewährleisten.

7.4 Implementierung mit anderer Hardware

Eine Implementierung des Systems mit anderer Hardware könnte ebenfalls Bestandteil späterer Entwicklungen sein. Mitte 2006 wurde der Rabbit-4000-Mikroprozessor vorgestellt, der ebenfalls in kostengünstigen Development-Kits angeboten wird. Der Prozessor unterscheidet sich vom Rabbit 3000 in folgenden Punkten:

- höherer Takt
- 16-Bit-Speicherbus
- DMA-Kanäle
- direkt integrierter Ethernet-Controller
- Serielle Schnittstelle mit 491,5 kBd

Durch den 16-Bit-Speicherbus und die DMA-Kanäle ist eine deutliche Steigerung der Leistung zu erwarten, zumal das Versenden von Netzwerkdaten nahezu ohne Prozessorlast funktionieren würde. Die Mehrleistung käme der Implementierung des oben beschriebenen Minimalabstandsalgorithmus zugute. Eventuell steht auch genügend Rechenleistung für weitere Vorverarbeitungsalgorithmen zur Verfügung.

Ebenfalls denkbar wäre eine Implementierung auf FPGA-Boards. Es wäre hier ein sinnvolles Konzept zu erarbeiten, wie eine sogenannte Soft-CPU an zusätzlich zu programmierende Hardware angebunden werden könnte. Es könnten weitaus komplexere Algorithmen implementiert werden, da aufgrund der Möglichkeit der Parallelverarbeitung durch hardwareseitig implementierte Funktionen eine sehr hohe Rechenleistung zur Verfügung stehen würde. Auf diese Art ist möglicherweise auch Linien- und Eckenerkennung implementierbar. Entsprechende Algorithmen sind in [Zha05] beschrieben.

7.5 Synchronisation zweier Lasermesssysteme

Die SICK LMS 200 bieten die Möglichkeit, durch eine Verbindung zweier Systeme eine gewisse Synchronisation zu erzeugen. Diese bewirkt, dass nur eins zur Zeit einen Laserstrahl aussendet, so dass es nicht zu Störungen zwischen den beiden Systemen kommt. Beim Anwendungsfall im Rahmen dieser Arbeit wäre es hingegen vorteilhaft, wenn beide Systeme genau im Gleichtakt messen würden.

Der Vorteil hierbei läge in einer höheren Aktualität des kompletten Datensatzes, da nicht nach einer eingetroffenen Messung auf Beendigung der nächsten Messung gewartet werden müsste. Möglicherweise ergäbe sich bezüglich der Performance auf dem Rabbit Powercore 3800 ein Vorteil, da es stets Zeiten mit sehr geringer Interruptaktivität geben würde, die dann komplett den Vorverarbeitungsfunktionen zur Verfügung stehen würden. Eine Störung bei gleichzeitiger Messung ist nicht zu befürchten, da die Systeme in entgegengesetzte Richtungen messen.

Zu entwickeln wäre eine Möglichkeit, das Synchronisationssignal in geeigneter Weise zwischen den beiden Lasermesssystemen zu verzögern. Es ist auch eine Lösung denkbar, bei der vom Rabbit Powercore 3800 für beide Systeme das Signal generiert und über eine entsprechen Schaltung zuführt wird.

7.6 Abschluss

In diesem letzten Kapitel wurden einige Ansatzpunkte für weitere Entwicklungen aufgezeigt. Auch ohne Verwirklichung dieser Weiterentwicklungen stellt das im Rahmen dieser Arbeit entwickelte System eine entscheidende Verbesserung des Serviceroboters dar. Das Problem der hohen Systemauslastung wurde gelöst und es kann auf zuverlässig eintreffende Messdatensätze zugegriffen werden. Weitere laufende und zukünftige Arbeiten, die sich thematisch mit dem Serviceroboter beschäftigen, profitieren von den Leistungen des entwickelten Systems, da auf dem Serviceroboter nun eine höhere Rechenkapazität zur Verfügung steht.

A Kurzbeschreibung zum Einbinden der Messdaten

In diesem Anhang wird erläutert, wie eine schnelle Einbindung der Lasermessdaten in eigene C-Programme erfolgen kann. Die Messdaten werden vom Powercore 3800 im lokalen Netzwerk über Multicast-UDP-Pakete verschickt. Der Hostrechner ist aktuell so konfiguriert, dass die Messdaten nicht in das Fachbereichsnetzwerk weiter geroutet werden. Eigene Anwendungen müssen folglich auf dem Hostrechner im mobilen Serviceroboter laufen.

In Listing 4.2 ist ein Beispielcode zu sehen, der Multicast-UDP-Pakete empfängt. Die Multicast-IP-Adresse ist in Zeile 41 für diese Anwendung bereits passend gesetzt. Auch der Port ist in Zeile 32 bereits auf 2222 gesetzt. Die dritte Konfigurationseinstellung ist die Netzwerkschnittstelle, auf der die Multicast-UDP-Paketen empfangen werden sollen, sie ist in Zeile 43 auf `eth0` gesetzt.

Der Beispielcode gliedert sich in zwei Teile. Der erste Teil öffnet den Socket und konfiguriert ihn für den Multicast-Empfang. Der zweite Teil ist ein Main-Loop, die `for`-Schleife, die mit dem `select()`-Aufruf auf neue UDP-Pakete wartet. Nach dem der `select()`-Aufruf zurückgekehrt ist, muss noch überprüft werden ob UDP-Daten vorliegen und diese in den Puffer passen. Erst danach werden die Daten in Zeile 76 abgeholt.

Eine Überprüfung und Abarbeitung der Telegramme ist im Beispielcode in Zeile 84 nur bedingt vorhanden, es wird ausschließlich der Empfang eines Telegramms ausgegeben. Das genaue Telegrammformat ist in Kapitel 5.6.1 beschrieben. Aus den darin beschriebenen Informationen ist der folgende C-Beispielcode für den Empfang von Messdatentelegrammen entstanden:

```
if (len >= 730 &&                // Telegramm muss lang genug sein
    buf[0] == 0x02 &&            // STX
    (buf[1] & 0xfe) == 0x80 &&   // Adresse 0x80 oder 0x81
    buf[2] == 0xD6 &&            // Länge LSB
    buf[3] == 0x02 &&            // Länge MSB
    buf[4] == 0xB0 &&            // Responsecode (Kommando)
    buf[5] == 0x69 &&            // 2 Byte für Anzahl der Messwerte
    buf[6] == 0x41)              // und Messbereich (mm-Modus)
  do_scandata(buf);              // Abarbeiten der Messdaten
```

Auf eine Überprüfung des CRC wird verzichtet, da ausschließlich geprüfte Telegramme vom Rabbit 3000 verschickt werden.

Eine Funktion zum Parsen der einzelnen Messwerte des Lasermesssystems könnte dann wie folgt aussehen:

```
void do_scandata(const unsigned char *buf) {
  int i;
```

```
  unsigned int dist;
  unsigned char den;

  printf("Messdaten von 0x%2X:\n", buf[1]);

  buf += 7;  // ersten 7 Byte sind Header usw.

  for (i = 0; i < 361; i++) {
    dist = *(buf++);
    dist |= ((*buf)&0x1f)<<8;
    den = (*(buf++))>>5;

    if (dist > 0x1ff7) dist = -1; // ungueltige Entfernung

    printf("%3u. dist = %4d  den = %d\n", i, dist, den);
  }
}
```

B Dynamic C - Installationsanmerkungen

Gefundene Problemlösungen für die Dynamic-C-Entwicklungsumgebung werden in diesem Anhang dokumentiert, so dass es einfacher wird, sie zu installieren und zu benutzen.

B.1 Windows - Mehrbenutzerinstallation

Die Installation von Dynamic C erfolgt normalerweise mit einem Administrator-Account. Das hat zur Folge, dass die Benutzer des Windows-Systems die Dateien von Dynamic C selber nicht überschreiben dürfen.

Dynamic C geht jedoch davon aus, dass es im Programmverzeichnis Schreibrechte hat. Übersetzt der Administrator ein Beispielprogramm, so entstehen in dem Bibliotheksverzeichnis `/Lib/` zu jeder Datei eine `.HX1` und eine `.MD1` Datei. Im BIOS-Verzeichnis `/Bios/` entstehen neben der `RabbitBios.c`-Datei die Dateien `Rabbitbios.BDL`, `Rabbitbios.brk`, `Rabbitbios.HDL`, `Rabbitbios.map` und `Rabbitbios.rom`. Damit ein regulärer Benutzer auch Programme übersetzen kann, müssen diese fünf Dateien für ihn überschreibbar gemacht werden.

Sollen User auch in dem Verzeichnis keine Schreibberechtigung für die fünf Dateien erhalten, so gibt es die Möglichkeit, dass in den Compileroptionen ein benutzerdefiniertes BIOS ausgewählt wird. Die Datei `/Bios/RabbitBios.c` muss dazu in ein Verzeichnis kopiert werden, in dem Schreibrechte vom Benutzer vorhanden sind. Die Einstellung findet sich unter:
Options | Compiler | User Defined BIOS File | Use | ...

Bei Benutzung des Kommandozeilencompilers ist die Option `-bf BIOSDateipfad` zu gebrauchen.

B.2 Linux - Kommandozeile mit Wine

Unter Linux lässt sich der Kommandozeilencompiler mit Hilfe von Wine[1] benutzen. Zur Verfügung stand ein Wine 0.9.15. Es wird keine komplette Dynamic-C-Installation gebraucht. Der Compiler besteht bei Dynamic C 9.40 aus den Dateien `Dccl_cmp.exe`, `cc3250mt.dll`, `vcl50.bpl` und `vclx50.bpl`. Die komplette Bibliothek befindet sich in dem Verzeichnis `/Lib/`. Das BIOS ist in `/Bios/` mit den `*.bin`-Dateien und der `RabbitBios.c` Quelldatei.

Damit der Compiler das Bibliotheksverzeichnis `/Lib/` findet, muss es in einer `lib.dir`-Datei aufgeführt werden. Diese Datei wird im Verzeichnis des Compilers gesucht. Alternativ kann sie mit dem Kommandozeilenparameter `-lf pfad/lib.dir` angegeben werden. Im einfachsten Fall liegt sie neben dem Compiler `Dccl_cmp.exe` und enthält folgende Zeile:

[1] Wine (*Wine Is Not an Emulator*) kann Windows und DOS-Programme unter Linux ausführen.

```
lib
```

Soll der Compiler den übersetzten Code nicht direkt an den Mikroprozessor übertragen, so muss eine rti-Datei angegeben werden. Diese enthält Informationen über den Mikroprozessor, so dass der Compiler weiß, wie er den Code zu übersetzen hat. Für den aktuell verwendeten Rabbit Powercore 3800 findet sich der Inhalt der rti-Datei im folgenden Listing:

```
[Parameters]
Board ID=0x2301
Board Name=PowerCore FLEX Board Series
CPU Name=Rabbit 3000 revision IL2T/IZ2T/UL2T/UZ2T
CPU ID=3000
CPU Revision=1
Crystal Speed (MHz)=25,8048
RAM Size (KBytes)=1024
Flash Size (KBytes)=512
```

C Arbeitsverteilung

Diese Diplomarbeit ist ein Gemeinschaftswerk von Stephan Pöhlsen und Hannes Bistry. Stephan Pöhlsen spezialisierte sich auf die Entwicklung der UDP-Umsetzung auf Hostrechnerseite und die Ansteuerung der SICK-Lasermesssysteme. Hannes Bistry befasste sich insbesondere mit dem Rabbit Powercore 3800 und dem RS-422-Standard. Die Entwicklungsarbeit erfolgte gemeinsam am Arbeitsbereich.

Hannes Bistry schrieb:

In Kapitel 1:	Abschnitte 1.1 bis 1.3
In Kapitel 2:	Abschnitte 2.3 und 2.4
In Kapitel 3:	Abschnitte 3.1 bis 3.4
In Kapitel 4:	Abschnitt 4.2
In Kapitel 5:	Abschnitte 5.1, 5.3 und 5.5
In Kapitel 7:	Abschnitte 7.1 bis 7.5

Stephan Pöhlsen schrieb:

In Kapitel 2:	Abschnitte 2.1, 2.2 und 2.5
In Kapitel 4:	Abschnitte 4.1 und 4.3
In Kapitel 5:	Abschnitte 5.2, 5.4, 5.6 und 5.7
In Kapitel 6:	Abschnitte 6.1 bis 6.4
Im Anhang:	Abschnitte A, B.1 und B.2

Der Abstract und Kapitel 1.4, sowie alle Kapiteleinleitungen und Zusammenfassungen sind gemeinsam geschrieben.

D Danksagungen

An dieser Stelle möchten wir uns bei den Personen bedanken, die uns die Anfertigung dieser Arbeit ermöglicht haben.

Herrn Prof. Dr. J. Zhang danken wir dafür, dass wir an seinem Fachbereich unsere Diplomarbeit schreiben durften, für die zur Verfügung gestellten Arbeitsmittel und für das Vertrauen, das er uns durch die Vergabe der Arbeit entgegengebracht hat.

Für die Übernahme des Zweitgutachtens danken wir Herrn Prof. Dr.-Ing. D. P. F. Möller ganz herzlich.

Folgenden Personen des Fachbereich danken wir für die Unterstützung während unserer Arbeit:

- Dr. Andreas Mäder, für die vielen hilfreichen Tipps (insbesondere bezüglich VHDL) und das Einrichten von Benutzeraccounts
- Daniel Westhoff, für die Erklärungen zu der Software des Serviceroboters und für die Zusammenarbeit bei der Abstimmung der zu implementierenden Funktionen
- Tim Baier-Löwenstein, für das Einrichten des Serviceroboters und für das Bereithalten des Raumschlüssels
- Manfred Grove, für das Beschaffen von Komponenten

Zusätzlich sind wir noch vielen weiteren Personen aus unserem privaten Umfeld dankbar, denen wir unseren Dank persönlich ausrichten möchten.

Literaturverzeichnis

[Bon05] E. Bonnert. *Meilenstein der Fahrzeug-Robotik*. c't - magazin für computertechnik 2005/23, Hannover, Oktober 2005

[Cat05] J. Catsoulis. *Designing Embedded Hardware*. Second Edition. O'Reilly Media, Sebastopol, California (USA). 2005

[DynCFRM] Z-World Inc. *Dynamic C Function Reference Manual*. Davis, California (USA), September 2004

[DynCTCP1] Z-World Inc. *Dynamic C TCP/IP User's Manual – Volume 1*. Davis, California (USA), Oktober 2004

[DynCUM] Z-World Inc. *Dynamic C User's Manual*. Davis, California (USA), August 2004

[ELROB] Internetauftritt der European Land Robot Trial `http://www.m-elrob.eu/`

[Fra04] J. Fraden. *Handbook of Modern Sensors – Physics, Design, and Applications*. Third Edition, Springer-Verlag New York Inc., Januar 2004

[Kai93] B. Kainka. *Messen, Steuern, Regeln über die RS-232-Schnittstelle*, 5. Auflage, München, Deutschland: Franzis-Verlag GmbH, 1993

[Kam92] Dr.-Ing. K.D.Kammeyer. *Nachrichtenübertragung*. B.G.Teubner. Stuttgart. 1992

[Lue99] H. D. Lüke. *Signalübertragung*. 7. Auflage, Springer-Verlag Berlin Heidelberg New York, 1999

[Mae05] Dr. A. Mäder. *VHDL Kompakt*. Hamburg, Mai 2005

[Moe03] Prof. Dr.-Ing. D. P. F. Möller. *Rechnerstrukturen*. Springer-Verlag Berlin Heidelberg New York. 2003

[PCFlex] Z-World Inc. *PowerCore FLEX – Customizable Microprocessor Core Modules*. Davis, California (USA), April 2005

[PCUM] Z-World Inc. *PowerCore FLEX – User's Manual*. Davis, California (USA), Februar 2005

[R3000DH] Rabbit Semiconductor. *Rabbit 3000 Microprocessor – Designer's Handbook*. Davis, California (USA), Juni 2003

[R3000UM] Rabbit Semiconductor. *Rabbit 3000 Microprocessor User's Manual*. Davis, California (USA), Juni 2005

[RabInst] Rabbit Semiconductor. *Rabbit 2000/3000 Microprocessor – Instruction Reference Manual*. Davis, California (USA), Januar 2004

[RC89] Dr. rer. nat. W. Rosenstiel, Dr. rer. nat. R. Camposano. *Rechnergestützter Entwurf hochintegrierter MOS-Schaltungen* Springer-Verlag Berlin Heidelberg New York. 1989

[RS422] TIA/EIA-422-B. *Electrical Characteristics of Balanced Voltage Digital Interface Circuits*, Washington D.C. (USA), Mai 1994

[SickQM] SICK AG. *Quick Manual for LMS communication setup – Hardware setup and measurement mode configuration*. Version 1.1, März 2002

[SickTB] SICK AG. *Lasermesssysteme LMS 200/LMS 211/LMS 220/LMS 221/ LMS 291 – Technische Beschreibung*. Juni 2003

[SickTL] SICK AG. *Telegramme zur Bedienung/Konfiguration der Lasermesssysteme LMS 2xx – Telegrammlisting*. Firmware-Version V2.10/X1.14, April 2003

[SN74LS540] Texas Instruments Inc. *SN54LS540, SN54LS541, SN74LS540, SN74LS541 - octal buffers and line drivers with 3-state outputs*. Dallas, Texas (USA), März 1988

[SN75C1168] Texas Instruments Inc. *SN65C1167, SN75C1167, SN65C1168, SN75C1168 - dual differential drivers and receivers*. Dallas, Texas (USA), 2003

[Sta04] William Stallings. *Data and Computer Communications* Seventh Edition. Pearson Education, Inc. Pearson Prentice Hall. Upper Saddle River, New Jersey(USA).2004

[SW02] A. Schneider, D. Westhoff. *Autonomous Navigation and Control of a Mobile Robot in a Cell Culture Laboratory*. Bielefeld, Juni 2002

[Tan03] A. S. Tanenbaum. *Computer Networks*. Fourth Edition, Upper Saddle River, New Jersey (USA): Prentice Hall, 2003

[TG01] A. S. Tanenbaum, J. Goodman. *Computerarchitektur*. 4. Auflage, München, Deutschland: Pearson Studium, 2001

[UniOS] Internetauftritt der Universität Osnabrück
`http://kos.informatik.uni-osnabrueck.de/`

[Xil03] Xilinx Inc. *Linear Feedback Shift Register*. San Jose, California (USA), März 2003

[Zha05] J. Zhang. *Vorlesung: Angewandte Sensorik*. Hamburg, Oktober 2005

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereichs einverstanden.

Hannes Bistry

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereichs einverstanden.

Stephan Pöhlsen