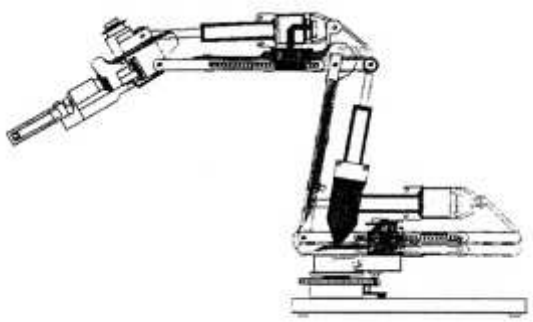
Besondere Lernleistung

Robotersteuerung

von Karsten Becker



Friedrich-List-Schule, Wiesbaden Abitur-Jahrgang 2001

INHALTSVERZEICHNIS

1. Grundüberlegung

Ziel des Projektes ist es eine präzisere Steuerung für den Roboterarm zu realisieren. Von der vorhandenen Steuerung wurden keinerlei Informationen über die Bewegung des Armes an den Computer zurückgemeldet. Dies verhindert ein zielgenaues Steuern des Armes. Der Arm hätte zwar über Zeitintervalle gesteuert werden können, doch wäre dies sehr ungenau geworden. Außerdem war keine Information darüber verfügbar, ob der Roboterarm die Bewegung auch tatsächlich ausgeführt hat.

Ferner war es nicht möglich mehrere Bewegungen gleichzeitig auszuführen. Dies bedeutet in der Regel Zeitverlust, und ermöglich keine komplexen Bewegungsabläufe.

Diese Funktionsdefizite soll die neue Steuerung beseitigen. Mit ihr kann man den Arm auf 1 Grad genau ansteuern. Die Position des Armes ist zu jedem Zeitpunkt ermittelbar. Um die Sicherheit beim Betrieb des Armes zu erhöhen, soll dieser seine Bewegung abbrechen und eine Meldung an den Computer geben, wenn es zu Störungen der Bewegung kommt. Dies kann zum Beispiel passieren, wenn die Bewegung des Roboters verhindert wird oder die Programmierung des Ablaufes falsch war.

1.1 Lösungsansätze

1.1.1 Lösung über Standardkomponenten

Hierbei werden nur TTL-Bausteine und diskrete Bauteile auf der Platine benötigt. Diese sind einfach zu beschaffen und preiswert. Allerdings wird die Konstruktion der Platine kompliziert.

1.1.2 Lösung über rekonfigurierbare Logik

Bei dieser Lösung werden die gesamten TTL-Bauteile in einem Bauteil zusammengefasst. Dadurch ergibt sich eine sehr kompakte Lösung, die aber teuer ist. Die Konstruktion ist noch schwieriger als bei der ersten Lösung. Allerdings kann man durch diese Lösung das Kalibrierungsboard, welches bei der ersten Lösung ja nötig ist, wegfallen lassen.

1.1.3 Gewählte Lösung

Ich habe mich zuerst für den ersten Lösungsansatz entschieden, weil er billiger ist. Im Vergleich:

- Lösung 1 kostet 24 DM pro Motorplatine
- Lösung 2 kostet 35 DM pro Motorplatine

Allerdings ist bei der ersten Lösung der Aufwand zum Fertigen der Platine enorm hoch. Man muß die Platine beidseitig herstellen, durchkontaktieren und Löten. Weil so viele Bauteile auf der Platine sind, ist auch die Gesamtlänge der Leiterbahnen höher, was auch die Fehlerrate bei der privaten Fertigung drastisch erhöht. Bei einer Industriellen Fertigung sollten diese Fehler jedoch nicht auftreten, so dass die erste Lösung für die Industrielle Fertigung ist und die zweite eher für die private. Die von mir präsentierte Lösung ist dann auch die zweite Lösung.

2 Anlagenbeschreibung

Die gesamte Anlage besteht aus 5 Teilen. Der wichtigste Teil ist der Roboterarm. Er ist mit 6 Motoren ausgestattet. Mit diesen kann er:

- die Hand öffnen/schließen
- die Hand rechts/links drehen
- die Hand nach oben/unten kippen
- den Ellenbogen strecken/beugen
- den Arm strecken/beugen
- den Arm rechts/links seitwärts bewegen

An jeden Motor ist eine Platine angeschlossen. Diese regelt den Motor nach den Vorgaben des Mikrocontrollers, einem weiteren Teil der Anlage.

Ein Mikrocontroller ist ein Prozessor, wie zum Beispiel ein Intel Pentium. Er wird jedoch mit einer viel geringeren Geschwindigkeit (8 MHz) getaktet und ist einfacher zu programmieren.

Die Platinen sind über 16 Leitungen miteinander verbunden. 6 Leitungen bilden den Adressbus, 8 liefern die Daten für die Platinen, 2 geben die Richtung der Kommunikation vor.

Eine weitere Leitung von der Motorplatine zum Mikrocontroller dient zur Signalisierung für den Mikrocontroller. Wenn diese den High-Pegel führt, muss der Mikrocontroller als nächstes die Betriebsparameter, also die Position des Motors und den Strom, von der entsprechenden Platine abfragen. In der Zwischenzeit bewegt sich der Motor nicht.

An das Bussystem können auch GIO-Platinen angeschlossen werden. GIO steht für General Input Output. Mit GIO-Platinen kann man die Robotersteuerung um verschiedene Funktionen erweitern, wie z.B. ein LCD-Display, PWM-Steuerung oder diverse Sensoren.

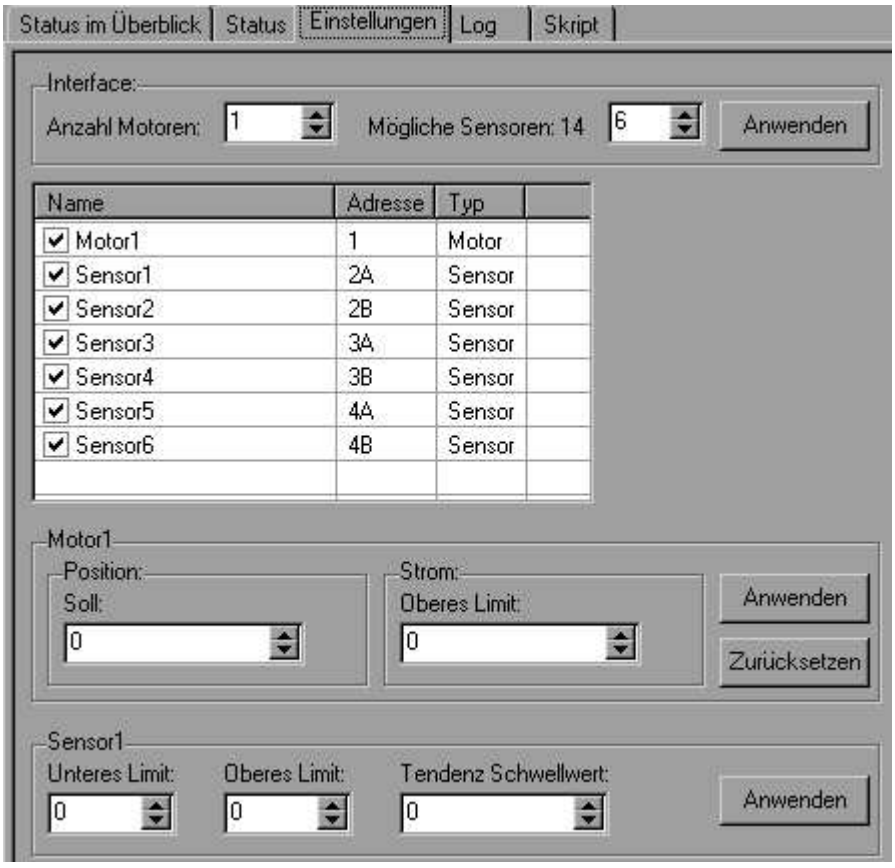
Da 6 Leitungen für die Adresse zu Verfügung stehen, sind bis zu 64 Geräte anschließbar. Dann müsste nur die Stromversorgung anders dimensioniert werden. Die Kommunikation des Mikrocontrollers mit dem Computer findet über die serielle Schnittstelle statt.

2.1 Software

Die Software stellt die Schnittstelle zwischen dem Benutzer und dem Mikrocontroller dar. Sie visualisiert die ankommenden Daten und führt die komplexeren Regelmechanismen durch. Generell könnte man den Mikrocontroller auch über jedes beliebige Terminal-Programm, also ein einfaches Programm zum Kommunizieren über die serielle Schnittstelle, konfigurieren, aber dann wären komplexe Bewegungsabläufe nicht mehr möglich.

Realisiert wurde die Software in Delphi. Der Vorteil dieser Sprache ist eine leicht zu bedienende graphische Oberfläche. Außerdem benutze ich diese Sprache schon sehr lange und bin vertraut mit ihr.

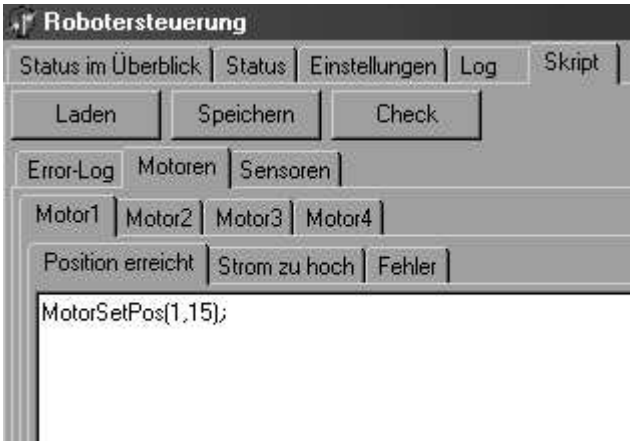
2.1.1 Das Benutzerinterface



In der Software gibt man als erstes die Anzahl der Motoren und GIO-Platinen an. Man kann ihnen auch Namen geben um die Arbeit zu erleichtern. In den Einstellungen kann man dann bei den Motoren den maximalen Strom und die Position, zu der sie fahren sollen, angeben. Bei den GIO-Platinen kann man einen unteren und einen oberen Schwellwert angeben.

Wenn einer der Werte über- bzw. unterschritten wird, wird das im Register „Skript“ dafür vorgesehene Skript ausgeführt. In den Skripten kann man dann entweder Warnungen ausgeben oder die Werte neu setzen. So wird es z.B. möglich den Arm in eine neue Position fahren zu lassen, wenn er eine vorher definierte Position erreicht hat.

Das Eingabefeld zum Editieren der Skripte ist eine Ableitung von TRichEdit, einer Standardkomponente von Delphi, und könnte aufgrund ihrer Fähigkeit zur Formatierung von Text in einer späteren Version noch um eine SyntaxHighlighting-Engine erweitert werden. Dafür fehlte mir jedoch die Zeit.



SyntaxHighlighting macht das Lesen des Codes wesentlich einfacher, da Befehle und Konstanten anders formatiert sind als der restliche Quellcode.

2.1.2 Die Funktionsweise der ScriptEngine

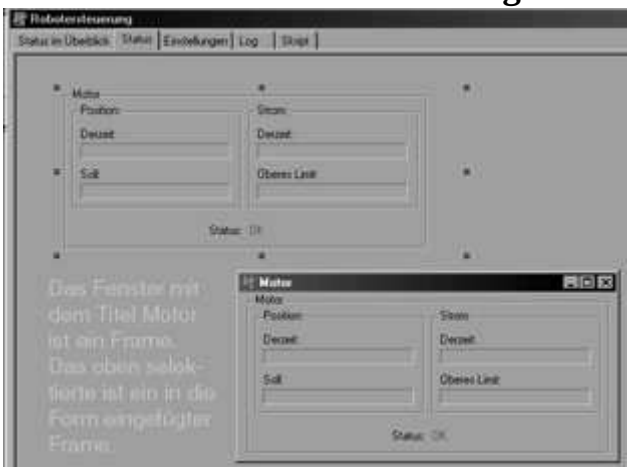
Die ScriptEngine, so nenne ich die befehl-ausführende Software, prüft als erstes die Syntax, also die sprachliche Richtigkeit des Scripts, und überführt diese in eine für den Computer interpretierbare Form. Sollten dabei Fehler auftreten, werden diese gemeldet und das Script wird nicht ausgeführt. Sollte das entsprechende Script zur benötigten Zeit nicht lauffähig sein, wird als nächstes das Error-Script ausgeführt. Sollte auch dieses nicht ausführbar sein, wird der Roboterarm angehalten und das Programm gibt eine Fehlermeldung aus. Damit soll sichergestellt werden, dass keine vom Benutzer unerwünschten Bewegungen ausgeführt werden. Die ScriptEngine unterstützt Variablen und kann auch einfache Sprungbefehle ausführen, wie man sie zum Beispiel aus Basic kennt.

Um die ScriptEngine möglichst flexibel zu halten, habe ich eine Actionlist verwendet, die nach Übereinstimmungen des Namens mit dem Befehl durchsucht wird. Wenn ein solcher Eintrag gefunden wurde, wird der entsprechende Befehl ausgeführt, wobei ihm die notwendigen Argumente schon in entsprechender Form bereit gestellt werden.

Unter Status findet man ausführliche Informationen zu jedem Motor bzw. GIO. Im Register „Status im Überblick“ findet man nochmals alle Informationen zusammengefasst.

Unter Log findet man die den Mikrocontroller betreffende Informationen.

2.1.3 Die Programmrealisierung

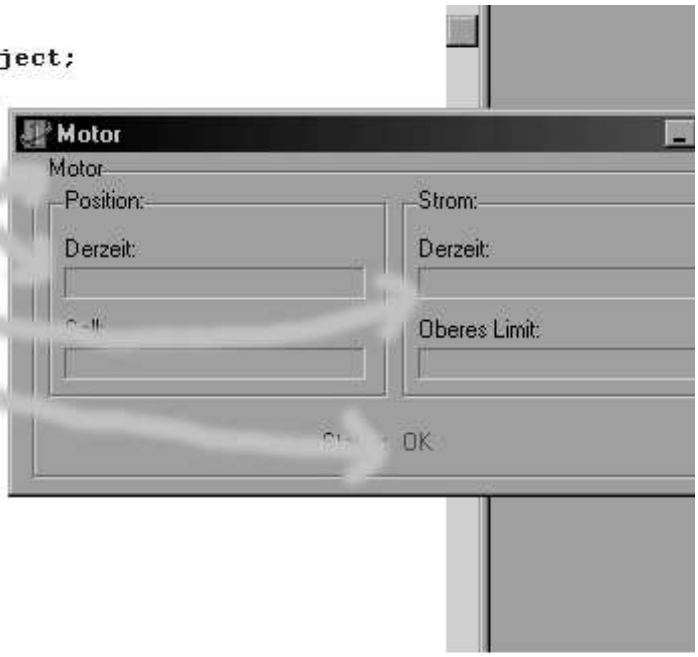


Für alle wiederkehrenden Elemente wurden Frames verwendet. Frames sind Elemente, die sich wie normale Formen bearbeiten lassen und auch so aussehen. Dadurch, dass jedes Frame seine eigene Datei für den Quellcode hat, wird der Quellcode übersichtlicher gegliedert.

Einen Frame kann man sich wie einen Ordner vorstellen. Die in diesem Ordner befindlichen Dateien bilden die Komponenten. Man kann nun den Ordner überall hinkopieren und der Inhalt bleibt unverändert. Bei einem Frame bedeutet das, dass die Komponenten ihre Eigenschaften, wie z.B. ihr Position auf dem Frame beibehalten. Frames können innerhalb einer Form beliebig platziert werden.

Um Arbeitsspeicher zu sparen, werden die Frames nur zur Laufzeit erzeugt, wenn sie benötigt werden, bzw. werden freigegeben, wenn sie nicht mehr benötigt werden.

```
type
  TUpdateproc = procedure () of object;
  TMotorInf = record
    motor          : TMotor;
    aktiv          : Boolean;
    Name           : String[20];
    PosDerzeit     : TProgressbar;
    StromDerzeit   : TProgressbar;
    KurzstatusOK   : TLabel;
    KurzstatusER   : TLabel;
    MotorOK        : TLabel;
    MaxI           : integer;
    SollPos        : integer;
    updateproc     : TUpdateproc;
    updateprocSet  : TUpdateproc;
    OnReached      : TScriptEdit;
    OnHighCurrent  : TScriptEdit;
    OnError        : TScriptEdit;
  end;
```



Die Anzahl der für die Steuerung benötigten Komponenten ist sehr hoch. Deshalb habe ich die wichtigsten Komponenten, wie z.B. die Progressbar, Labels und Scriptedits in einem record zusammengefasst. Um bei der Analogie mit dem Ordner zu bleiben, man kann sich die Bestandteile des Records wie Verknüpfungen auf Dateien aus einem Ordner vorstellen. Der Name der Motoren und GIO-Platinen ist auf 20 Zeichen begrenzt, weil dann der Array eine definierte Größe hat und damit das Speichern in einer Datei einfacher wird.

Hierdurch wird die Geschwindigkeit des Programms erheblich erhöht.

2.2 Hardware

2.2.1 Der Mikrocontroller

Er sammelt alle Informationen von den Motorplatinen und meldet bei Fehlern oder bestimmten Ereignissen diese an den Computer weiter.

Die Motorplatinen können jeweils nur 16 Takte der Motoren zählen. Danach wird der Mikrocontroller nach neuen Daten abgefragt. Im Mikrocontroller sind für jeden Motor 2 Byte Speicher reserviert. So kann der Mikrocontroller den gesamten Bewegungsraum des Roboterarms verwalten.

Informationen an den Computer werden weitergegeben wenn:

- Die beim Mikrocontroller gespeicherte Position erreicht ist
- Der Strom zu groß geworden ist
- Bei den GIO-Platinen das Minimum oder das Maximum erreicht ist

Zuerst wollte ich einen Intel Mikrocontroller 8031 für diese Aufgraben programmieren. Aber aufgrund der 16-Bit mathematischen Fähigkeiten entschied ich mich dann doch für einen Atmel AVR. Dieser ist nämlich für Hochsprachen optimiert. Die Programmierung in einer Hochsprache ist wesentlich einfacher als in Assembler. Die schon vorhandenen 16-Bit Algorithmen für diesen Mikrocontroller verkürzten die Zeit wesentlich.

2.2.2 Die Motorplatine:

Die Motorplatine ist mit jeweils einem Motor des Roboterarms verbunden und
wenn eines der Limits erreicht ist, wird die Bewegung des Motors gestoppt.

Wenn die Adresse an dem Bus mit der Adresse auf der Platine übereinstimmt werden die Daten von den Datenleitungen gelesen. 7 Bits werden an die beiden 4-Bit-Comparatoren des Typs 7485 weitergegeben. Das letzte Bit geht an die Thyristoren und einen Inverter, die damit die Bewegungsrichtung des Motors festlegen.

Ein Comparator ist mit einem AD-Wandler zusammengeschaltet der den zum Motor fließenden Strom mit einer Tast-Frequenz von 1 MHz in 8 Bit wandelt. Da zur Steuerung des Motors aber nur 4-Bit benötigt werden, werden auch nur die obersten 4-Bits beachtet. Das reicht auch für eine grobe Erfassung des Stromes aus, um die Funktionssicherheit des Roboterarmes zu gewährleisten.

Der zweite Comparator ist mit einem Counter des Typs 74161 verbunden. Der Counter zählt die von dem Motor bei jeder Umdrehung generierten Takte.

Der Überlauf des Counters und die >= Ausgänge der Comparatoren sind über einen ODER-Baustein des Typs 7432 zusammengefasst. Die Leitung nenne ich Full, und solange diese den Low-Pegel führt, bewegt der Motor sich. Zusätzlich ist die Leitung mit einem Port des

Mikrocontrollers verbunden, so dass nicht immer alle Platinen im Durchlauf abgefragt werden müssen. Dies verhindert Polling.

2.2.3 Die Verteilerplatine

Sie hat nur die Funktion die Versorgungsspannungen und Datenleitungen für die Motor-, GIO-Platinen und den Mikrocontroller bereit zu stellen.

2.2.4 Die mindestens notwendigen Teile einer GIO-Platine

GIO steht für General Input Output. Eine GIO-Platine ist also eine Platine, die es dem Mikrocontroller erlaubt entweder Daten von ihr auszulesen oder auf sie zu schreiben oder beides. Jedoch nicht gleichzeitig.

Die dafür nötigen Bauteile sind ein paar Jumper zum Einstellen der 6-Bit-Adresse, 2 UND-Bausteine des Typs 74LS08, die die an den Jumpern eingestellte Adresse mit der auf dem Adressbus anliegenden Adresse vergleichen, und ein weiterer UND-Baustein, der kontrolliert, ob auch alle Teile der Adresse mit der am Adressbus anliegenden Adresse übereinstimmen.

Zum Empfangen wird ein 74LS373 benötigt, der über einen Enable-Eingang die Signale bei Übereinstimmung der Adresse empfängt und speichert.

Zum Senden wird noch ein 74LS125 benötigt, der die Signale an den Datenbus anlegt, wenn er die Aufforderung dazu erhält.

2.2.5 Von mir entwickelte GIO-Platinen

2.2.5.1 Kalibrierungsboard

Diese Platine ergänzt die Motorplatine, um die Motoren an ihren Endpositionen kalibrieren zu können.

2.2.5.2 LCDBoard

Das LCDBoard wird benötigt um dem Anwender Informationen über mögliche Fehler mitzuteilen, zum Beispiel, wenn der Computer nicht mehr reagiert.

3 Aufgetretene Probleme und Lösungen

3.1 Hardware

Beim Erstellen der Hardware entstanden sehr viel mehr Probleme als bei der Software. Dies hängt damit zusammen, dass man bei der Software Fehler sehr leicht durch einen integrierten

Debugger finden kann, während bei der Hardware alle Fehler durch mühsames Analysieren gefunden werden müssen und sich die Fehler oftmals nicht reproduzieren lassen.

3.1.1 Der Mikrocontroller

3.1.1.1 Wechsel auf AVR-Mikrocontroller wegen Programmiersprache

Zunächst entwickelte ich die Grundzüge eines Programms für den Intel 8031 Mikrocontroller. Da für diese Aufgabe 16-Bit-Funktionen benötigt werden und der Intel 8031 nur mit 8 Bit rechnen kann, wäre ein großes und somit unübersichtliches Programm in Assembler zu erstellen gewesen. Der letztlich eingesetzte AVR-Mikrocontroller verarbeitet zwar auch nur 8 Bit, es gibt aber genügend C-Compiler, die für diesen Mikrocontroller 16-Bit Algorithmen bereitstellen. Aufgrund seiner Optimierung auf Hochsprachen ist immer noch genügend Rechenleistung übrig, um die anstehenden Kontrollarbeiten auszuführen.
Außerdem lassen sich die AVRs im System programmieren, so dass man kein spezielles Programmiergerät benötigt und auch keine weiteren Ports (Ein-/Ausgänge) durch EEPROMs belegt werden.

3.1.1.2 Überlegungen bezüglich der Baudrate

Schwierig wurde es dann auch bei der Programmierung des Mikrocontrollers. Der AVR kann die Befehle zwar genügend schnell ausführen. Aber das Senden von nur 10 Buchstaben über die serielle Schnittstelle erwies sich selbst mit 115200Kbaud als zu langsam.
Es hätten dann nur noch 640 zusätzliche Befehle ausgeführt werden können.

Controller	**AT90S8515**	**AT90S8515**	**AVRMega161**	
µCtakt	7372800	7372800	7372800	Hz
MotorFrequenz	160	160	160	Hz
MaxReaktionszeit/Motor	0,00625000	0,00625000	0,00625000	Sekunden
Baudrate	115200	115200	921000	Kbits/sek
MaxZeichen	7	10	20	Zeichen
Zeit/Status	0,00048615	0,00069448	0,00017376	Sekunden
MaxMotor	8	8	8	
MaxReaktionszeit/gesamt	0,00078125	0,00078125	0,00078125	Sekunden
Zeit/Befehl	0,00000014	0,00000014	0,00000014	Sekunden
MaxBefehle/Motor	2176	640	4479	Befehle
MaxStatusse/MaxReaktion	2	1	4	
MaxStatus/sek	2057	1440	5755	
AbfrageIntervall/Platine	32	22	90	Abfragen/Platine/sek

Diese Tabelle verdeutlicht die Timing-Probleme. Während der Motor 160 Umdrehungen pro Sekunde macht und dabei auch 160 Takte abgibt, muss der Mikrocontroller die Daten in 1/Takte, also der in der Zeile MaxReaktionszeit/Motor angegeben Zeit, aktualisieren, damit es nicht zu Aussetzern während der Bewegung kommt.

Damit der Computer bei einem evtl. Problem benachrichtigt werden kann, muss die Kommunikation mit dem Computer auch in der entsprechenden Zeit erledigt werden. Ausschlaggebend hierfür ist die Baudrate und die Anzahl der Zeichen, die man pro Benachrichtigung, im Folgenden Status genannt, benötigt.
Hierzu dividiert man die Baudrate durch 8 um die Geschwindigkeit in Byte/sek zu erhalten. Da ein Zeichen auch ein Byte ist, kann man nun diesen Wert durch die Anzahl der Zeichen teilen. Je mehr Zeichen benötigt werden, desto länger dauert die Kommunikation. Aber je mehr Zeichen pro Status, desto verständlicher und ausführlicher wird die Kommunikation. Die Zeit, die pro Status verloren geht, finden Sie in der Zeile Zeit/Zeichen.
In der Tabelle sieht man auch schon die drastischen Auswirkungen der Baudrate und der Zeichen auf die benötigte Zeit. Wenn jetzt aber alle 8 Motoren zur selben Zeit ihre Daten aktualisiert haben wollen, benötigt man eine 8-mal kleinere Zeit gegenüber der Zeit für einen Motor. Der Wert für diese Zeit steht in der Zeile MaxReaktionszeit/gesamt. In dieser Zeit muss der Mikrocontroller in der Lage sein den Motor mit neuen Daten zu versorgen und evtl. den Computer zu benachrichtigen. Für die Benachrichtigung gehen, wie schon erwähnt, Zeit/Status-Sekunden verloren. Diese Zeit kann also nicht mehr für andere Zwecke verwendet werden, wie zum Beispiel dem Abfragen der anderen Platinen. Die Anzahl der ausführbaren Befehle ist bei den AVRs gleich der Frequenz, also bei 8Mhz kann ein AVR auch 8Millionen Befehle/Sek. ausführen.
Die Anzahl der Befehle, die er noch ausführen kann, nachdem er einen Statusbericht versendet hat, findet sich in MaxBefehle/Motor. Wieviele Statusberichte er während der MaxReaktionszeit maximal senden kann, findet sich in MaxStatus/MaxReaktion.
In einer Sekunde kann er also maximal MaxStatus/Sek Status versenden. Wie oft er eine Platine in einer Sekunde abfragen kann, steht in der Zeile Abfrageintervall/Platine. Je mehr, desto besser.

3.1.1.3 Schlussfolgerung

Ein AT90S8515 Mikrocontroller ist ausreichend. Allerdings muss dann die Kommunikation auf das Nötigste beschränkt werden, was die Verständlichkeit der Meldungen für den Menschen nicht gerade vereinfacht.
Gegen den Einsatz eines AVRMega161 spricht allerdings, dass eine besondere Schnittstellenkarte in dem PC sitzen muss, da dieser normalerweise nur 115,2Kbaud unterstützt. Ferner ist der AVRMega161 noch nicht offiziell erschienen.
Für den AVRMega161 sprechen aber die schnellere Kommunikation mit dem verständlicheren Befehlssatz sowie ein höheres Abfrageintervall für GIO-Platinen.

3.1.1.4 Endabschaltung der Motoren

Bei der Programmierung des Mikrocontrollers stellte sich heraus, dass die Platine nicht die Endposition des Roboterarmes erkennen kann. Das kommt daher, dass der Roboterarm, wenn er seine Endposition erreicht hat, die Motoren abschaltet.

Dieses Problem lässt sich nur lösen, indem man die Endabschaltung der Motoren mit weiteren Bausteinen erfasst und für den Fall, dass die Endposition erreicht ist, eine Benachrichtigung an den Mikrocontroller sendet. Da aber auf der Motorplatine kein Platz mehr war, habe ich diese Kalibrierungsplatine als GIO realisiert. Der Bus ist dafür leistungsfähig genug.

3.1.2 Die Motorplatine

3.1.2.1 Probleme bei der Platinenherstellung

Hier ergaben sich die meisten Probleme. Das fing schon bei der Fertigung der Platinen an. Zuerst benutzte ich eine Schaumätzmaschine, bei der sich aber das Kupfer an den verschiedenen Stellen unterschiedlich stark abtrug und es von daher an einigen Stellen zu Unter- bzw. Überätzungen kam. Das war bedingt durch die horizontale Lage der Leiterplatten und die punktuelle Anordnung des Luftverteilers.

Eine andere Ätzanlage löste dann dieses Problem, indem sie die Platten vertikal aufnahm und der Luftverteiler sich gleichmäßig über die gesamte Länge ausbreitete.

Eine weitere Schwierigkeit bei der Herstellung war, dass ich beidseitig ätzen musste. Dafür müssen die beiden Seiten genau übereinander liegen. Das bekam ich in den Griff, indem ich die Platine schon vor dem Belichten an definierten Stellen anbohrte und so die Folien zum Belichten an den Bohrungen ausrichten konnte.

Als ich dann mit der Bestückung der Platinen beginnen wollte, stellte sich heraus, dass man die IC-Sockel nicht so gut beidseitig verlöten konnte. Deshalb musste ich mir noch eine Maschine zum mechanischen Durchkontaktieren beschaffen.

Beim mechanischen Durchkontaktieren kam es dann aber wider Erwarten zu anderen Problemen. Die Köpfe der Nieten überlappten die benachbarten Leiterbahnen. Deshalb musste ich den Abstand zwischen den Pads und den Leiterbahnen erhöhen. Dadurch wurde es aber immer schwieriger eine Anordnung zu finden, in der sich alle Bauteile miteinander verbinden ließen, ohne zusätzliche Kontakte einfügen zu müssen, die nur den Zweck haben, beide Platinenseiten miteinander zu verbinden. Bei einer späteren Serienproduktion ist dies aber kein Problem, da diese zusätzlichen Kontakte ohne größeren Mehraufwand erzeugt werden können.

3.1.3 Probleme bei der Realisierung der Schaltung

3.1.3.1 Lösung zu wired-or

Bei meinen ersten Überlegungen zu dem Design der Motorplatine hatte ich die Leitung, welche für die Abschaltung des Motors beim Überschreiten einer der Betriebsparameter sorgen soll, über eine wired-or realisiert. Allerdings habe ich dabei nicht beachtet, dass es sich um Tri-State Ausgänge handelt. Deshalb musste ich noch einen ODER-Baustein, den 7432, einbauen, der für einen sauberen TTL-Pegel sorgt.

3.1.3.2 Lösung zu I²C-Bus

Aber auch bei den Bausteinen zu dem I²C-Bus stellte sich die Realisierung der bidirektionalen Kommunikation als Problem heraus. Um die Betriebsparameter auch noch nach dem Schreiben auf den PCF8574 erhalten zu können, wäre ein Latch, also ein Zwischenspeicher für die Daten, erforderlich gewesen. Aber leider verfügt der PCF8574 nicht über einen Interruptausgang für empfangene Daten, so dass die Ansteuerung des Latches zu schwer gewesen wäre. Außerdem hätte für die Senderichtung auch noch ein Latch eingebaut werden müssen, dessen Ansteuerung auch nicht leicht gewesen wäre. Deshalb habe ich mich dazu entschlossen einen 2. PCF8574 auf die Platine zu setzen. Dieser ist ein A-Typ und hat eine andere festprogrammierte Basisadresse als der Normal-Typ, deshalb kann das Adressfeld vom anderen PCF8574 mitbenutzt werden. Jetzt erfüllt jeder der PCF8574 eine eigene Funktion. Der eine dient nur zum Empfangen der Daten vom Mikrocontroller und der andere nur zum Senden der derzeitigen Betriebsparameter an den Mikrocontroller. Leider wird hierdurch auch die Anzahl der möglichen Motoren auf 8 beschränkt.

3.1.3.3 Der neue Bus

Bei den Versuchen stellte sich der I²C-Bus als zu langsam heraus. Durch den Wechsel zu einem AVR, der in-system-programmable ist, stehen mir nun auch 2 Ports (1Port sind 8 Ein-/Ausgänge) zur Verfügung. Deshalb habe ich 6Bit Adresse, 8Bit senden/empfangen realisieren können. Das erhöht die Geschwindigkeit des Busses von 115KBit/s auf theoretische 216 Mbit/s. Das ergibt sich aus:

- 8ns Durchlaufzeit am 7408
- 9ns Durchlaufzeit am 7421
- 8ns für den zweiten 7408
- 12ns am 74373

macht insgesamt: 37 ns Verzögerung vom Setzen der Adresse zum Anlegen der Daten am Bus.

Aufgrund der geringen Geschwindigkeit des Prozessors können aber nur 64 Mbit genutzt werden.

3.1.3.4 Überlegungen zum 0,1 Ohm Widerstand

Als weitere von mir unterschätzte Schwierigkeit stellte sich der AD-Wandler heraus. Er benötigt einen Widerstand von 0,1 Ohm, den es aber nur sehr teuer mit 0,1% Toleranz und angemessen teuer mit 5% Toleranz gibt. Einer Realisierung des Widerstandes mittels einer 10cm Kupferbahn auf der Platine steht die starke Temperaturempfindlichkeit entgegen.

Ein Widerstand von 1 Ohm hätte nicht nur eine Verzehnfachung der Spannungsdifferenz an den Eingängen des AD-Wandlers zur Folge, sondern würde das Ergebnis um 200 mA verfälschen. Diese Möglichkeit musste ich ausschliessen.

Die Realisierung über Konstantandraht ist wenig für die kommerzielle Realisierung geeignet und lässt sich auch nur mit speziellen Flussmitteln löten.

Bei einer Toleranz von +/-5% schwankt auch die Spannungsdifferenz um +/-5%, bei niedrigen Strömen sogar um bis zu +/-13,3%. Bei einem 8-Bit-Wert sind 5% aber nur 12 Zähler, was vernachlässigbar ist, da es sich nicht auf die oberen 4-Bit auswirkt.