Implementierung einer interaktiven Messsoftware auf dem CAD System CATIA

Diplomarbeit

Verfasser: Ing. Markus Hitter

Betreuung: Dipl.-Ing. Michael Hoffmann
Prof. Dr.-Ing. Jörn Puscher

Diplomarbeit
Implementierung einer interaktiven Messsoftware
auf dem CAD System CATIA™

Markus Hitter
Fachbereich Maschinenbau
Fachhochschule Trier
Schneidershof
D-54298 Trier
http://www.fh-trier.de/~hitterm
hitterm@fh-trier.de

Abgegebene Ausgabe vom 30.8.1996

Vorwort

In den letzten Jahren werden verstärkt CAD (Computer Aided Design) Systeme zur Konstruktion von Teilen verwandt, deren Oberflächen mehr oder weniger frei geformt sind. Dabei müssen Teile, die als reales Modell vorliegen, zur Weiterverarbeitung in das CAD System überführt werden. Eine leichte Änderbarkeit der Konstruktionen ist aber nur möglich, wenn die CAD Modelle aus einer überschaubaren Anzahl geometrischer Primitive besteht.

Eine Überführung eingemessener Punktewolken in geometrische Primitive muß in der Regel „von Hand“ erfolgen, da eine Automatisierung schnell an unüberwindbare Grenzen stößt. „Von Hand“ heißt hier, daß man versucht, am Bildschirm geometrische Primitive zu finden, deren Oberfläche möglichst geringen Abstand zu den eingemessenen Punkten hält. Die Abwägung, welche Punktebereiche zu welchem Primitiv gehören, ist in der Regel nicht einfach und wird durch die zweidimensionale Darstellung am Computermonitor zusätzlich erschwert.

Selbst mit einer solchen Aufgabe konfrontiert, dauerte das Einmessen eines einfachen Automobil–Sitzkissens mehrere Wochen! Zur Verfügung standen eine tastende, CNC–gesteuerte Messmaschine, ein Messprogramm zum automatischen Abtasten viereckiger Flächen und ein 3D CAD Programm.

Aus dem häufigen hin und her zwischen Messmaschine und CAD Rechner entstand die Idee, den CAD Rechner neben die Messmaschine zu stellen und die Messungen direkt aus dem CAD Programm zu steuern. Dabei könnten bereits konstruierte Flächen als Orientierung für weitere Messungen dienen. Ebenso könnten offensichtlich einfache Flächen wie z.B. Ebenen schnell mit wenigen Punkten bestimmt werden. Die Arbeit der Messmaschine könnte auf dem CAD Bildschirm mitverfolgt werden und Fehlmessungen können rechtzeitig erkannt werden. Weitere Vorteile, aber auch Nachteile sind denkbar.

In dieser Diplomarbeit soll nun ein solches interaktives Messverfahren implementiert werden, um Untersuchungen zu erlauben, die eine Aussage über die tatsächliche Effizienz eines solchen Verfahrens erlauben.

Die Vorgehensweise wird dabei wie folgt sein: Zunächst werden die Wünsche und Zielsetzungen der Industrie ermittelt, um die Aufgaben und Ziele abzustecken. Daran orientiert sollen Programme/Module entworfen werden, die eine sinnvolle Untersuchung des Verfahrens erlauben. Die Implementierung muß so gut sein, daß das Verfahren voll genutzt werden kann und keine Nachteile durch Hilfsarbeiten wegen noch fehlender Progammteile entstehen.

Die Diplomarbeit soll weitgehend an der FH Trier durchgeführt werden. Dort steht eine tastende, CNC–gesteuerte Messmaschine (Stiefelmayer) samt Steuerungsrechner (IBM PC/Linux) zur Verfügung. Als CAD Programm kommt CATIA (Dassault Systems/IBM) zum Einsatz, das samt Programmierumgebung genutzt werden kann.

Markus Hitter, im Frühjahr 1996

Inhaltsverzeichnis

Kapitel 1

Anforderungen der Industrie

Um die Zielsetzungen der folgenden Arbeiten festzulegen, wurde eine Anzahl Leute zu diesem Thema befragt, die aufgrund ihrer bisherigen Tätigkeiten bereits über Erfahrungen im einmessen realer Modelle und deren Überführung in ein CAD System besitzen. Dabei wurden sowohl Fragen gestellt, die sich aus der Sicht des Anwenders ergeben, als auch Fragen, inwieweit sich naheliegende Möglichkeiten der Programmierung sinnvoll anwenden lassen.

Ein dabei häufig verwandter Begriff ist die „Punktewolke“. Eine Punktewolke ist eine mehr oder weniger unstrukturierte Sammlung von einigen hundert bis zu mehreren Millionen Punkten in einem CAD Modell. Sie ist häufig das Ergebnis einer Messung und wird dann im CAD System weiterverarbeitet.

Ein weiterer Begriff ist die „Flächenrückführung“. Unter Flächenrückführung versteht man allgemein die Vorgänge, wenn die Oberflächen eines realen Modells in ein CAD Modell überführt werden. Das verwandte Verfahren (z.B. über eine Punktewolke) spielt dabei zunächst keine Rolle.

1.1 Eigene Erfahrungen

Eigene Erfahrungen mit Flächenrückführung erhielt ich Dank einer Idee unseres Professors Lortz, der zur Verbesserung der Mitarbeiterschulung eines Autositzherstellers die Bauteile eines Autositzes in CATIA visualisiert haben wollte. Aus etwas falsch verstandener Aufgabenstellung und der Gewohnheit, im CAD möglichst exakt zu konstruieren, versuchte ich mich nun also an einer genauen Flächenrückführung.

Zur Verfügung stand das CAD System CATIA nebst Ausrüstung zum scannen und vektorisieren gescannter Bilder. Desweiteren wurde eine mechanisch tastende 3D Koordinatenmessmaschine zur Hilfe genommen, mit der viereckige Flächen abgetastet werden konnten. Die aufgetretenen Probleme werden am Beispiel eines Sitzschaumstoffes deutlich.

Zunächst versuchte ich, die Konturen des Bauteils mit Meterstab und einigen selbstgefertigten Schablonen zu erfassen. Dieses Verfahren erzeugt zwar in jedem Fall einfache geometrische Primitive, jedoch besteht beinahe keine Möglichkeit, frei geformte Linien oder Flächen zu erfassen. Auch sind die Messvorgänge relativ umständlich, wodurch sich leicht Messfehler einschleichen. Nach einigen Tagen hatte ich mit diesem Verfahren zwar ein leicht veränderbares CAD Modell, jedoch ließ die Qualität der Freiformflächen stark zu wünschen übrig, denn bereits ein grober optischer Vergleich von Bildschirm mit Realität zeigte offensichtliche Unterschiede.

Um die Freiformflächen zu verbessern, legte ich dann den Schaumstoff in verschiedenen Richtungen auf einen Flachbettscanner. Die resultierenden Bilder wurden mit einem Tracing–

Programm vektorisiert und in das CAD System eingelesen. Die Resultate waren recht zufriedenstellend. Wesentlichstes Problem war, daß der Scanner vom Bauteil aus bestimmten Richtungen Konturen nur sehr unscharf bis gar nicht erkannte. Schätzen dieser Konturen lieferte aber immer noch ein besseres Ergebnis als das Verfahren mit Meterstab und Schablone. Ein Zerschneiden des Schaumstoffs hätte dieses Problem sicher weitgehend behoben, dies war jedoch nicht erlaubt.

Den nächsten Versuch startete mit der tastenden Messmaschine. Eine Tastkugel war für den weichen Schaumstoff nicht geeignet, daher mußte mit einem optischen Taster gemessen werden. Ein optischer Taster kann bauartbedingt nur in eine Richtung messen. Eine solche Messung erzeugte dann eine Datei, die über einige Umwege zu einer Punktewolke im CAD System konvertiert werden kann. Erste Versuche, alle notwendigen Konturen mit einer Messung zu erfassen, schlugen fehl, da die Seiten kaum erfasst wurden. Es konnten also keine Randkurven erzeugt werden, die als Orientierung für die Konstruktion der Seitenflächen hilfreich gewesen wären. Eine Kombination der „Messmaschinenflächen“ mit den Ergebnissen der bisherigen Konstruktion war aus demselben Grund kaum möglich.

Die nächsten (und letzten) Messversuche führte ich dann mit dem optischen Taster durch, der nacheinander in verschiedenen Stellungen an der Messmaschine angebracht wurde. Durch die Veränderung der Tasterstellung änderte sich jedesmal der Nullpunkt des Tasters, deshalb mussten für jede Messreihe einige Referenzpunkte angefahren werden. Zwischendurch wurde ein Hilfsprogramm geschrieben, das die Messdaten teilweise automatisch analysieren konnte und schon bei der Übertragung in das CAD System vorgefertigte SPLINES erzeugte. Im CAD System gelang die Zusammenführung der einzelnen Wolken gut, jedoch erwies es sich als äußerst schwierig, die daraus erzeugten einzelnen Netze miteinander zu verbinden, teilweise war dies sogar unmöglich. Ebenso wurde bei diesem Verfahren keinerlei Regelgeometrie erzeugt womit die Veränderbarkeit des CAD Modells weitgehend verloren war.

Bei den letzten Konstruktionsarbeiten versuchte ich dann, gezielt Punkte aus den vorhandenen Wolken auszuwählen, die sich besonders für die Erzeugung von Regelgeometrien eignete. Dieses letzte Verfahren war das erste, das ein brauchbares Modell erzeugte. Ich erzielte eine weitgehende Übereinstimmung von Massen und Formen zusammen mit relativ einfachen Geometrieelementen. Die Erkenntnis, daß dabei jedoch die meisten Messpunkte ungenutzt verworfen wurden und daß die Auswahl der geeigneten Punkte am Bildschirm umso schwieriger wird, je mehr Messpunkte vorliegen, veranlasste mich zu dieser Diplomarbeit.

1.2 Gespräch mit Fa. ElectroLux, L–Echternach

Bei einem Besuch der Fa. ElectroLux sprach ich mit Herrn Bungartz.

Die Echternacher Abteilung von ElectroLux beschäftigt sich vor Allem mit Rapid Prototyping, aber auch mit der Erstellung von Kunststoff—Spritzgußformen. Typische Bauteile sind Prototypen oder Formen für Gehäuse von Haushaltsgeräten und andere Spritzgußteile ähnlicher Größenordnung. Die Bauteile werden in der Regel im CAD–CAM Verbund gefertigt, d.h. es werden Modelle in einem CAD System (CATIA™) erstellt, von denen dann die erforderlichen Fräsbahnen automatisch berechnet werden.

Bisher werden handgeformte Modelle in das CAD System übertragen, indem sie mit Messschiebern, Radienlehren und Ähnlichem vermessen werden. Häufig übernimmt diese Arbeit bereits der Kunde und hält die Ergebnisse in einer Zeichnung fest. Der Kunde verspricht sich dabei eine einklagbare Grundlage für die spätere Qualitätsprüfung des Endproduktes. Nicht selten erzeugen jedoch die gegebenen Maße ein CAD Modell, das einen etwas anderen optischen Eindruck hinterläßt, als das Original. In einem solchen Fall hat die Einhaltung der gegebenen Maße Vorrang vor der optischen Vollkommenheit.

Allein aus dem geringen Mechanisierungsgrad der Maßerfassung läßt sich ableiten, daß hier noch Möglichkeiten offenstehen, die Vorgänge zu beschleunigen. Daher hat man bei ElectroLux bereits umfangreiche Recherchen betrieben um ein System zu finden, mit dem man schnell und problemlos Formen erfassen kann, die sich für eine Weiterverarbeitung im

CAD System eignen. Eine Erfassung von Punkten allein auf der Formoberfläche ist dabei uninteressant, da diese sich nicht zur Weiterverarbeitung eignen.

Die meisten derzeit verfügbaren Systeme setzen auf die Erfassung möglichst *vieler* Messpunkte, um die Form der Oberfläche möglichst genau zu erfassen. Die Erfassung mehrerer Millionen Punkte in wenigen Minuten scheint dabei kein Problem mehr zu sein. Diese Mengen lassen sich nicht mehr mit einem mechanischen Messverfahren erfassen, sondern werden durch die Auswertung optischer Reflektionen an der Werkstückoberfläche gewonnen. Eine ausreichende Genauigkeit von ca. 0,1 mm wird hierbei erreicht. Häufig wird ein mehr oder weniger einfaches Programm mit angeboten, das die Punktewolken trianguliert, d.h. immer drei Punkte zu einem dreieckigen Oberflächenstück zusammenfasst. Diese triangulierten Oberflächen lassen sich dann auf dem Computerbildschirm rendern und anschaulich darstellen.

Für eine Weiterverarbeitung in Form von Radiusänderungen, Oberflächenglättungen o.Ä. sind diese triangulierten Oberflächen jedoch nicht geeignet. Daher sind bereits erste Programme erhältlich, die versuchen, freigeformte Oberflächenstücke, sogenannte Patches, aus den Punktewolken zu erzeugen. Diese Patches folgen bereits mathematischen Regeln und können damit verändert werden. Auch die Zusammenführung mehrerer Patches zu einer Gesamtoberfläche ist möglich, dies ist in manchen CAD Systemen überraschend schwierig, da diese schon mit geringen Ungenauigkeiten nicht mehr fertig werden.

Diese Möglichkeiten scheinen bisher bei ElectroLux nicht ausreichend, um die recht hohen Anschaffungskosten eines solchen Systems zu rechtfertigen.

Auf die Frage nach einer brauchbaren Lösung forderte Herr Bungartz eine Erfassung der geometrischen Grundelemente wie Ebenen, Kugelsegmente, Rundungsradien etc. oder zumindest Oberflächenstücke, die diesen Grundelementen nahe kommen. Meine Idee, gezielt wenige Punkte zu messen, aus denen dann diese Elemente erzeugt werden können, fand er vielversprechend, jedoch vermutete er Schwierigkeiten, diese Grundelemente bereits im Voraus zu erkennen. Eine Messgenauigkeit von ca. 0,1 mm sollte in jedem Fall ausreichend sein, wichtiger als die exakte Maßhaltigkeit sei in der Regel das optische Erscheinungsbild der Oberflächen bei *ausreichender* Maßhaltigkeit.

Für den Fall der Erfassung weniger Messpunkte mit einer Messmaschine schwebt Herrn Bungartz eine Maschine ohne Antrieb vor, um die Steuerung der Maschine so einfach wie möglich zu halten. Die Messpunkte würden dann von Hand angefahren, aber dennoch sofort vom Computer erfasst.

1.3 Gespräch mit Fa. Bertrand, Stuttgart

Bei Fa. Bertrand in Stuttgart telefonierte ich mit Herrn Schlipf.

Bertrand ist ein bundesweit vertretenes Ingenieurbüro, das für die Automobilindustrie arbeitet. Hauptkunde der Stuttgarter Abteilung ist Mercedes Benz. Bei Bertrand beschäftigt man sich unter Anderem damit, Werkzeuge für die Blechbearbeitung zu konstruieren. Typischerweise erhalten sie dabei ein Gipsmodell eines Blechteils, z.B. eines Autodaches und müssen dieses in ein CAD System übertragen, um es dann mit Falzen etc. auszustatten. Mercedes fordert dabei eine maximale Abweichung der Form von 0,1 mm, auch wenn Funktionalität und optisches Erscheinungsbild des Teils mit einer gröberen Toleranz erreicht werden könnte.

Auch bei Bertrand beschäftigt man sich bereits mit dem Gedanken, Modelle per Messmaschine zu erfassen. Auch hier ergaben die Marktrecherchen, daß die derzeit verfügbaren Systeme hauptsächlich darauf abzielen, große Mengen Punkte zu messen. Herrn Schlipf ist kein System bekannt, das etwas „Intelligenz“ einbringt, d.h. das versucht die Messergebnisse brauchbar zu interpretieren. Auch ihm ist ein System bekannt, das Patches aus den Punktewolken erzeugt, dies sei aber auch das Meiste und noch nicht ausreichend.

Neben der Erfassung neuer Modelle sollten auch bereits erfasste Modelle geprüft werden, indem das aus dem CAD Modell gefrästen Stücke nachgemessen werden.

Da auch die Gipsmodelle nur eine begrenzte Genauigkeit aufweisen, kann sich Herr Schlipf vorstellen, daß die Anforderungen an die Maßhaltigkeit etwas zurückgeschraubt werden, solange die Flächen nur einen hohen Glättungsgrad erreichen.

1.4 Gespräch mit Fa. Bertrand, Köln

Bei Bertrand in Köln telefonierte ich mit Herrn Vogt.

Die Kölner Vertretung von Bertrand beschäftigt sich mit einem ähnlichen Aufgabenbereich wie die Stuttgarter Kollegen. Hauptkunde sind hier jedoch die Ford–Werke.

Nach Ansicht von Herrn Vogt sind die Oberflächen einer Automobilkarosserie zu komplex, um sie in einfache Regelgeometrien zerlegen zu können, obwohl solche auf jeden Fall zu bevorzugen wären. Kompromisse der Designer bezüglich geringerer Komplexität der Oberflächen seien auf keinen Fall zu erwarten. Möglicherweise würde es jedoch Sinn machen, Teile der Oberfläche zunächst mit solchen einfachen Elementen anzunähern, um sie später durch genaueres Nachmessen zu verbessern. Dies und auch Gründe der Qualitätssicherung machen Möglichkeiten zum Nachmessen der Resultate notwendig.

Auch ihm ist bisher keine Software bekannt, die nennenswerte Unterstützung beim Erfassen und nachmessen frei geformter Körper gibt. Auf die Frage nach den Wunschvorstellungen an eine solche Software legte er Wert auf eine einfache und intuitive Benutzerführung, die neben ausreichender Funktionalität ein wesentliches Merkmal sei, um erfolgreich mit einem Programm arbeiten zu können.

1.5 Gespräch mit Fa. Nothelfer, Wadern–Lockweiler

Mein Besuch bei Fa. Nothelfer brachte mich mit mehreren Gesprächspartnern zusammen.

Fa. Nothelfer beschäftigt sich vor Allem mit dem Bau von Presswerkzeugen für Karosseriepressen. Ein typisches Bauteil hat diverse Tonnen Gewicht und wird im Zeitraum von mehreren Wochen gebaut. In der Regel liefert der Kunde (Automobilhersteller) bereits fertige CAD Zeichnungen, die nur noch für die Fertigung aufbereitet werden müssen. Die Qualitätsprüfung erfolgt dann anhand von Probepressungen, deren Resultate dann auf einer Messmaschine geprüft werden. Diese Prüfung erfolgt häufig anhand eines Prüfplanes, den der Kunde mitliefert.

Da Nothelfer vom Kunden bereits fertige CAD Modelle erhält, stellt sich hier die Problematik mit der Komplexität der Flächen nicht. Messmaschinen werden hier „nur“ zur Qualitätsprüfung eingesetzt.

Trotz computergesteuerter Messmaschinen werden die Prüfpläne meist manuell abgefahren, da die automatische Messbahnerzeugung nur selten ausreichend funktioniert. Meist kollidiert dann die Messbahn mit den Spannwerkzeugen oder anderen Hindernissen, die im CAD Modell nicht enthalten sind.

Ein Vorgang ist hier noch erwähnenswert: Das Messen von Beschnittkanten. Dies ist das seitliche abmessen eines Blechrandes. Dies erfordert bei Verwendung des üblichen Kugeltasters eine sehr exakte Tasterführung, um den dünnen Rand nicht zu verfehlen. Durch vorheriges messen der Blechoberfläche könnte dieser Vorgang automatisiert werden, dies würde zwar mehr Messsicherheit geben, aber auch mehr Zeit in Anspruch nehmen. Eine vollständige Automatisierung wird wegen der Unwägbarkeiten nicht möglich sein.

1.6 Gespräch mit Fa. Keiper–Recaro, Kaiserslautern

Zu Keiper–Recaro wurde ich von Herrn Diesch eingeladen.

Keiper–Recaro ist einer der führenden Hersteller von Pkw–Sitzen. In Kaiserslautern befindet sich die Entwicklungsabteilung. Die meisten Teile eines Pkw–Sitzes, wie Verstellschienen, Rahmen etc. werden *nicht* „designed“, sondern im CAD konstruiert und danach gefertigt. Die Messmaschine kommt beim Erfassen der Formen der Sitzschaumstoffe zum Einsatz.

Schon jetzt werden die Formen für diese Schaumstoffe mit einer Messmaschine erfasst. Dabei wird eine Rasterschablone, Rasterabstand meist 100 mm, auf das Messobjekt aufgelegt um die Messpunkte festzulegen. Diese Messpunkte werden dann halb manuell gemessen und in das CAD System übertragen. Diese Arbeiten werden von einem anderen Mitarbeiter erledigt, als dem, der später die Punkte zu Flächenstücken verbindet. Das relativ grobe Raster ist ausreichend, da die Toleranz bei Schaumstoffen mindestens 1 mm beträgt. Schon allein durch die wenigen Messpunkte bedingt hält sich die Komplexität der Flächenstücke in Grenzen.

Ergibt sich beim nachmessen eines gefertigten Schaumstoffs kein ausreichendes Ergebnis, wiederholt sich die Prozedur von vorn, bis das Ergebnis zufriedenstellend ist. Ein Rundlauf dauert dabei ca. eine Woche, üblich sind zwei bis drei Rundläufe.

Häufig ist allein eine optisch ansprechende Erfassung der Teile gefragt, diese werden dann als „bunte Bilder“ Kundenangeboten o.Ä. beigefügt. Die groben und schnellen Messungen hierfür können später nicht weiterverwandt werden.

1.7 Zusammenfassung

Nach den fünf Gesprächen über CAD und Messmaschine hatte ich den Eindruck gewonnen, daß weitere Nachforschungen keine wesentlich neuen Erkenntnisse mehr zu diesem Thema ergeben würden.Alle angesprochenen Ingenieure hatten in etwa die selben Probleme und ähnliche damit verbundenen Wünsche.

Die Anforderungen lassen sich in zwei Gruppen teilen:

1. Es gibt bisher keine schnelle, einfache und kostengünstige Methode, im CAD erstellte Modelle an einem Realteil nachzumessen. Die zur Zeit benutzten Methoden machen kaum Verwendung von den Fähigkeiten eines Rechners, dieser wird bestenfalls als Notizblock angewandt.

   Die Notwendigkeit dieser Forderung ist klar: Der Weg von der Konstruktion in CAD bis zum fertigen Bauteil ist so lang und mit Fehlerquellen behaftet, daß man schon aus Gründen der Qualitätssicherung das Ergebnis prüfen muß. Häufig sind bereits die Eingabedaten (siehe 2.) so unsicher, daß eine mehrfache Nachänderung der Konstruktion unumgänglich ist.

2. Es besteht die Notwendigkeit, vorliegende, reale Modelle in einem CAD System zu erfassen und konstruktiv weiterzuverarbeiten. Dies wird notwendig sein, solange man nicht in der Lage ist, ein Computerbild dreidimensional zu sehen und anzufassen. Die Darstellungsmöglichkeiten auf einem Monitor sind einfach nicht ausreichend um ordentliche Designarbeit zu ermöglichen.

   Die bisherigen Verfahren auf diesem Gebiet sind außerordentlich umständlich und zeitaufwendig. Dem Konstrukteur bleibt bisher kaum etwas anderes übrig, als das Bauteil im Rechner nachzukonstruieren und fortwährend diese Konstruktion mit den Messpunkten zu vergleichen. Dies ist so aufwendig, daß sogar ein ausmessen des Bauteils von Hand häufig schneller und kostengünstiger ist. Die automatisch erstellten Patches stellen lediglich eine Zusammenfassung der Messergebnisse dar und eignen sich für Änderungen der Konstruktion nur bedingt.

Kapitel 2

Der neue Ansatz

Die in weiten Bereichen nicht zufriedenstellenden Möglichkeiten lassen noch zahlreiche Wege offen, die noch erprobt werden müssen.

Der Bereich des Nachmessens bereits konstruierter Teile war bereits Bestandteil einer Gruppe von Diplomarbeiten an der FH Trier. Die Arbeiten sind dort unter dem Namen NCMESS bekannt. Sie implementieren ein Verfahren, bei dem aus einer CAD Konstruktion heraus Messwege erstellt werden. Diese Messwege können dann auf einer Koordinatenmessmaschine weitgehend ohne manuelle Interaktion abgefahren werden. Das Ergebnis einer solchen Messfahrt kann dann in das CAD System rückübertragen und mit der Konstruktion verglichen werden.

Leider erreichte dieser vielversprechende Ansatz bis heute nicht die „Reife“, um in der Industrie Anwendung zu finden.

Der Gegenstand dieser Diplomarbeit ist die Erforschung eines Verfahrens zur Flächenrückführung, d.h. einmessen eines vorliegenden realen Modells in ein CAD System, ohne daß vorher Daten von diesem Modell dort vorliegen.

2.1 Abstraktion und Anforderungskatalog

Die Abstraktion des Verfahrens ist einfach: Erfassung der Oberfläche eines realen Bauteils in einer Form, die von einem CAD System verarbeitet werden kann bei möglichst guter Annäherung an die geometrischen Grundelemente.

Damit ein solches Verfahren effektiv eingesetzt werden kann, muß es folgende Kriterien erfüllen:

1. Es setzt keine modellspezifischen Daten im CAD System voraus.
2. Es produziert Daten im CAD, die sich für die Weiterverarbeitung eignen. Diese Weiterverarbeitung sollte nicht aufwendiger sein, als wenn das Modell direkt im CAD erstellt worden wäre.
3. Die Qualität der Daten muß ausreichend sein, um daraus das Endprodukt ableiten zu können.
4. Der Zeitaufwand für die Erstellung der Daten muß so gering wie möglich sein. Als Maßstab hierfür kann ein CAD Modell ähnlicher Komplexität, aber mit erfundenen Maßen dienen.
5. Die Komplexität des CAD Modells sollte so gering wie möglich sein. Dies gilt sowohl für die Zahl der Konstruktionselemente, als auch für die Komplexität der einzelnen Konstruktionselemente. Zum Beispiel ist für die konstruktion einer Ebene die Verwen-

dung des Konstruktionselementes „Ebene“ (plane) eindeutig einer ebenen Freiformfläche (patch) vorzuziehen.

6. Das Verfahren muß sich für die Erstellung aller notwendigen Konstruktionselemente eignen.
7. Der zusätzliche Aufwand für Maschinen muß gering sein, um rentabel zu bleiben.
8. Der zusätzliche Aufwand für Personal und Schulung sollte aus dem selben Grund gering sein.

2.2 Theorethische Lösungsmöglichkeiten

Es sind zahlreiche Lösungsmöglichkeiten denkbar. Hier eine Aufstellung mit einer kurzen Diskussion:

1. Erfassung der Bauteilgeometrie mittels Messschiebern, Radienlehren etc.

   Diese Möglichkeit ist die zur Zeit am häufigsten angewandte. Sie erfordert einen hohen Zeitaufwand schon bei der Datenerfassung. Häufig werden Maße erfasst, die für die spätere Konstruktion ungeschickt sind, da bei der Erfassung die Vorgehensweise bei der CAD Konstruktion noch nicht festliegt. Das Personal bei der Erfassung ist meist ein anderes als das am CAD System, so daß solche Schwierigkeiten kaum aus dem Weg zu räumen sind. Mit diesem Verfahren können keine frei geformten Oberflächen gemessen werden, diese können nur durch relativ wenige Koordinaten erfasst werden. Werden Messungen und Konstruktion getrennt voneinander durchgeführt, geht der Aufwand für die Niederschrift der Ergebnisse später nutzlos verloren. Sollte sich bei der Konstruktion herausstellen, daß Messungen fehlen, sind diese nur mit sehr hohem Aufwand nachzuholen.

2. Erfassung der Bauteiloberfläche in Form von Punktewolken, mit oder ohne Triangulierung.

   In diesem Bereich treten zur Zeit viele Anbieter in den Markt. Es ist bereits möglich, Bauteiloberflächen innerhalb weniger Minuten zu erfassen, zu triangulieren und auf einem Bildschirm darzustellen. Leider sind solche triangulierten Oberflächen nur sehr eingeschränkt für die Weiterkonstruktion geeignet. Dem Konstrukteur bleibt in vielen Fällen nichts anderes übrig, als diese Oberflächen mit neuen Konstruktionselementen nachzubilden und dann Vergleiche anzustellen. Dies bedeutet einen ähnlich hohen Arbeitsaufwand wie das vorherige Verfahren, deshalb hat sich diese Methode im konstruktiven Bereich wohl noch nicht durchsetzen können.

3. Erfassung von Punktewolken mit anschließender „intelligenter“ Auswertung.

   Es wäre natürlich besonders elegant, die Auswertung der Punktewolken einem Programm zu überlassen. Dieses würde dann die Punktewolke in Gruppen einteilen und mit den geometrischen Grundelementen vergleichen, um das jeweils günstigste zu verwenden. Das Ergebnis wäre ein CAD Modell, das sich konstruktiv nicht von einem frei konstruierten unterscheidet. Da es jedoch eine nahezu unendliche Variationszahl der Grundelemente gibt, man vergleiche allein die Zahl der Funktionen eines guten CAD Programms, dürfte es in naher Zukunft kaum möglich sein ein solches Programm zu erstellen und in endlicher Rechenzeit ablaufen zu lassen. Auch ist mir noch kein Vorstoß in diese Richtung bekannt.

4. Messen von Punkten, so wie sie für die Konstruktion gebraucht werden.

   Zunächst legt der Konstrukteur durch betrachten des realen Modells fest, welches Geometrieelement er verwenden will. Je nach Element misst er dann genau die Punkte mit einer Koordinatenmessmaschine, die für dieses Element gebraucht werden. Wie bei einer freien Konstruktion legt er danach die weiteren Elemente fest und misst jedesmal von neuem nur die benötigten Punkte. Um das Verfahren zu beschleunigen, werden Messmaschine und CAD System zu einem Arbeitsplatz zusammengefasst, die Messmaschine wird interaktiv in das CAD System eingebracht. Diese

Möglichkeit halte ich für sehr vielversprechend. Der Konstrukteur entscheidet während der Konstruktion, wo und wie viele Punkte er einmisst. Damit kann er die Messpunkte so legen, wie sie für die Konstruktion optimal sind.

5. Auswählen von Punkten aus einer Wolke, wie sie für die Konstruktion gebraucht werden

   Natürlich wäre es auch möglich diese Punkte, die in 4. benötigt werden, aus einer eingemessenen Wolke auszuwählen. Dabei müßte die Punktewolke so dicht sein, daß sie für jedes Geometrieelement die notwendigen Konstruktionspunkte enthält. Damit tritt jedoch wieder das Problem der schwierigen Handhabung solcher Wolken am Bildschirm auf. Eine Möglichkeit, die Handhabung zu vereinfachen, wäre z.B. mit der triangulierten Bauteiloberfläche eine Messmaschine im Rechner zu simulieren.

2.3 Gründe für die Auswahl des interaktiven Verfahrens

Wie bereits aus dem Thema der Diplomarbeit zu entnehmen ist, habe ich mich für das interaktive Verfahren (4.) entschieden. Ich halte es bei dem derzeitigen Stand der Technik für besonders vielversprechend. Da bereits ausreichende Erfahrungen mit den unter 1. und 2. genannten Verfahren vorliegen und beide die Ingenieurswelt nicht gerade zu Höhenflügen veranlassten, halte ich es für unumgänglich, einen Versuch mit dem interaktiven Verfahren zu unternehmen.

Mit etwas Glück lassen sich damit die Vorteile von 1., nämlich die gute Anpassung der Messungen an die Bauteilgeometrie, mit Vorteilen von 2., die Verwendung einer Koordinatenmessmaschine verbinden. Durch die Interaktivität fallen Verzögerungen aufgrund unvollständiger oder mangelhafter Messungen fast völlig weg.

Der Aufwand an Maschinen ist nicht höher als bei einem anderen Verfahren, das eine Messmaschine verwendet. Wie sich im nächsten Kapitel herausstellen wird, sind die Probleme der direkten Datenübertragung zwischen Messmaschine und CAD System vor Allem programmiertechnischer Natur. Ein Nachteil ist sicher, daß während der gesamten Konstruktionsphase im CAD die Messmaschine belegt ist und umgekehrt.

Das 3. Verfahren, eine selbstständige Analyse der Bauteiloberfläche durch ein Programm, halte ich heute noch nicht für realisierbar. Auf jeden Fall würde dieses Verfahren ein mittleres bis größeres Software–Projekt bedeuten.

Die 5. Möglichkeit halte ich für eine Notlösung. Häufig verliert man schon den Überblick bei einer normalen 3D–Zeichnung, für unstrukturierte Punktewolken trifft dieses dann besonders zu. Größere Punktewolken können wohl nur automatisch verarbeitet werden.

Kapitel 3

Die Implementierung

Nun steht also fest, daß das Verfahren, das Punkte einzeln oder in keinen Gruppen misst, am erfolgversprechendsten ist. Nachdem ich an dem eingangs erwähnten Sitzkissen quasi gescheitert war, beschloß ich etwa an dieser Stelle der darauffolgenden Überlegungen, das Thema in Form meiner Diplomarbeit zu behandeln.

3.1 Aufgabenspezifizierung

Selbstverständlich müssen hier alle im Anforderungskatalog enthaltenen Punkte vorausgesetzt werden. Um das Verfahren praktikabel zu gestalten, sollte Folgendes zusätzlich beachtet werden:

1. Die im CAD System erzeugten Punkte sollten den normalen Konstruktionspunkten des Systems entsprechen, damit sie auch wie solche weiterverarbeitet werden können.
2. Die Messpunkte sollen in das geöffnete CAD Modell eingebracht werden.
3. Die Übertragung der Messdaten und die Erzeugung der entsprechenden Punkte sollte möglichst verzögerungsfrei geschehen, d.h. der Benutzer sollte das Resultat einer Messung unmittelbar auf dem Bildschirm sehen.
4. Die Punkte sollten so verarbeitet werden, wie sie erzeugt werden, d.h. einzeln.
5. Steuerung der Messmaschine sollte vom CAD Bildschirm aus möglich sein.
6. Es müssen alle notwendigen Verfahrbefehle ausgeführt werden können, also Messfahrt, Eilfahrt auch in schräger Richtung.
7. Die Messgenauigkeit den Maschine sollte voll genutzt werden können.

3.2 Die Messmaschine

Bei der zur Verfügung stehenden Messmaschine handelt es sich um eine 3D Koordinatenmessmaschine der Fa. Stiefelmayer[1]. Sie hat einen Verfahrweg von ca. 1000 mm auf allen Achsen. Die Maschine ist mit Servo–Gleichstrommotoren ausgerüstet und verfügt über ein davon getrenntes Wegerfassungssystem. Dies ermöglicht die Durchführung von Messungen sowohl bei Motorantrieb als auch beim bewegen von Hand.

Ursprünglich wurde die Maschine von einem mitgelieferten Computer gesteuert. Dieser wurde jedoch schon für das NCMESS Projekt durch einen IBM kompatiblen PC ersetzt, um vollen Zugriff auf die Maschinensteuerung zu erlauben. Dieser PC ist mit einer MCU 6/

1. Leider existiert Fa. Stiefelmayer nicht mehr. Aus Fachkreisen ist jedoch zu hören, daß die deren Reste in der Fa. ZEISS Messtechnik aufgegangen sind.

ASM 2003 Regelkarte von Rösch & Walter[1] ausgerüstet, die vollen Hardwarezugriff erlaubt. Diese Regelkarte kann, muß aber nicht, selbstständig die Lage halten und Trapez–Fahrprofile abarbeiten. Ebenso können die Motoren direkt angesprochen werden. Die Regelparameter für diese Karte werden mit einem speziellen Konfigurationsprogramm, das auf diesem PC unter dem Betriebssystem DOS läuft, zusammengestellt und können im Normalfall während des gesamten Betriebs unverändert bleiben. Wegen dieses neuen Steuerungscomputers ist es jedoch zur Zeit nicht mehr möglich, die Maschine mit dem „Joystick“ zu bewegen. Möglicherweise kann diese Funktionalität jedoch wieder implementiert werden.

Als Messtaster stehen ein mechanischer Tastschalter und ein Lichttaster zur Verfügung. Der Lichttaster arbeitet berührungslos mit einem Lichtstrahl, kann jedoch nur in seiner Längsachse messen. Der Tastschalter kann mit Messkugeln verschiedener Durchmesser bestückt werden. Beide Taster können über Schwenkköpfe, Verlängerungen etc. praktisch in jeder Position an der Messmaschine angebracht werden.

3.3 Das CAD System

Als CAD System wird hier an der FH Trier in der Regel CATIA von Dassault Systems/IBM[2] verwandt. Schon das NCMESS Projekt wurde auf diesem System implementiert. CATIA ist im 3D Bereich sehr mächtig und bietet alle erdenklichen Funktionen, es gab daher keinen Anlaß, auf ein anderes CAD System auszuweichen. CATIA läuft auf einer Reihe von UNIX–ähnlichen Betriebssystemen, hier auf IBM's AIX.

Es gibt drei Methoden, die Funktionalität von CATIA zu erweitern: Batch–Programmierung, IUA–Programmierung und GII –Programmierung.

Bei der Batch–Programmierung erstellt man eigenständige Programme in FORTRAN oder C, die CATIA's Libraries nutzen und so Veränderungen an Modellen vornehmen können. Da dies jedoch nicht am geöffneten Modell geschehen kann, ist diese Art der Programmierung nicht geeignet.

IUA–Programme sind in einer FORTRAN–ähnlichen Programmiersprache geschrieben und werden von CATIA zur Laufzeit interpretiert. Es gibt Befehle, um Nachrichten in die Nachrichtenzeile auszugeben, Text und Objektauswahlen vom Benutzer entgegenzunehmen und Modelle darzustellen. Wichtigste Eigenschaft ist jedoch, daß sowohl Unterprogramme aus den CATIA–Libraries als auch selbstgeschriebene Unterprogramme aufgerufen werden können. Datenaustausch erfolgt dabei über die mitgegebenen Parameter. Werden von einem selbstgeschriebenen Unterprogramm aus Funktionen der CATIA Libraries aufgerufen, können so Änderungen am aktuellen Modell vorgenommen werden. IUA ist bereits in der Basis–Lizenz von CATIA enthalten und enthält alle notwendigen Funktionen, daher habe ich diese Art der Programmierung gewählt.

GII–Programmierung ist die mächtigste Möglichkeit, CATIA zu programmieren. Sie bietet vollen Zugriff auf das User–Interface und die CATIA–interne Speicherverwaltung. Da sie jedoch nur mit einer sehr teuren Extralizenz verfügbar ist und IUA ausreichende Möglichkeiten bietet, habe ich auf diese Art der Programmierung verzichtet, um notfalls auch außerhalb der FH Trier Änderungen vornehmen zu können.

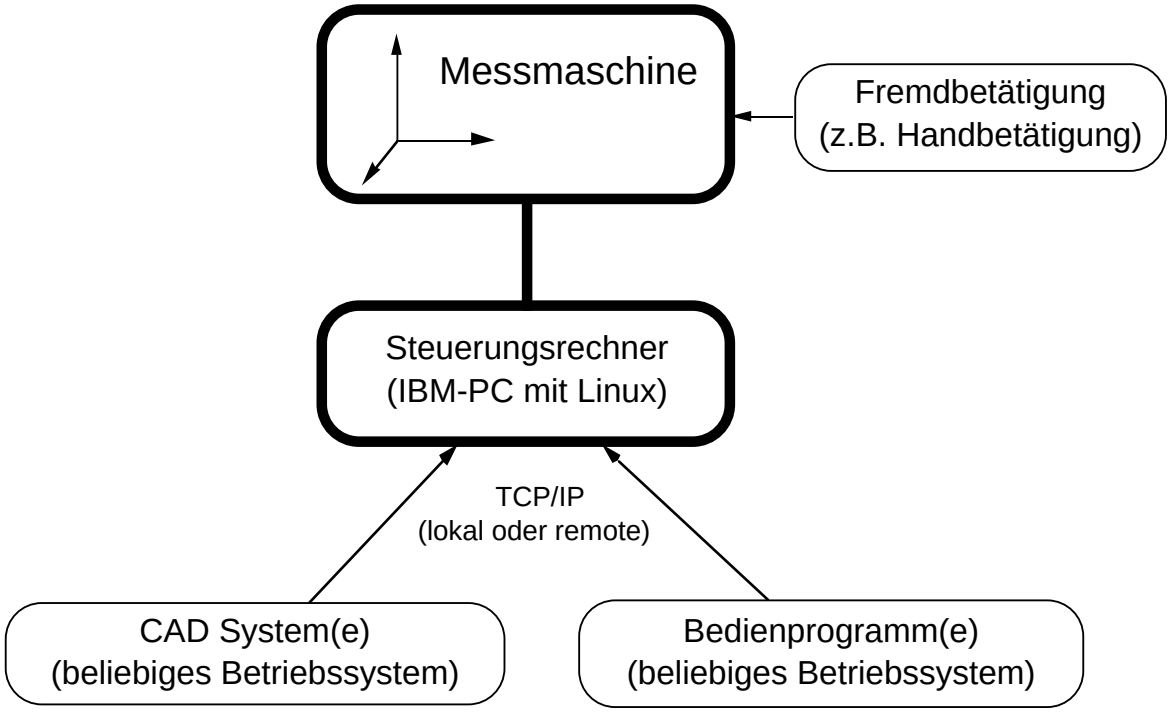
3.4 Arbeitsteilung und Datenübertragung

Da CATIA nicht auf einem IBM Kompatiblen PC läuft und die Regelkarten nur bedingt in eine AIX Workstation eingebaut werden können, liegt es auf der Hand, daß Maschinensteuerung und Erzeugung der Punkte im CAD von zwei getrennten Programmen übernommen

1. Rösch & Walter Industrieelektronik GmbH, In der Spöck 10, 77656 Offenburg
2. IBM Deutschland Informationssysteme GmbH, 70548 Stuttgart
   Siehe: `http://www.ibm.de/go/dm/documente/catia1.html`
   und `http://www.catia.com`

werden müssen. Aus diesem und aus Gründen der Modularität und Erweiterbarkeit habe ich mich entschlossen, jede Aufgabe von einem eigenen Programm übernehmen zu lassen: Ein Programm für die reine Steuerung der Maschine, eins für die Maschinenbedienung (User Interface) und ein IUA Programm in CATIA um die Punkte zu erzeugen und darzustellen.

Alle Programme kommunizieren untereinander mit Nachrichten, die Befehle oder Ergebnisinformationen enthalten. Zentrale (Server) ist dabei das Maschinensteuerungsprogramm, das alle Nachrichten entgegennimmt und gegebenenfalls an die anderen Programme weiterreicht. Ein solches System erlaubt es ohne weiteres, Steuerbefehle von mehr als einem Programm erzeugen zu lassen und auch die Ergebnisse von verschiedenen Programmen auswerten zu lassen. Soll z.B. ein neues Steuergerät eingesetzt werden, muß nur ein zusätzliches Programm gestartet werden, das entsprechende Verfahrbefehle erzeugt, die anderen Programme arbeiten weiter wie bisher. Dies läßt sich dann etwa wie folgt darstellen:



Als Nachrichtenmedium wählte ich TCP/IP–Sockets. Sie sind Standard auf jedem UNIX Betriebssystem und auch auf allen anderen mir bekannten Betriebssystemen (DOS, MacOS...) implementiert. Auch das Internet baut auf TCP/IP auf, daher ist die notwendige Vernetzung der Rechner kein Problem und war hier schon geschehen.

Um die Möglichkeit zu haben, gegebenenfalls ein Bedienprogramm auch auf dem Maschinensteuerungsrechner laufen zu lassen, ist ein Multitasking–Betriebssystem erforderlich. Windows, ein auf IBM PC's gängiges Betriebssystem liefert zwar die notwendigen Voraussetzungen, erzeugte jedoch beim Betrieb des NCMESS Systems immer wieder unerklärliche Probleme (Abstürze). Aus diesem Grund, und weil TCP/IP auf UNIX besser implementiert ist, entschloß ich mich, das Betriebssystem Linux auf dem Steuerungsrechner zu installieren. Linux ist eine frei erhältliche UNIX Variante. Die Entscheidung sollte sich als richtig herausstellen. Außerdem bietet Linux wie alle UNIX Varianten volles Multitasking, d.h. die Programme laufen vollständig unabhängig voneinander und sind nicht auf gegenseitige Zuteilung von Prozessorzeit angewiesen.

Alle Programme sind in der Programmiersprache C geschrieben, auch die Unterprogramme, die vom CATIA IUA Programm aufgerufen werden. Das Bedienprogramm nutzt die Grafiklibrary XFORMS, die für AIX, Linux und andere UNIX Betriebssysteme frei verfügbar ist (frei, solange sie im Ausbildungsbereich eingesetzt wird). Die Portierung eines Programms von AIX nach Linux und umgekehrt ist völlig problemlos, in der Regel kann der selbe Quellcode verwendet werden.

Kapitel 4

Die einzelnen Programme

Nachdem in den bisherigen Kapiteln die Gründe für die Vorgehensweise erläutert wurden, folgen jetzt Beschreibungen für die Programme. Details der Programmierung folgen in einem späteren Kapitel, diese sind für den normalen Anwender weniger interessant. Um den Konflikt mit anderen Namensgebungen zu vermeiden, habe ich das System „interaktive Messsoftware“ getauft, was eigentlich keinen Namen darstellt, aber mit I.Mess. oder IMESS abgekürzt werden kann.

4.1 MESSD

MESSD läuft auf dem Steuerungsrechner und ist das zentrale Programm von IMESS. Es setzt die eingehenden Befehle in Maschinenaktionen um und gibt die Erkenntnisse der Maschine an die Kommunikationspartner weiter.

4.1.1 Hardwarevoraussetzungen

Als einziges Programm ist es auf den Steuerungsrechner gebunden, da es direkt die Regelkarte MCU 6 anspricht. Als Steuerungsrechner wurde während der Diplomarbeit ein 486DX33 Rechner mit 4 MB RAM verwandt, dies war für MESSD voll ausreichend, für den gleichzeitigen Betrieb des Bedienprogramms QUICKMESS auf dem Rechner mußte jedoch auf 8 MB RAM aufgerüstet werden, um ausreichende Geschwindigkeit zu erreichen.

Auf dem Rechner muß natürlich die MCU 6 Steuer/Regelkarte installiert sein. Dies ist ausführlich in der mitgelieferten Anleitung beschrieben. Für diese Diplomarbeit wurde sie so übernommen, wie sie für die NCMESS Arbeiten installiert wurde. Als einzige Änderung wurde die Signalleitung, die mit dem Messtaster verbunden ist, auf die entsprechenden Eingänge aller drei Achsen gelegt. Somit löst ein ansprechen des Tasters einen Interrupt auf Bit 4 an allen Achsen aus, nicht nur auf der X–Achse.

Weiterhin ist die Installation einer Ethernetkarte sinnvoll, um eine ausreichende Kommunikationsgeschwindigkeit mit den anderen Programmen zu gewährleisten. Im vorliegenden Fall belegten sowohl die Ethernetkarte als auch die MCU 6 Interrupt Nr. 3 als Werkseinstellung, dies ist zu prüfen und gegebenenfalls zu ändern.

4.1.2 Softwarevoraussetzungen

Wie bereits in Kapitel 3.4 beschrieben, wurde MESSD für das UNIX-ähnliche Betriebssystem Linux programmiert. Hier wurde die Distribution „Slackware 3.0“ mit einem Kernel Version 1.2.13 (a.out) verwandt. Jede andere Distribution sollte auch geeignet sein. Für den Betrieb von MESSD allein ist eine Grundinstallation ausreichend, TCP/IP ist darin bereits enthalten. Soll auch QUICKMESS auf diesem Rechner betrieben werden, ist X–Windows

erforderlich. Sollen die Programme rekompiliert werden, ist zusätzlich der C Compiler samt Headern und Libraries erforderlich. Alles zusammen war eine 80 MB Festplattenpartition bei der Entwicklung der interaktiven Messsoftware knapp ausreichend.

Für die Konfiguration der MCU 6 sind einige Programme für das Betriebssystem DOS mitgeliefert, die auch benötigt werden. Es bleibt also nichts anderes übrig, als sowohl DOS als auch Linux als Betriebssystem zu installieren. Eine solche Installation ist bei allen bekannten Linux Distributionen beschrieben, die Vorgehensweise ist dort zu entnehmen. Der Rechner kann dann wahlweise gebootet werden, als Bootmanager kommt der „Linux Loader“ (LILO) zum Einsatz.

4.1.3 Justage der Messmaschinenregelung

Sind Hardware und Betriebssysteme installiert, kann der Rechner unter DOS gebootet werden, um die Funktionstüchtigkeit der Hardware zu testen und eine Konfigurationsdatei zu erstellen. Dazu zunächst den Treiber MCUTSR durch starten des Programmes MCBT installieren. Dies sollte vom Installationsverzeichnis der MCU 6 Software aus geschehen.

Dann das Konfigurationsprogramm MCFG starten und alle Parameter für alle Achsen justieren. Die Regler sind so zu justieren, daß sowohl sehr langsame als auch schnelle Fahrt ruckfrei durchgeführt wird. Die Beschleunigungen sollten für alle Achsen gleich eingestellt werden und sich nach der Achse mit der größten Trägheit richten. Im Zweifelsfall sollte lieber zugunsten einer ruckfreien Fahrt entschieden werden, die erzielte Beschleunigung ist zweitrangig.

Alle hier eingestellten Maßeinheiten, Beschleunigungen etc. werden während des gesamten Betriebs der Maschine beibehalten, einzig die Verfahrgeschwindigkeiten „Home Velocity“ und „Jog Velocity“ werden vom Steuerprogramm nach Bedarf gesetzt. „Jog Target Velocity“ sollte auf Null gesetzt werden. Im MCU 6 Handbuch sind zahlreiche weitere Hinweise für die Justagen enthalten, eine Liste der eingestellten Parameter an der genutzten Maschine siehe im Anhang.

Nachdem die Konfiguration gesichert wurde, befindet sich das MCU 6 Betriebssystem in der Datei `RWTOS.BTL` und die Regelparameter in der Datei `SYSTEM.DAT`. Diese beiden Dateien werden von MESSD benötigt und müssen in das Linux-Filesystem übertragen werden. Dies geschieht am einfachsten, indem sie zunächst auf eine Diskette kopiert werden.

4.1.4 Erstellung einer `messdrc`

`messdrc` ist eine Textdatei und beschreibt einige Parameter, die das Steuerungsprogramm je nach verwendetem Taster beachten sollte. Für die Erstellung kann ein Editor, z.B. VI verwendet werden. Es ist auf exakte Schreibweise zu achten und ein Dezimalpunkt statt Dezimalkomma zu verwenden. Eine kurze Erläuterung der verfügbaren Befehle ist hier und in der Beispieldatei enthalten:

Ein # in einer Zeile markiert den Rest der Zeile als Kommentar.

`SpeedLevel` gibt einen Multiplikator für die Verfahrgeschwindigkeiten an. Der Standardwert ist hier `1.0`, sinnvolle Werte bewegen sich zwischen `0.1` und `1.5`, erlaubt sind alle Werte, die mit einer `double` Variable dargestellt werden können.

`TipReversed` gibt an, ob der Taster mit einem öffnenden oder schließenden Kontaktschalter ausgerüstet ist. Diese Variable kann auf `true` oder `false` gesetzt werden, Standardwert ist `false`. Den falschen Wert erkennt man daran, daß das Programm einen eingebauten Taster als angefahren meldet, obwohl er frei ist. Ist *kein* Taster eingebaut, hat diese Meldung jedoch keine Bedeutung.

`TipRadius` gibt den Wert an der für die Radiuskorrektur der Tasterkugel verwendet wird. Für Lichttaster oder wenn die Radiuskorrektur abgeschaltet werden soll, kann dieser Wert auf Null gesetzt werden. Standardwert ist hier `0.0`. Auch hier sind alle Werte erlaubt, die mit einem `double` dargestellt werden können, gängige Werte dürften zwischen `0.0` und `2.5` liegen.

Ist einer der Befehle nicht enthalten, wird der Wert auf den Standardwert gesetzt. Es hat sich

als sinnvoll erwiesen, für jeden verfügbaren Taster eine Datei zu erstellen und je nach Einspannung die entsprechende Datei nach `messdrc` zu linken.

4.1.5 Installation von MESSD

Den Rechner mit Linux booten. Für den Betrieb des Steuerprogramms müssen folgende Dateien bzw. Links in einem Verzeichnis vorliegen:

1. Das Programm selbst.
2. Das MCU 6 Betriebssystem in der Datei `rwtos.btl`.
3. Die Regelparameter in der Datei `system.dat`.
4. Eine Tasterbeschreibungsdatei `messdrc`.

Natürlich muß der ausführende User Ausführungsrecht auf das Programm und Leserecht auf die anderen Dateien haben. Da das Programm direkt auf die Hardware zugreift, muß es setuid `root` sein oder vom User `root` ausgeführt werden. Ist dies nicht der Fall, erhält man eine Fehlermeldung, die auf die nicht vorhandenen Hardwarezugriffsrechte hinweist, sobald man das Programm startet.

Die Dateien `rwtos.btl` und `system.dat` werden einfach von der DOS-Diskette heruterkopiert, es ist darauf zu achten, daß beide Namen jetzt ausschließlich aus Kleinbuchstaben bestehen.

4.1.6 Starten von MESSD

Das Steuerungsprogramm sollte jetzt lauffähig sein, auf jeden Fall aber eine sinnvolle Fehlermeldung abgeben, falls eine Unstimmigkeit auftritt. Das Programm benutzt keine grafische Oberfläche, daher läuft es idealerweise von der Konsole, kann aber ebensogut remote über ein Terminal gestartet werden. Im letzteren Fall ist zu beachten, daß die Maschine unter Umständen auch aus großer Entfernung, d.H. ohne Sichtkontakt, gestartet werden kann, wenn die Stromversorgung der Regelelektronik eingeschaltet ist.

MESSD kann durch Eingabe des Programmnamens gestartet werden. Es können beim starten einige Parameter mitgegeben werden, die die Menge der ausgegebenen Meldungen beeinflußt, die für die Fehlersuche hilfreich sein können:

`-h` startet das Programm nicht, sondern gibt eine Kurzauskunft über die möglichen Parameterangaben.

`-v` startet das Programm nicht, sondern gibt die Version des Programms auf den Bildschirm aus.

`-dc` kopiert die ein- und ausgehenden Nachrichten auf den Bildschirm. Dies ist nützlich, wenn Kommunikationsprobleme mit den anderen Programmen auftreten.

`-dm` gibt Auskunft über die eingeleiteten Verfahrbefehle und gemessenen Koordinaten. Dies ist nützlich, wenn Zweifel darin bestehen, ob die eingehenden Befehle vom Programm richtig verarbeitet werden.

`-dt` gibt regelmäßig die Transputer Bits auf den Bildschirm aus. Dies ist nützlich, um Probleme rund um den Transputer der MCU 6 aufzudecken, z.B. fehlerhafte Verkabelung, mangelhafte Stromversorgung, notwendiger reboot der MCU 6 etc. Die Bedeutung der Bits sind dem MCU 6 Handbuch zu entnehmen.

`-p` gibt Informationen zur Ablaufgeschwindigkeit des Programms. Dies ist eigentlich nur beim programmieren nützlich oder wenn Zweifel an der ausreichenden Rechenleistung des verwendeten Rechners besteht. Rundenzeiten unter 0,5 Sekunden sind in Ordnung, Zeiten darüber lassen das Programm unter Umständen zu spät auf veränderte Bedingungen an der Maschine reagieren, z.B. wenn der Taster angefahren ist.

`-r` erzwingt einen reboot der MCU 6, d.h. ein erneutes einlesen der Dateien `rwtos.btl` und `system.dat`. Dies ist z.B. erforderlich wenn eine neue `sys-`

`tem.dat` eingelesen werden soll, ohne die Stromversorgung der Regelelektronik abzuschalten oder die MCU 6 nur noch teilweise richtig reagiert.

Es können auch mehrere Optionen gleichzeitig angegeben werden, keine der Optionen beeinflußt jedoch die Funktionsweise des Programms im normalen Betrieb, was den eigentlichen Aufgabenbereich des Programms betrifft.

Um die Konfigurationsdatei `messdrc` zu wechseln bzw. erneut zu lesen, ist das Steuerungsprogramm neu zu starten. Alle erkannten Befehle aus dieser Datei werden beim starten des Steuerungsprogramms auf dem Bildschirm ausgegeben.

4.1.7 Betrieb von MESSD

MESSD sollte eigentlich ohne zutun des Users unauffällig laufen. Nützlich wird das Programm erst in Verbindung mit den anderen Programmen. Die Bildschirmausgaben dienen lediglich der Zusatzinformation. Die Meldung „`messd OK`“ gibt an, daß die Initialisierungsphase erfolgreich durchlaufen wurde und das Programm betriebsbereit ist.

Das Programm erlaubt keine Interaktionen außer unmittelbar zu Beginn: Das Programm fragt, ob es die Home Position, den fest eingebauten Maschinennullpunkt anfahren soll. Wird die Frage mit Nein beantwortet, liegt der Maschinennullpunkt an der momentanen Position des Tasters, dies ist in der Regel ausreichend. Wird die Frage jedoch mit Ja beantwortet, sollte vorher sichergestellt sein, daß ein anfahren des Nullpunktes nicht zu Kollisionen irgendeiner Art führen, da diese vom Programm nicht erkannt werden können. Also besser vorher des Messtaster ausbauen, wenn man Schäden am Taster vermeiden will.

Erhält das Programm eine Verfahrbefehl, schaltet es selbstständig die Lageregelung ein und führt diesen Befehl aus. Dabei werden die mechanischen Begrenzungen erkannt und die Maschine beim Erreichen einer solchen gestoppt bzw. gar nicht erst gestartet. Der nächste Verfahrbefehl kann nur noch in einer Richtung ausgeführt werden, die von der Begrenzung weg zeigt.

Spricht der Messtaster an, wird ein akustisches Signal ausgegeben und der gemessene Punkt an die anderen Programme weitergereicht. Wurde die Maschine nicht mit den Motoren bewegt, wird die Radiuskorrektur mit Hilfe des Bahnvektors der Bewegung auf dem letzten Millimeter berechnet. In diesem Fall kann auch nur ein Punkt pro Sekunde gemessen werden, um Probleme z.B. mit zittrigen Fingern zu vermeiden. Wurde die Maschine *mit* den Motoren bewegt, berechnet sich die Radiuskorrektur aus dem aktuellen Verfahrbefehl und es wird der Rückzug des Tasters 0,5 mm hinter den Messpunkt eingeleitet. Unter extremen Umständen, z.B. wenn Werkstückoberfläche und Bewegungsbahn einen sehr spitzen Winkel bilden, kann dieser Rückzug erfolglos verlaufen, d.h. der Taster bleibt angefahren und muß von Hand dort wegbewegt werden.

Ist der Messtaster zu Beginn eines Verfahrbefehls angefahren, wird die Maschine nicht gestartet, sondern eine entsprechende Fehlermeldung ausgegeben. Die Maschine kann dann nur noch von Hand bewegt werden, bis der Taster frei ist.

Wird die Maschine mit den Motoren bewegt, ist die Lageregelung eingeschaltet und Bewegungen von Hand werden von der Regelung verhindert. Bleiben die Motoren einige Sekunden ungenutzt, schaltet sich die Lageregelung selbstständig ab und die Maschine kann wieder von Hand bewegt werden.

Während des Betriebs gibt das Programm Warnungen aus, wenn es auf Unstimmigkeiten irgendeiner Art trifft. Sollte also etwas nicht wie erwartet funktionieren, zunächst die Ausgaben des Programmes auf Hinweise prüfen um die Fehler dann zu beseitigen. Klappt die Kommunikation mit der MCU 6 schon zu Beginn nicht, kann man deren Initialisierung wiederholen. Können Unstimmigkeiten nicht während des Betriebs beseitigt werden, ist das Programm neu zu starten, dies war während der Erprobung des Programms aber nur in Ausnahmefällen erforderlich. Können Unstimmigkeiten offensichtlich nicht während des Betriebes behoben werden, bricht das Programm selbstständig nach Ausgabe der Fehlermeldung ab.

4.2 QUICKMESS

Hauptaufgabe von QUICKMESS ist es, eine Bedienoberfläche für MESSD zur Verfügung zu stellen, muß jedoch keineswegs auf dem gleichen Rechner wie MESSD laufen. Alle Eingaben an der grafischen Benutzeroberfläche werden in entsprechende Befehle umgesetzt, die an MESSD weitergereicht werden. Ebenso werden Meldungen von MESSD über die aktuelle Position der Maschine und gemessene Punkte angezeigt.

4.2.1 Hardwarevoraussetzungen

Für QUICKMESS sind keine außergewöhnlichen Hardwarevoraussetzungen erforderlich, der Computer muß lediglich in der Lage sein, ein TCP/IP Verbindung zum Messmaschinenrechner aufzubauen. Dies ist der Fall, wenn QUICKMESS auf dem Messmaschinenrechner selbst läuft oder der Rechner sich im selben Netzwerk befindet.

4.2.2 Softwarevoraussetzungen

Da QUICKMESS eine grafische Benutzeroberfläche benutzt, ist eine solche erforderlich. Prinzipiell wäre es möglich, das Programm auf eine beliebige Oberfläche zu portieren, die gleichzeitig TCP/IP bedienen kann, ich halte jedoch UNIX mit X–Windows für sehr empfehlenswert, schon weil das CAD System die gleiche Oberfläche benutzt. Im momentanen Quellcode wird die XFORMS[1] Library verwendet, die für alle gängigen X–Windows Systeme verfügbar ist. Den Disketten zu dieser Diplomarbeit liegen fertige Programme für AIX und Linux bei.

4.2.3 Installation von QUICKMESS

QUICKMESS kann an jede beliebige Stelle kopiert werden und benötigt neben dem Ausführungsrecht keine besonderen Rechte oder Einstellungen. Soll auch das formschöne Logo zu sehen sein, muß sich die Datei `pict-mb.xbm` im aktuellen Verzeichnis befinden, d.h. in dem Verzeichnis, von dem aus QUICKMESS gestartet wird.

4.2.4 Starten von QUICKMESS

Da QUICKMESS auf eine TCP/IP Verbindung zu MESSD angewiesen ist, kann es nur gestartet werden, wenn MESSD bereits läuft. MESSD muß auf einem Rechner laufen, der mit dem Rechner, auf dem QUICKMESS läuft, eine TCP/IP Verbindung aufbauen kann.

Wiederum ist es moeglich, das Programm durch einfache eingabe des Namens zu starten. Es wird dann versucht, eine TCP/IP Verbindung zu MESSD auf `localhost` aufzubauen, also zu dem eigenen Rechner. Dies ist sinnvoll, wenn QUICKMESS auf dem Steuerungsrechner gestartet wird. Soll jedoch die Verbindung zu MESSD auf einem anderen Rechner aufgenommen werden, muss man dessen Nummer/Namen angeben, z.B. `quickmess mbpc18` oder `quickmess mbpc18.fh–trier.de` oder `quickmess 143.93.62.168`.

Wie bei MESSD können auch hier Optionen mitgegeben werden, um (hoffentlich) hilfreiche Ausgaben zu produzieren:

- `-h` startet das Programm nicht, sondern gibt eine Kurzauskunft über die möglichen Parameterangaben.
- `-v` startet das Programm nicht, sondern gibt die Version des Programm aus.
- `-d` gibt Informationen zur Fehlersuche, kopiert unter Anderem die ein- und ausgehenden Nachrichten auf den Bildschirm. Dies ist nützlich, falls Probleme mit dem Betrieb des Programmes auftreten.
- `-p` gibt Informationen zur Ablaufgeschwindigkeit des Programms. Nicht unbedingt

1. XFORMS: Copyright © T.C. Zhao und Mark Overmars.
Siehe `http://bragg.phys.uwm.edu/xforms` und `ftp://ftp.cs.ruu.nl/pub/XFORMS`

notwendig, war aber schon 'mal programmiert.

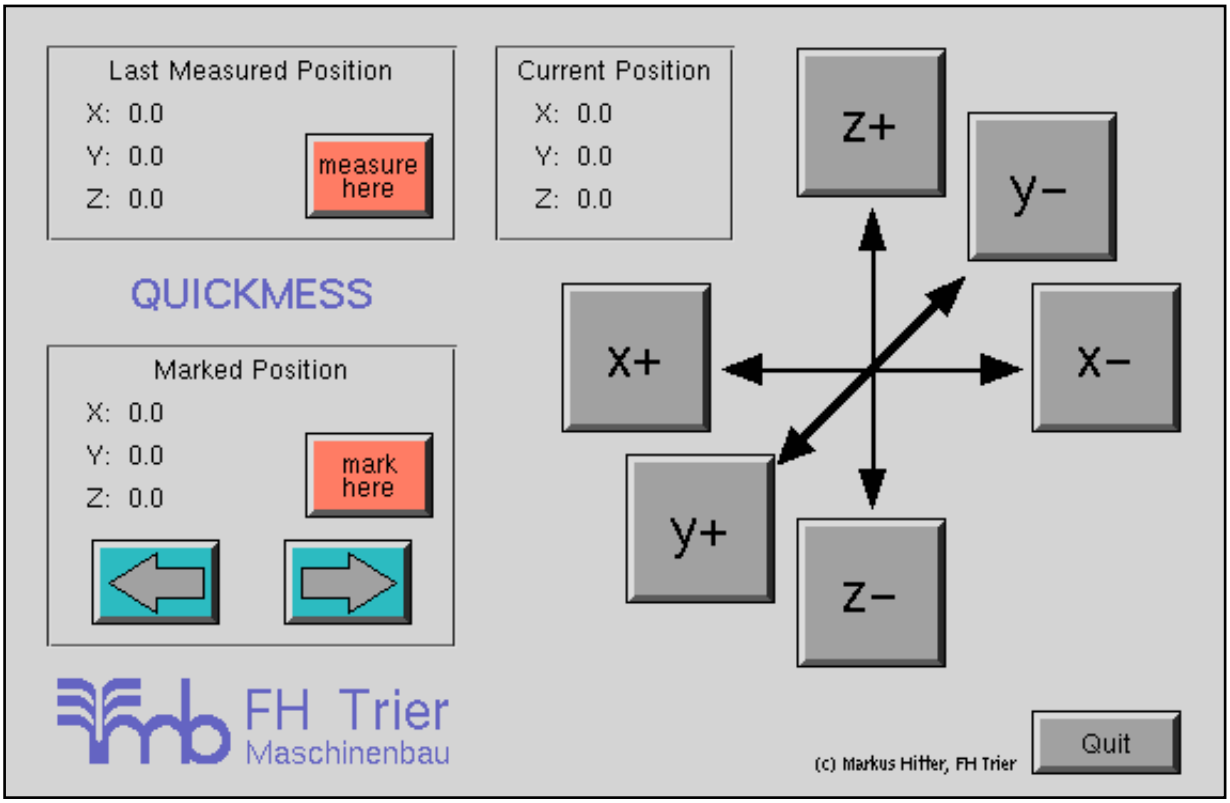
Auch hier beeinflussen die Optionen nicht die eigentliche Arbeitsweise des Programms, lediglich die richtige Angabe des Rechners, auf dem MESSD läuft, ist entscheidend.

4.2.5 Betrieb von QUICKMESS

Bemerkung:

Geht die Verbindung zu MESSD verloren, z.B. weil MESSD gestoppt wurde, wird auch QUICKMESS selbstständig beendet. Dies kann u.U. zu unerwartetem Programmabbruch führen. Der Betrieb kann jedoch wieder aufgenommen werden, nachdem die Verbindung wieder hergestellt wurde. Da alle Informationen bei MESSD gespeichert werden, kann also evtl. sogar der Maschinennullpunkt weiter gehalten werden.

Hat man QUICKMESS erfolgreich gestartet und konnte das Programm Verbindung mit MESSD aufnehmen, sieht man die grafische Oberfläche:



Die verschiedenen Bereiche haben folgende Bedeutung:

Der Bereich „Last Measured Position“:

Hier werden die Koordinaten des letzten Messpunktes angezeigt. Zu Beginn enthält dieses Feld natürlich keinen gültigen Wert. Der Button „measure here“ kann dazu verwendet werden, an der aktuellen Position der Maschine einen Messpunkt ohne Radiuskorrektur zu generieren, ohne daß der Messtaster anfährt. Dies ist z.B. nützlich, wenn zu Beginn schnell die groben Abmaße des Bauteils markiert werden sollen.

Der Bereich „Current Position“:

Hier wird immer die aktuelle Position der Maschine angezeigt, wie sie von MESSD gemeldet wird. Dies wird ca. einmal je Sekunde erneuert, auch wenn die Maschine von Hand bewegt wird. Eine Radiuskorrektur ist hier nicht enthalten.

Der Bereich um das Koordinatenkreuz:

Mit diesen 6 Buttons wird die Maschine verfahren. Es ist sinnvoll, den Rechner, auf dem QUICKMESS läuft, so vor der Messmaschine anzuordnen, daß die optische Per-

spektive auf dem Bildschirm der Perspektive zur Messmaschine entspricht.

Drückt man einen Mausknopf auf einem der Buttons, fährt die Maschine in die entsprechende Richtung los. Läßt man den Mausknopf wieder los, hält die Maschine wieder an. Rutscht man bei gedrücktem Mausknopf von dem Button herunter, fährt die Maschine weiter, auch wenn die Maus dann losgelassen wird. In diesem Fall kann die Maschine nur angehalten werden, indem man erneut einen der Buttons betätigt.

Die Verfahrgeschwindigkeit kann durch die Wahl verschiedener Mausknöpfe beeinflußt werden. Der rechte Mausknopf wählt den Eilgang (100% der Maximalgeschwindigkeit), der linke Mausknopf den Messgang (2% der Maximalgeschwindigkeit). Der mittlere Mausknopf wählt eine mittlere Geschwindigkeit (20% der Maximalgeschwindigkeit).

Natürlich stoppt die Maschine in jedem Fall, wenn der Taster angefahren wurde oder eine der Endlagen erreicht wurde. Dies wird von MESSD veranlaßt und wurde dort beschrieben.

Der Bereich „Marked Position“:

Die Anzeigen und Buttons in diesem Bereich sind für die Realisierung von Schrägfahrt vorgesehen, also ein verfahren der Maschine in einer Richtung, die nicht unbedingt einer der Koordinatenachsen entspricht. Dies ist notwendig, wenn man Flächen in Normalenrichtung anfahren will, um eine korrekte Radiuskorrektur zu erhalten.

Der Button „mark here“ markiert zunächst die aktuelle Position der Maschine, dies hat für anderen Programme keine Auswirkungen. Diese Position wird dann in Form von Koordinaten angezeigt.

Fährt man dann die Maschine mit den normalen Buttons auf eine andere Position, hat man den Verfahrvektor für die Schrägfahrt festgelegt. Mit den beiden Pfeilbuttons in diesem Bereich kann die Maschine dann auf dem Vektor von der markierten Position zur aktuellen Position verfahren werden. In welche Richtung die Maschine sich auf diesem Vektor bewegt, hängt von dem Verhältnis der beiden Positionen ab, u.U. ist ein antippen eines der beiden Buttons notwendig, um das herauszufinden. Da die Vefahrrichtung bei Verwendung der beiden Buttons immer auf diesem Vektor bleibt, bleibt die Verfahrrichtung erhalten, bis wieder einer der sechs Buttons zum Verfahren entlang der Koordinatenachsen benutzt wird.

Eine typische Vorgehensweise ist dabei wie folgt: Zunächst den Taster in die Nähe des gesuchten Messpunktes fahren, ohne daß der Taster anspricht. Hier den Button „mark here“ drücken. Dann den Taster entlang der Koordinatenachsen von der Fläche entfernen, bis er sich wieder auf dem Vektor der gewünschten Messrichtung befindet. Nun kann mit den beiden Pfeilbuttons zur Fläche zurückgefahren werden, bis der Taster anspricht. Der Messpunkt wird dann mit der korrekten Radiuskorrektur aufgenommen.

Der Button „Quit“:

Beendet das Programm. Dies hat für die anderen Programme keine Auswirkungen.

An der Kommandozeile (dort, wo das Programm gestartet wird), kann der Benutzer nicht in den Programmablauf eingreifen. Hier werden jedoch Fehlermeldungen und die Informationen zur Fehlersuche ausgegeben. Sollten also Probleme im Ablauf des Programms auftreten, werden hier Fehlermeldungen zu lesen sein.

4.3 CATIA Modul

Aufgabe des CATIA Moduls ist es, die gemessenen Punkte in das CAD Modell zu übertragen. Dies geschieht unmittelbar nach erfolgter Messung. Zur besseren Orientierung kann auch der Messtaster an der stets aktuellen Position dargestellt werden. Wegen der eingeschränkten Programmiermöglichkeiten der grafischen Oberfläche in CATIA kann die Messmaschine von hier nicht bedient werden, es ist daher empfehlenswert, zugleich QUICKMESS auf dem Rech-

ner laufen zu lassen oder die Messmaschine von Hand zu bewegen.

4.3.1 Hardwarevoraussetzungen

Sind die Voraussetzungen für den Betrieb von CATIA erfüllt, kann auch die interaktive Messsoftware auf dem Rechner betrieben werden. Zumindest sind mir keine Konfigurationen bekannt, bei denen dies nicht der Fall ist.

4.3.2 Softwarevoraussetzungen

Hier gilt das selbe wie bei der Hardware: läuft CATIA auf dem Rechner, kann auch das CATIA Modul von IMESS darauf betrieben werden. Es ist nur die Basislizenz erforderlich, die IUA und die Funktion POINT enthält. Die der Arbeit beiliegenden Binärfiles laufen auf einer RS6000 und wurden unter AIX 3.2.5 und CATIA 4.1.5 getestet, sollten aber auch mit AIX 4 und CATIA Version 3 funktionieren. Für andere Betriebssysteme muß der Load neu kompiliert werden (siehe Kapitel: Hinweise für Programmierer).

4.3.3 Installation des CATIA Moduls

Wie die gesamte Installation von CATIA ist auch die Installation von Zusatzmodulen relativ kniffelig und sollte von einer Person durchgeführt werden, die sich mit der Installation von CATIA auskennt. Das CATIA Modul besteht aus einer IUA Prozedur (`IMESS.iuaproc`)und einem Load (`IMESS`). Hier wird die Installation für einen einzelnen User in CATIA 4.1.5 beschrieben und kann für eine systemweite Installation oder andere CATIA Versionen sinngemäß abgewandelt werden.

Zunächst das CATIA Environment laden und den Bestimmungsort der IUA Prozedur bestimmen: `catpath -l catia.IUAPROC_USR` gibt einen oder mehrere Pfade aus. Ist dies nicht der Fall, muß in der `USRENV.dcls` einer angelegt werden (z.B. mit `catia.IUAPROC_USR = '$HOME/code/iua';`). Dann die IUA Prozedur dorthin kopieren. Innerhalb von CATIA kann nun mit der Funktion IUA/FILE/PROC den Pfad für die zusätzlichen Module ausgewählt werden, der Pfad ist dann mit `USER FILE` markiert. Um die Installation zu prüfen, kann man in der Funktion IUA/EXECUTE den (Such-)String `IMESS` angeben, das Modul der interaktiven Messsoftware sollte gefunden werden.

Um CATIA noch den Rechner bekanntzugeben, auf dem MESSD läuft, muß die IUA Prozedur noch in einer Zeile editiert werden. Dies kann von innerhalb CATIAs mit der Funktion IUA/MODIFY oder (besser) mit einem beliebigen Texteditor aus dem Betriebssystem heraus geschehen. Eine Zeile lautet: „`LOAD imess ENTRY im_open 'mbpc18 '`“, hier muß der Name `mbpc18` in den Namen des Rechners geändert werden auf dem MESSD laufen wird, analog zu der entsprechenden Angabe beim starten von QUICKMESS. Das Leerzeichen hinter dem Namen nicht vergessen, sonst kann der Name nicht erkannt werden. Die Prozedur kann nun probeweise gestartet werden (mit `/m imess`), man muß sie jedoch schon bei der ersten Frage durch angabe von NO beenden, da der benötigte Load noch fehlt.

Die Installation des Loads geht ähnlich: mit `catpath -l catia.IUAMODULE_USR` das passende Verzeichnis finden und den Load (`IMESS`) dorthin kopieren. Dann mit der Funktion IUA/FILE/LOAD den Pfad auswählen. Um sicherzustellen, daß der Load zwischen zwei Aufrufen nicht aus dem Speicher entfernt wird (und damit notwendige Daten verloren gehen), muß folgende Zeile in die USRENV.dcls eingefügt werden:

```
catia.IUA_UNLOAD_SHARED = FALSE;
```

Da das Modul sehr oft neu gestartet wird, empfiehlt sich die Installation als permanente Funktion, sie kann dann zu jedem Zeitpunkt, also auch von innerhalb anderer Funktionen aufgerufen werden (ähnlich z.B. NoShow). Dies ist nicht unbedingt notwendig, aber relativ einfach, man muß nur folgende Zeilen in die `USRENV.dcls` einfügen:

```
/* Put IMESS into the ToolBar as a permanent function. */
CATFRM.PERMFUNC.MAXCOUNT     = 21;
CATFRM.PERMFUNC(21).TYPE     = 'PUSH';
CATFRM.PERMFUNC(21).LABEL    = 'IMESS';
CATFRM.PERMFUNC(21).COMMAND  = 'm imess';
CATFRM.PERMFUNC(21).CUS_NAME = 'IMESS';
CATFRM.PERMFUNC(21).HELP     = 'Receive Points from IMESS';
```

Die 21 ist dabei die Zahl, die man erhält, wenn man das Ergebnis von `catpath -l CATFRM.PERMFUNC.MAXCOUNT` um eins erhöht. In diesem Fall ist IMESS der 21. Button in der Reihe der permanenten Funktionen.

Nun sollte die IUA Prozedur ausgeführt werden können, nachdem man CATIA neu gestartet hat um die Änderungen in der USRENV.dcls einzubringen.

4.3.4 Starten des CATIA Moduls

Die IUA Prozedur kann durch klicken auf den neu installierten Button bei den permanenten Funktionen oder durch Eingabe von `/m imess` in der Eingabezeile gestartet werden. Treten Probleme bei der Ausführung aus, ist es empfehlenswert, die Prozedur über die Funktion IUA/EXECUTE/CTRLON zu starten, dann produziert CATIA zusätzliche Fehlermeldungen, die hilfreich sein können. Parameter können keine mit angegeben werden, der Name des Messmaschinenrechners ist fest in der IUA Prozedur codiert, wie bei der Installation des Moduls beschrieben.

Natürlich muß vor Ausführung der Prozedur MESSD auf dem Steuerungsrechner gestartet sein, sonst tritt im Laufe der Abarbeitung der Fehler: „`no Partner on Port 1066`“ auf, da die Kommunikation nicht aufgenommen werden konnte.

Soll die Visualisierung des Messtasters aktiviert werden, muß in dem aktuellen Modell ein Detail mit (exakt) dem Namen `MEASURE TOOL` existieren. Dies kann mit der Funktion DETAIL/CREATE angelegt werden, aus einer Library in das Modell geholt werden oder bereits im Urmodell enthalten sein. Gegebenenfalls ist der Name des Details mit der Funktion DETAIL/MANAGE/RENAME zu korrigieren. Existiert das Detail nicht, wird der Taster nicht visualisiert, alle anderen Funktionen bleiben jedoch erhalten. Bei der Visualisierung entspricht der Nullpunkt des Detail-Workspace immer der momentanen Position der Maschine. Die Zeichnung des Tasters sollte nicht zu kompliziert ausfallen, da sie häufig verschoben werden muß (jede Sekunde einmal) und eine aufwendige Zeichnung unter Umständen nicht schnell genug dargestellt werden kann. Eine Kugel um den Nullpunkt mit Zylinder als Taststift hat sich bewährt.

4.3.5 Betrieb des CATIA Moduls

Nach dem starten des Moduls wird man zunächst gefragt, wie viele Punkte man messen möchte. Die Angabe von 0 oder NO führt zu einem Abbruch der Funktion, YES entspricht der Eingabe einer eins. Bei Eingabe einer Zahl größer als eins bricht die Funktion erst nach dem Erreichen der gewünschten Punktezahl ab. Natürlich kann die Funktion jederzeit erneut gestartet werden, um weitere Punkte zu messen.

Danach wird die Verbindung zu MESSD aufgenommen und das Tasterdetail im Modell dargestellt, falls vorhanden. Auch ohne Tasterdetail werden immer die aktuellen Maschinenkoordinaten in der Ausgabezeile gezeigt. Das Modul wartet nun auf eingehende Messpunkte. Wurde die gewünschte Anzahl Messpunkte erreicht, wird die Funktion selbstständig beendet.

Üblicherweise wird dann die Messmaschine von Hand bewegt oder QUICKMESS in den Vordergrund geholt, um die Punkte zu messen. Die Messpunkte erscheinen automatisch im CATIA Modell, sobald sie gemessen sind. Währenddessen kann das Modell wie gewohnt gezoomt und gedreht werden. Aus programmiertechnischen Gründen geschieht dies jedoch

etwas ruckender als gewohnt, ebenso wird ständig ein „Buffer Refresh“ ausgelöst.

Auch wegen der mangelnden Programmiermöglichkeiten des CATIA User Interface kann die Funktion vor Erreichen der gewünschten Messpunktezahl nur durch die INTERUPT Funktion unterbrochen werden. Dann sind jedoch die Punkte, die seit dem letzten Starten der Funktion gemessen wurden, verloren. Im Zweifelsfall müssen also nutzlose Punkte gemessen werden, um die Funktion ordentlich zu beenden, z.B. durch antippen des Tasters mit dem Finger oder dem „measure here“ Button von QUICKMESS.

Kapitel 5

Nutzung der Programme

Nun ist es wohl an der Zeit, das Zusammenspiel der Programme aufzuzeigen. Auch wenn dies nicht mehr zum eigentlichen Thema der Diplomarbeit gehört, wurden doch schon während der Entwicklung der Programme erste Erfahrungen gesammelt, die hier erläutert werden sollen.

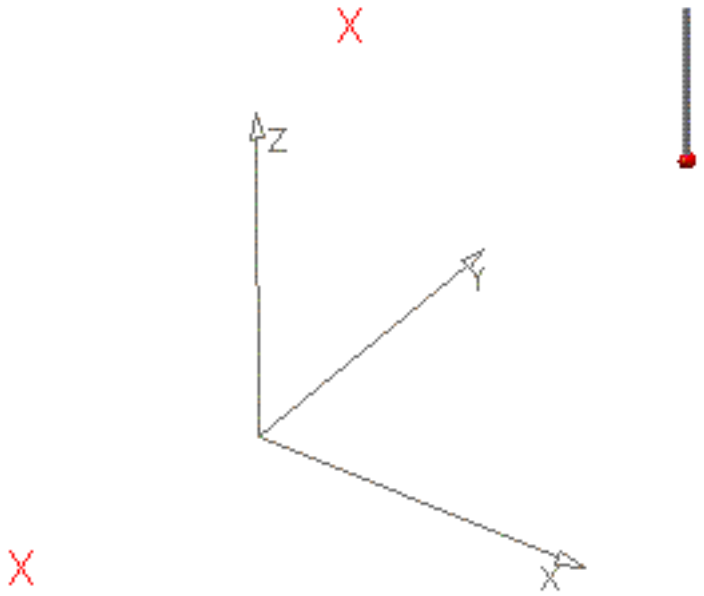
5.1 Typische Vorgehensweise anhand eines Beispiels

Das einfachste Beispiel ist natürlich immer ein Würfel. Dieser ist hier mit einer Bohrung und einer Verrundung ausgestattet. Da die Komplexität des realen Modells und damit auch des CAD Modells wenig Einfluß auf die Aktionen mit der Messmaschine hat, soll dieses Beispiel hier genügen.

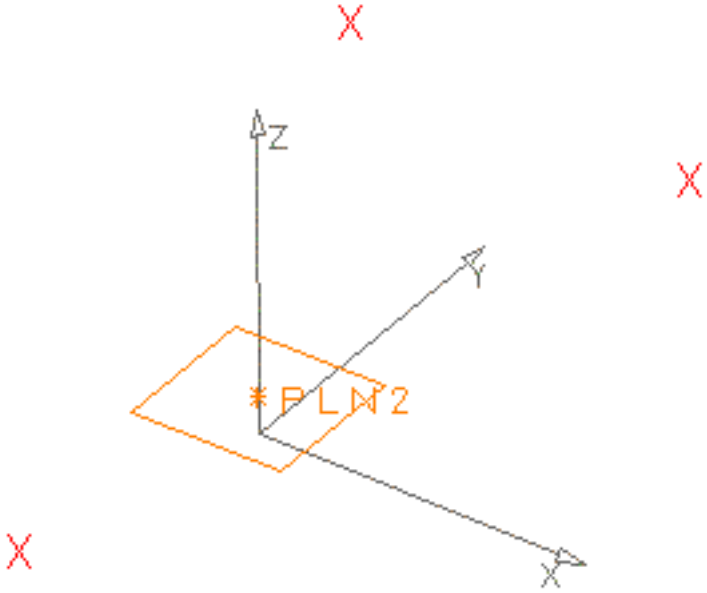
Zunächst wird MESSD auf dem Steuerungsrechner gestartet, wobei kein Anfahren des Maschinennullpunktes erforderlich ist. Der zu messende Würfel wird am besten exakt parallel zum Koordinatensystem der Messmaschine ausgerichtet. Ist dies nicht möglich, kann man auch ein neues Koordinatensystem in CATIA konstruieren und in diesem messen, die ist hier jedoch nicht beschrieben. Ebenso sollte der Würfel so hoch liegen, daß auch die Auflagefläche vermessen werden kann.

Hat man dann auch noch CATIA und QUICKMESS gestartet, kann man beginnen, indem man die Auflagefläche einmisst. Diese ist bekanntlich eben und wird deshalb mit der Funktion PLANE/TROUGH konstruiert. Das Vorgehen ist einfach: in die Funktion PLANE/ TROUGH gehen, dann die permanente Funktion IMESS starten, eine 3 als gewünschte Punktezahl eingeben und QUICKMESS in den Vordergrund holen. Jetzt mit QUICKMESS die Maschine so lange verfahren, bis die drei Punkte gemessen sind. Geht etwas schief, z.B. hat man aus Versehen einen falschen Punkt gemessen, zunächst die anderen Punkte messen, dann mit der Funktion ERASE den falschen löschen und eine weitere Messung für einen einzelnen Punkt wiederholen. Diese Punkte sollten aus Gründen der Messgenauigkeit möglichst weit auseinander liegen. Das Vorgehen kann dabei im CATIA Fenster beobachten werden, auch wenn es im Hintergrund steht. Gegen Ende dieser ersten Messung sieht das CATIA Bild

dann etwa wie folgt aus:

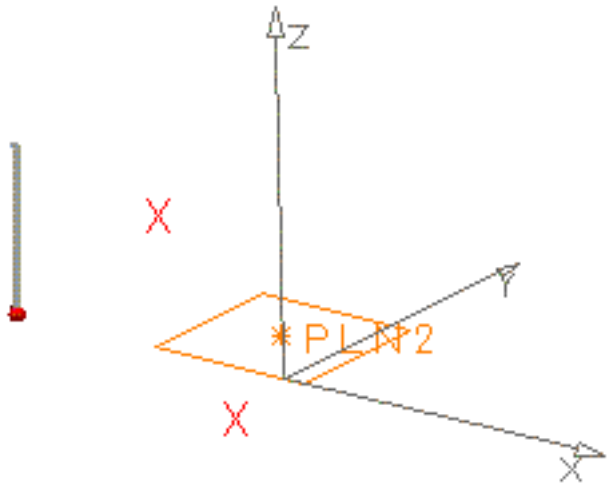


Sowie die drei Messpunkte vorhanden sind, beendet sich IMESS selbstständig und die Punkte können zur Konstruktion der Auflageebene des Würfels angewählt werden. Da die Unterseite des Würfels eben ist, ist die Auflageebene gleichzeitig die Ebene der Unterseite des Würfels:

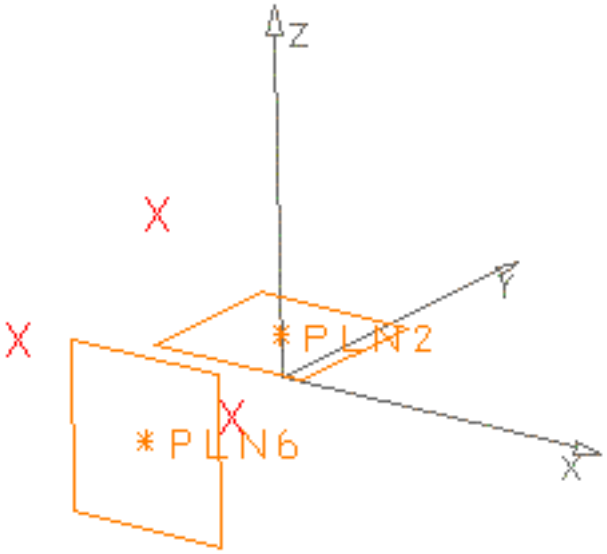


Nun kann man damit beginnen, die Seitenflächen des Würfels zu bestimmen. Auch hier kann man sich wieder die Kenntnis, daß ein Würfel von ebenen Flächen begrenzt wird, zunutze

machen und die erste dieser Ebenen durch lediglich drei Messpunkte erfassen:

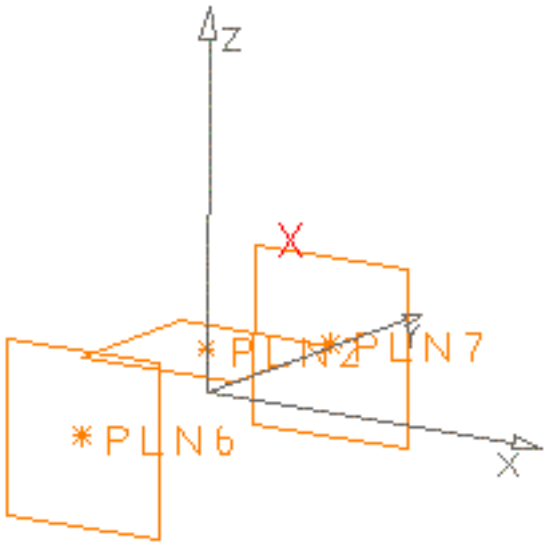


Um im Laufe der weiteren Konstruktion nicht die Übersicht zu verlieren, ist es empfehlenswert, die nicht mehr benötigten Punkte in den No Show Bereich zu stellen oder gar zu löschen:

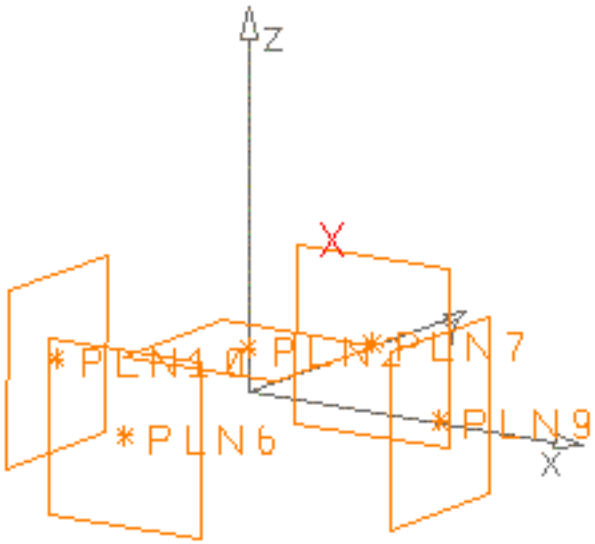


Die anderen drei seitlichen Flächen lassen sich sogar durch nur einen einzigen Punkt festle-

gen, da die Flächen eines Würfels ja bekanntlich senkrecht aufeinander stehen:

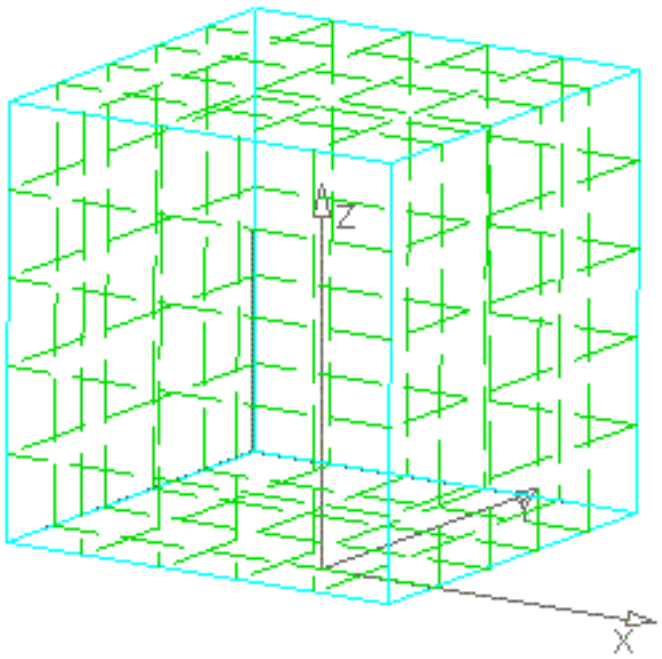


Fügt man nun noch einen Messpunkt für die obere Seite hinzu, sind die Begrenzungsebenen des Würfels komplett:



Nun kann der Würfel komplettiert werden, in dem man die Kanten durch schneiden jeweils zweier Ebenen konstruiert. Dann kann man diese Kanten kürzen und Faces dazwischen

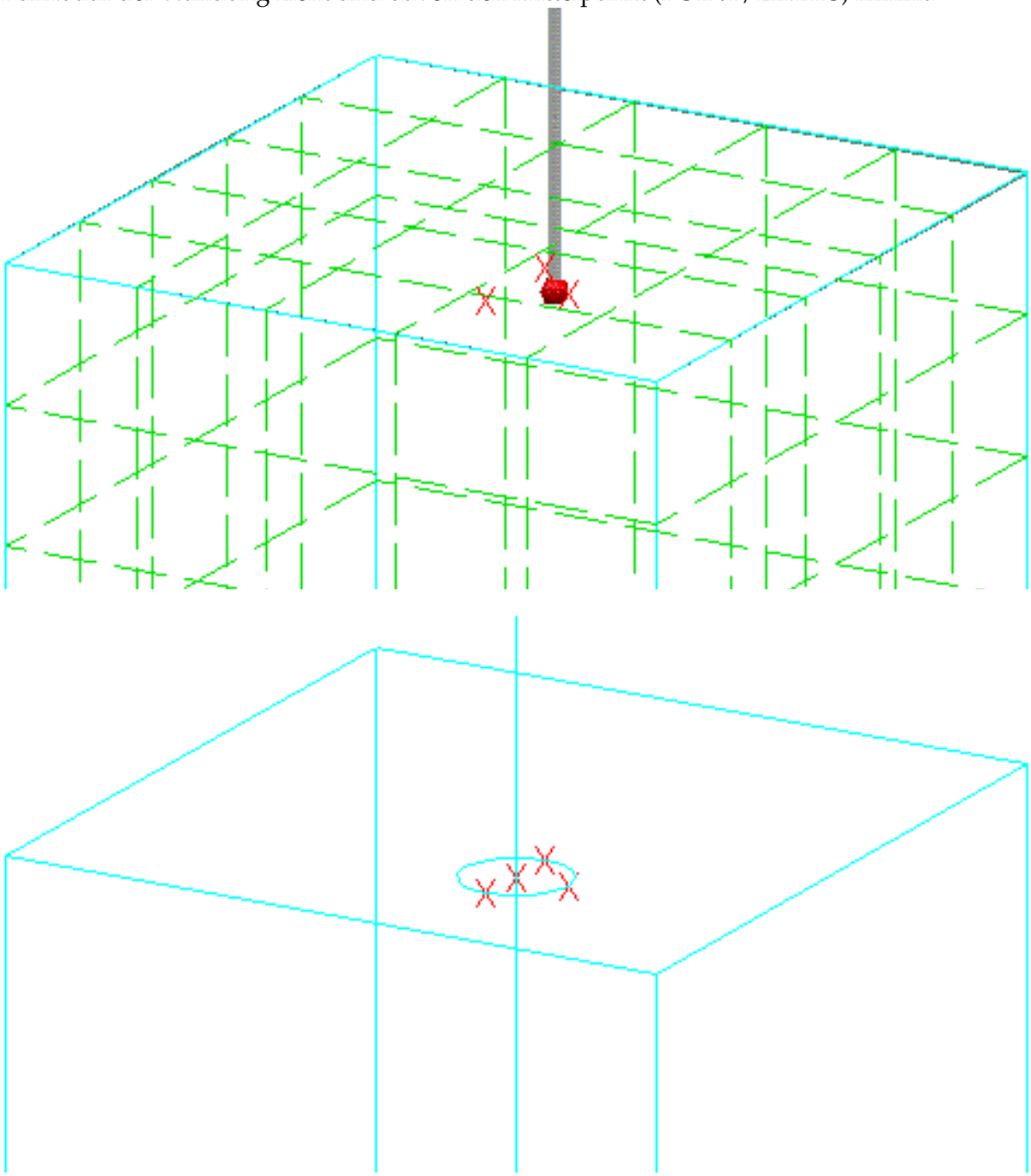
legen, der Würfel ist fertig:



Bestehen zu irgendeinem Zeitpunkt Zweifel an den implizierten Annahmen, z.B. daß eine der Ebenen auch wirklich eben ist, kann man jederzeit zusätzliche Messpunkte aufnehmen und mit der Sollposition vergleichen. Eine nicht ebene Fläche macht sich z.B. bemerkbar, wenn ein vierter Messpunkt einen nennenswerten Abstand zu der Ebene aufweist, die mit den ersten drei Punkten gebildet wurde.

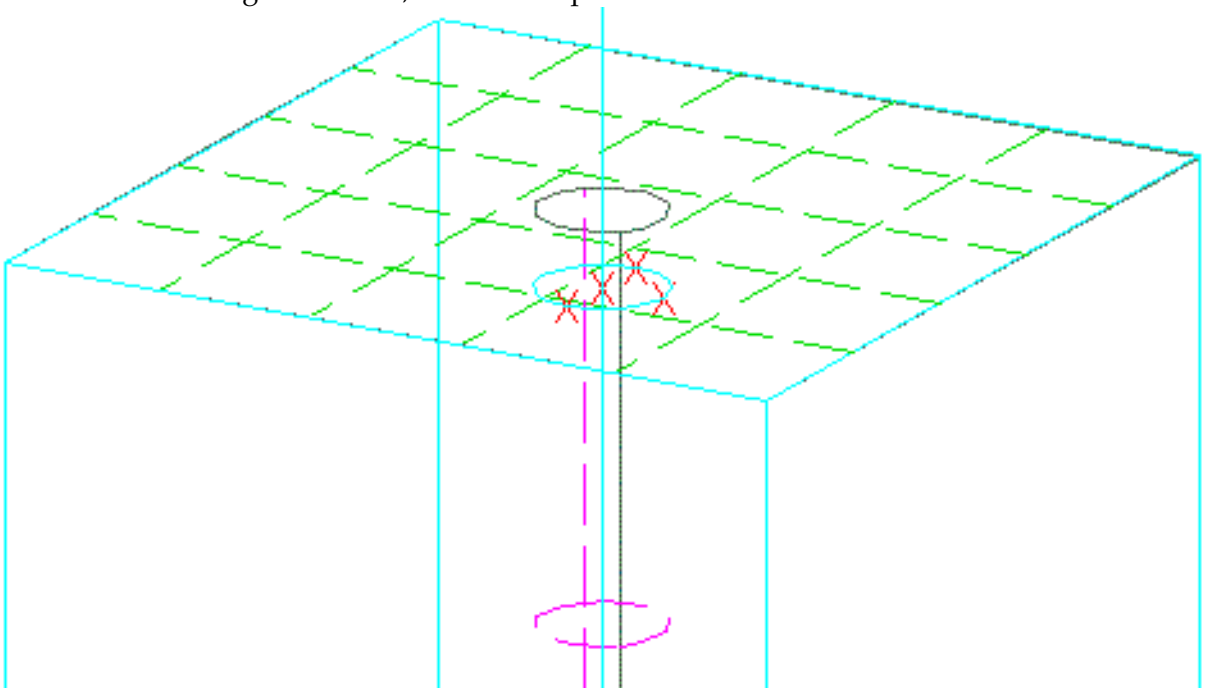
Nun fehlen noch Verrundung und Bohrung an dem Würfel. Von der Bohrung ist bekannt, daß deren Mittellinie parallel zu den vertikalen Kanten des Würfels liegt. Also muß ein Punkt auf der Mittellinie und ein Punkt an der Bohrungswandung bestimmt werden, dann ist die Geometrie bestimmt. Den Punkt auf der Mittellinie erhält man, indem man einen Kreis durch drei

Punkt auf der Wandung zieht und davon den Mittelpunkt (POINT/LIMITS) nimmt:

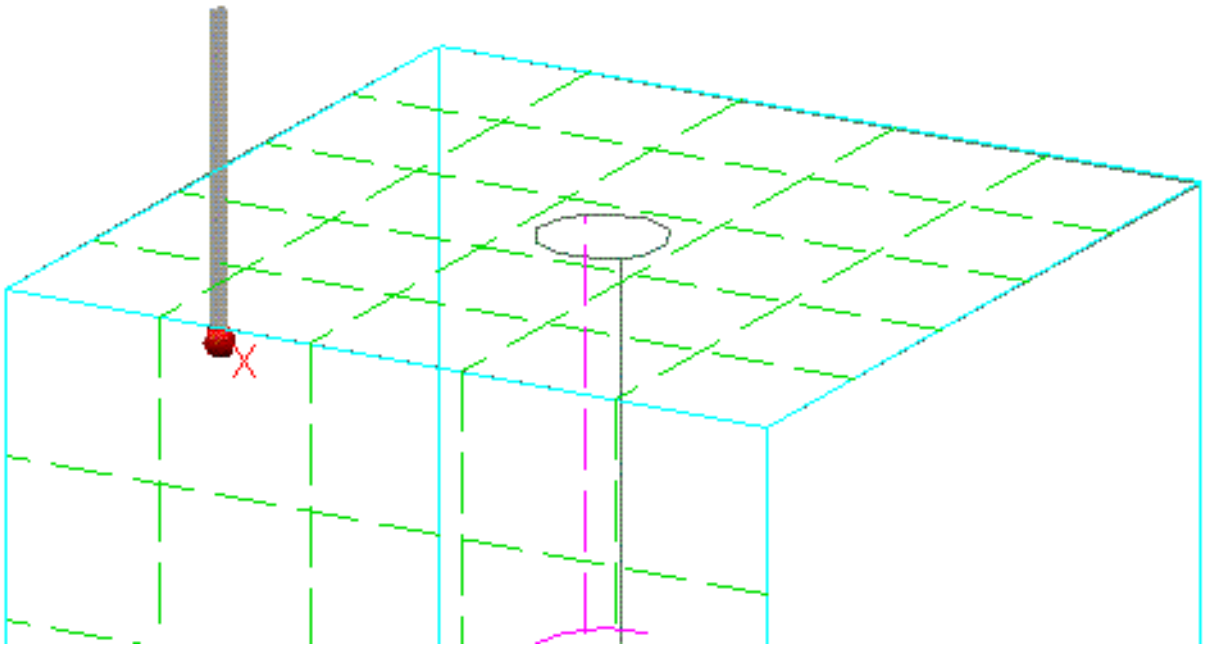


Nun ist es kein Problem mehr, den entsprechenden Zylinder zu konstruieren. Ein „ziehen“ des Kreises entlang einer der Würfelkanten ist nicht empfehlenswert, da nicht sicher ist, daß die Ebene, auf der der Kreis liegt, senkrecht zu den Körperkanten ist. Wäre dies nicht so,

würde die Bohrung nicht rund, sondern elliptisch. Hier die bessere Konstruktion:



Um den Radius für die Verrundung des Würfels (zwischen Vorderseite und Oberseite) zu bestimmen. ist lediglich ein einzelner Messpunkt erforderlich:



Dieser Messpunkt sollte in Schrägfahrt aufgenommen werden, falls ein Kugeltaster verwendet wird, um den Messfehler aufgrund der Radiuskorrektur gering zu halten.

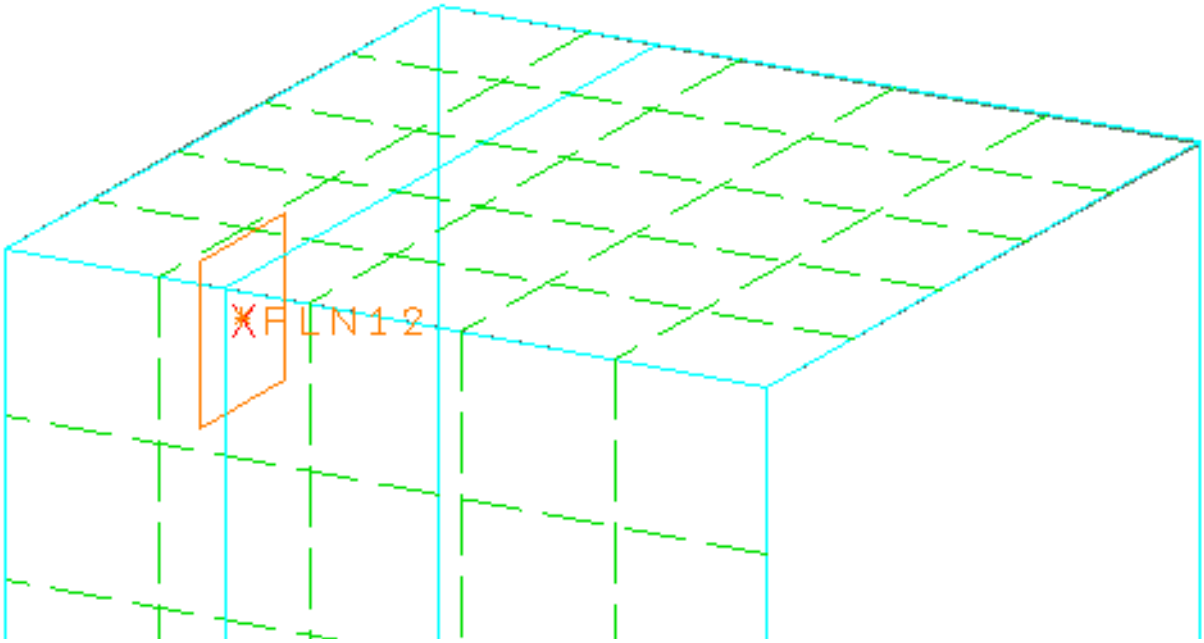
Bemerkung:

> Der Fehler bei der Radiuskorrektur ist vom Durchmesser der Messtasterkugel und von der Abweichung der Messverfahrrichtung von der Flächennormalen abhängig. Schlimmstenfalls, wenn die Verfahrrichtung annähernd parallel zur Werkstückoberfläche ist, ist dieser Fehler 50% des Kugeldurchmessers. Bei einer Abweichung der Verfahrrichtung von 45°, was unter allen Umständen ohne Schrägfahrt eingehalten werden kann, liegt der Fehler bei ca. 29% des Kugeldurchmessers, bei einer Abweichung von 10°, die mit bloßem Auge noch zu erfassen ist, ist dieser Fehler nur noch 1,5% des Kugeldurchmessers. Die Formel zur Berechnung lautet:

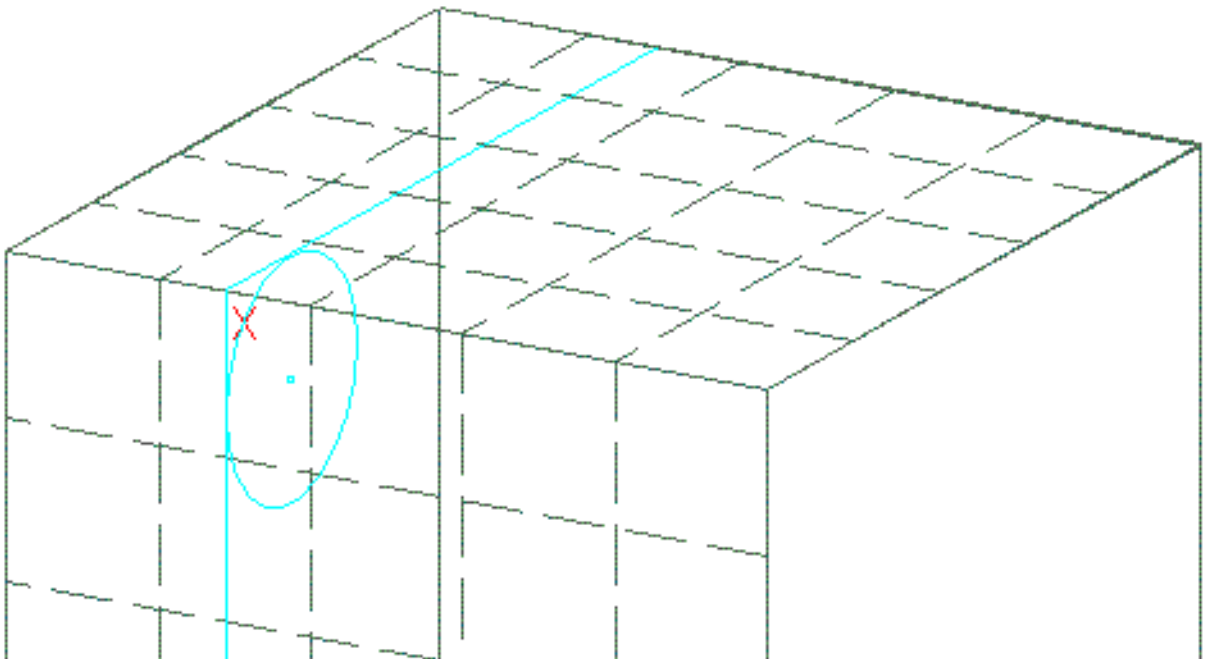
`0,5 * (1 - cos (Winkelabweichung)) * Messkugeldurchmesser`.

Daraus würde eigentlich folgen, daß man sehr kleine oder gar spitze Taster verwenden sollte, diese bringen jedoch andere Probleme, besonders auf rauhen und weichen Oberflächen.

Der gewünschte Radius liegt jetzt tangential zu den begrenzenden Flächen und durch den gemessenen Punkt sowie auf einer Ebene senkrecht zu der entsprechenden Körperkante. Also legt man zunächst eine Plane senkrecht zu dieser Kante durch diesen Punkt und schneidet diese dann mit den Begrenzungsflächen:

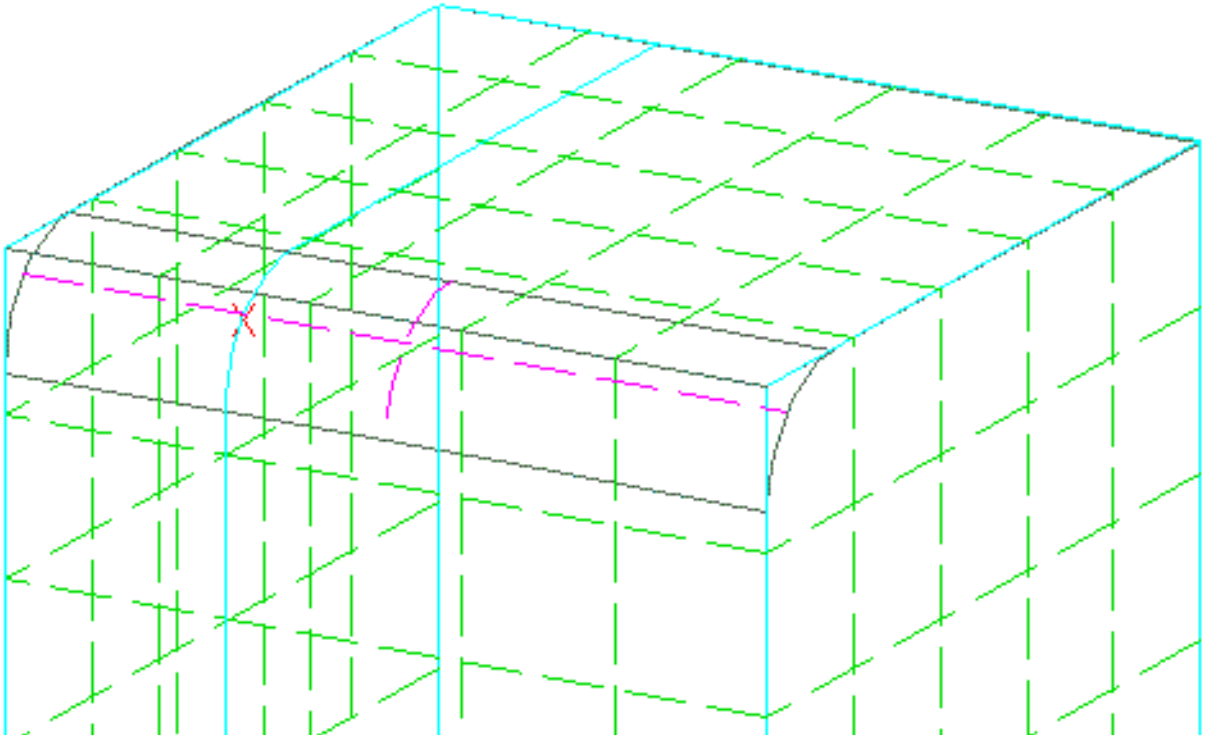


Im 2D–Modus kann man nun einen Kreis tangential zu diesen Schnittlinien und durch den Messpunkt legen (CURVE2/CIRCLE/MULTITGT):

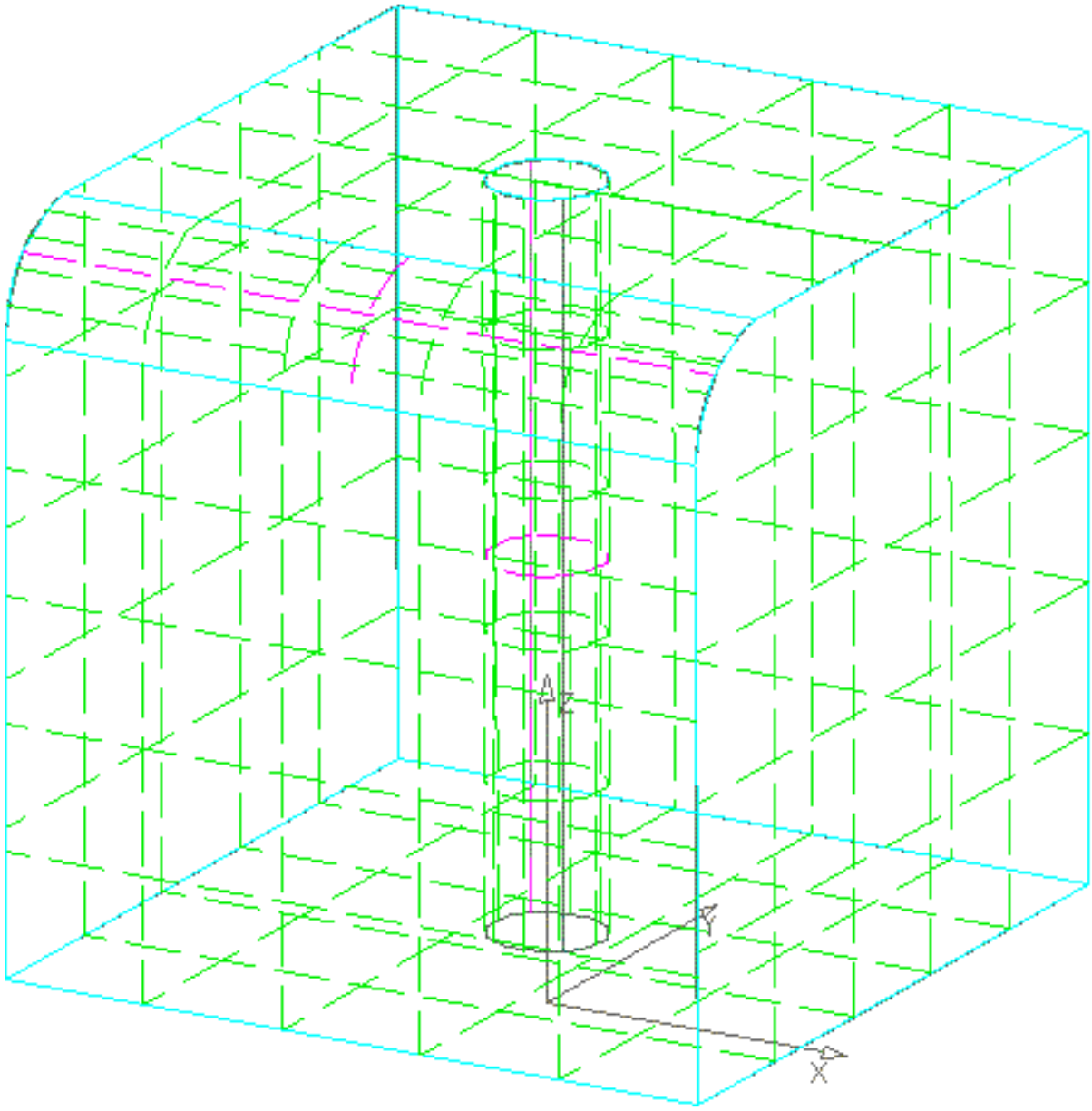


Nach beschneiden des Kreises zu einem Radius kann dieser Radius an der Körperkante ent-

langgezogen werden, die Verrundung ist fertig:



Geschafft! Nach kürzen der überstehenden Flächen sieht das Endergebnis wie folgt aus:



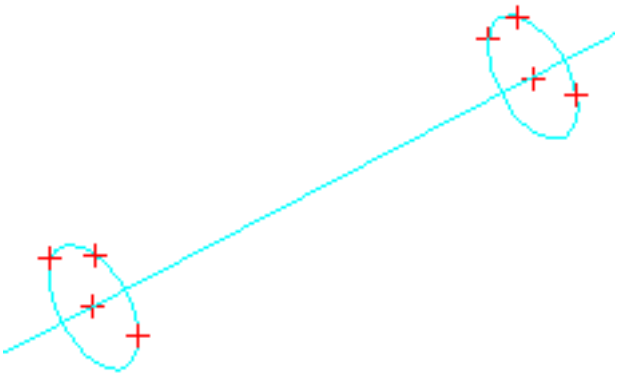
Wie gesagt, war dies ein besonders einfaches Beispiel, es zeigt jedoch auf, daß die Konstruktion mit Hilfe von Messpunkten sich kaum von der Vorgehensweise unterscheidet, die man anwendet, wenn man mit bekannten Maßen arbeiten würde.

5.2 Ein weniger triviales Beispiel

Hier ist noch das Ergebnis einer Messung, die im Rahmen eines CAD Projektes von zwei anderen Studenten mit den gerade fertiggestellten Programmen durchgeführt wurde. Es handelt sich dabei um das Verdichterschaufelrad eines LKW–Turboladers.

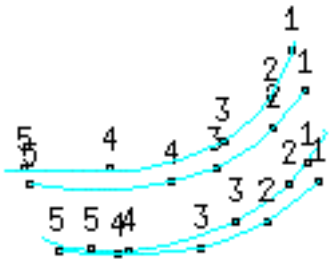
Für die zu messenden Punkte kamen natürlich vor Allem die Schaufelflächen in betracht. Aber auch die Äußere Kontur der Schaufelenden wurde mit der Messmaschine bestimmt.

Zunächst wurde die Rotationsachse des Schaufel-



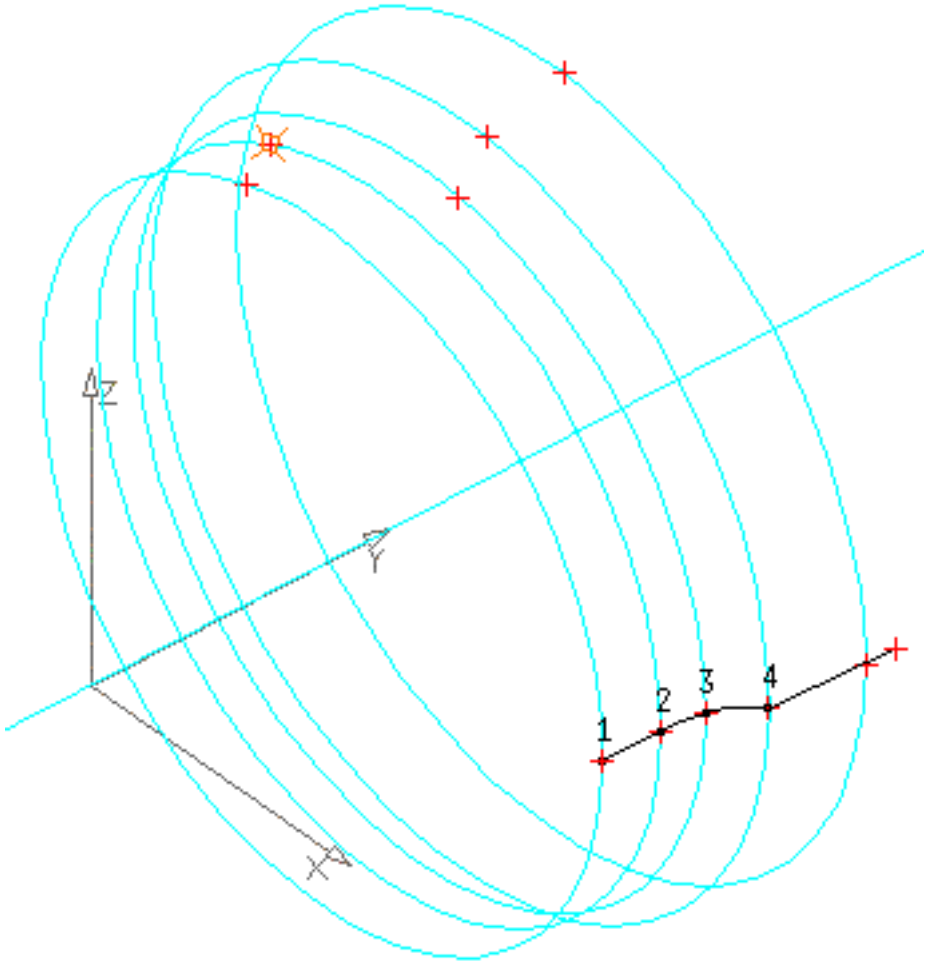
rades bestimmt, indem an zwei möglichst weit auseinanderliegenden Stellen Umfangskreise der Laderwelle gemessen wurden. Damit war das Koordinatensystem in ausreichendem Maße bestimmt.

Als nächstes wurden in der Nähe des äußeren und am inneren Endes der Schaufeln auf jeder Seite ein Reihe von 5 Punkten aufgenommen. Diese Messung war aufgrund der beengten Verhältnisse zwischen den Schaufeln nicht einfach, konnte jedoch sofort geprüft und korrigiert werden. Auf dem Bild sind die Punkte bereits zu Constraints zusammengefasst und mit einem Spline verbunden. Jeweils ein Punkt am vorderen und hinteren Ende der Schaufeln komplettierten die Messung in diesem Bereich.



Auch an der Außenkontur der Schaufeln wurden nur 5 Punkte aufgenommen. Da diese jedoch nicht auf einer Ebene lagen, die durch die Rotationsachse führt, wurde ein Kreis um diese Rotationsachse durch die Messpunkte gezogen und diese Kreise dann mit dieser Ebene geschnitten. Diese Schnittpunkte konnten dann mit einem Spline verbunden und zur kon-

struktion der Außenkontur verwendet werden:



Die gesamten Messungen für dieses Bauteil dauerten ca. 90 Minuten inclusive der Korrektur nicht zufriedenstellender Messungen. Bevor die Messmaschine abgeschaltet wurde, konnte die Konstruktion der resultierenden Flächen so weit fortgeführt werden, daß sicher war, daß keine weiteren Messungen notwendig waren. Erst dann wurde das Bauteil ausgespannt und einige weitere Konturen, wie z.B. der Sechskant, mit einem Messschieber gemessen. Wäre es dennoch notwendig geworden, erneut zur Messmaschine zu gehen, wäre eine erneute Bestimmung des Koordinatensystems (wie bei allen anderen Verfahren, die eine Messmaschine verwenden) erforderlich geworden.

Kapitel 6

Hinweise für Programmierer

Hier folgen nun eine Reihe Hinweise und Beschreibungen, die erst dann interessant werden, wenn man Veränderungen an den Programmen vornehmen will.

6.1 Übersicht

Als Programmiersprache wurde für alle Programme und Module die Sprache C mit einigen Erweiterungen von C++ verwendet. Die C++ Erweiterungen dienen nur einer besseren Strukturierung und sollten kein Problem darstellen, wenn man nur mit C vertraut ist. Die Verwendung eines C++ Kompilers ist jedoch erforderlich, bei der Entwicklung wurde der GNU Kompiler verwandt. Die Dateien mit Code haben die Endung `.cc`, solche mit Headern die Endung `.h`. Zu jeder Code–Datei außer denen, die das Hauptprogramm enthalten, existiert eine Header–Datei.

Jedes der Programme ist in einem eigenen Verzeichnis untergebracht. Dateien, die von allen Programmen gemeinsam benutzt werden, befinden sich im Verzeichnis `libimess`. In diesem Verzeichnis ist jedoch kein Makefile, da die Programme häufig für verschiedene Plattformen benötigt werden, die untereinander nicht binärkompatibel sind. Die zugehörigen Objekte befinden sich im Verzeichnis des Programmes, das die Routinen nutzt.

Die Programme bzw. das CATIA Modul werden durch Ausführen des `make` Befehls ohne weitere Parameter in dem jeweiligen Verzeichnis gebaut. Wurde die Verzeichnisstruktur geändert, ist es u.U. notwendig, das Makefile entsprechend anzupassen. Betriebssystemspezifische Einstellungen werden jeweils in einem speziellen Makefile getroffen, daher muß die Systemvariable `UNAME` den Namen des Betriebssystems enthalten. Auf RS6000 z.B. ist `UNAME` auf `AIX` gesetzt und verwendet entsprechend `Makefile.AIX`.

Der Code für das Steuerungsprogramm MESSD befindet sich im Verzeichnis `messd` und enthält keine weiteren Besonderheiten. Vor Inbetriebnahme des Programms ist natürlich zu beachten, was bei der Installation des Programmes beschrieben wurde.

Beim Programm QUICKMESS im Verzeichnis quickmess ist zu beachten, daß es XFORMS verwendet und im Makefile entsprechend Angaben enthalten sein müssen, daß der Kompiler die Dateien `forms.h` und `libforms.a` findet.

Das CATIA IUA Programm wird von CATIA Interpretiert und muß daher nicht kompiliert werden. Der Load ist etwas ungewöhnlich zu kompilieren, dies sollte jedoch auch noch nach einer Ortsveränderung des Quellcodes funktionieren. Einzelheiten dazu sind in der CATIA Dokumentation und dem dortigen Makefile enthalten. Der Quellcode für den Load ist im gleichen Verzeichnis wie der Load selbst. Position von Load und IUA Programm sind von der CATIA Installation abhängig, wie bei der Installation dieser Teile beschrieben.

Prinzipiell habe ich versucht, den Quellcode so reichhaltig zu kommentieren, daß man bereits durch lesen der Kommentare über die Funktionsweise der Programme informiert wird. Natürlich sind alle Ausnahmen vom Standard – soweit sie überhaupt erforderlich waren – besonders ausführlich beschrieben.

6.2 Das Zusammenspiel der Programme

Die Programme kommunizieren untereinander über BSD–Sockets, dies ist ein Verfahren, das in einem TCP/IP Netz eine Verbindung ähnlich einer seriellen Schnittstelle herstellt. Die Programmierung dieses Verfahrens ist in einschlägigen Büchern über UNIX–Betriebssysteme, aber auch in der Online–Hilfe von AIX beschrieben. Großer Vorteil dabei ist, daß die beiden kommunizierenden Programme beliebige Entfernungen überwinden, aber auch auf dem gleichen Rechner laufen können. Programme wie TELNET, FTP und X–Windows verwenden auch Sockets. Die Verwendung des Ports 1066 ist fest einkompiliert, dieser Port wurden in dem von mit verwandten Lehrbuch benutzt und führte bisher nicht zu Konflikten.

Bei der Kommunikation der Programme ist das Steuerungsprogramm MESSD das zentrale Serverprogramm, alle Nachrichten gehen von oder zu diesem Programm. Dies ist auch der Grund, warum MESSD immer als erstes Programm laufen muß. Kommt eine Nachricht von einem Programm, verteilt MESSD diese Nachricht an alle verbundenen Programme, wenn es der Typ der Nachricht erfordert. Natürlich verteilt MESSD auch alle selbstproduzierten Nachrichten, wie z.B. die aktuelle Maschinenposition an alle verbundenen Programme. Es bleibt dem empfangenden Programm überlassen, ob es die Nachricht auswertet oder nicht.

Die gesamte Kommunikation erfolgt mit Nachrichten im Klartext, d.h. es werden nur die Zeichen A–Z, a–z, 0–9 und der Dezimalpunkt verwendet. Dies hat dem Vorteil, daß Betriebssystemunterschiede wie z.B. die byte–order keine Probleme verursachen und auch eine Verbindung mit z.B. TELNET hergestellt werden kann. Jede Nachricht wird durch ein Zeilenende (`\n`) abgeschlossen. Die zulässigen Nachrichten sind in der Datei `libimess/improtocol.h` definiert.

6.3 Die Funktionen des gemeinsamen Codes

Neben der oben beschriebenen Protokolldefinition in der Datei `improtocol.h` befindet sich noch Code für die Fehlerbehandlung und eine Library für die Behandlung von Vektoren im Verzeichnis `libimess`.

In `error.cc` werden die Funktionen `WarningMsg()` und `ErrorMsg()` angeboten. Beide vereinfachen das Abfangen von Fehlern, da sie mit einem einzelnen Funktionsaufruf eine ordentlich formatierte Fehlermeldung ausgeben. Auch die Fehlermeldung des Betriebssystems wird ausgegeben, falls verfügbar. Im Gegensatz zu `WarningMsg()` wird bei `ErrorMsg()` die Programmausführung über die Funktion `exit()` abgebrochen. Daher ist es notwendig, evtl. notwendigen Code zum Aufräumen mit der Funktion `atexit()` zu installieren oder im Destruktor vorzusehen.

In der Datei `lib3d.h` wird die Klasse `Double3d` definiert. Es existiert keine extra Code–Datei, da die (kurzen) Definitionen der Funktionen bereits in der Deklaration enthalten sind. Deren Daten bestehen lediglich aus drei `double` Variablen, eben um einen Raumvektor darstellen zu können. Weiterhin werden noch Funktionen und Operatoren wie `Abs()`, `Len()` etc. angeboten, um ein möglichst normales arbeiten mit Vektoren zu ermöglichen. So ist es z.B. möglich, 2 Vektoren durch `c = a + b;` zu addieren.

6.4 Die Kommunikation von MESSD mit der Hardware

MESSD kommuniziert direkt mit der Hardware, ohne einen Treiber oder ein TSR–Programm dazwischenzuschalten. Dies ist einfacher als vorher einen Treiber zu schreiben und sinnvoll, da Standardprogramme niemals mit dieser Hardware arbeiten werden.

6.4.1 Was von der mitgelieferten Software abgeleitet wurde

Der direkte Hardwarezugriff erfolgt in der Datei `tpu.c`, eine Abwandlung der bei der Hardware mitgelieferten Datei `tpulink.c`. Es sind nur Routinen zum aufnehmen der Verbindung, sowie zum lesen und schreiben enthalten, alles andere ist in anderen Dateien enthalten. Der Hardwarezugriff wurde für LINUX angepasst und nicht benötigter Code weggelassen. Diese Datei muß unbedingt mit dem gleichen Kompiler in der gleichen Optimierungsstufe wie der Kernel kompiliert werden, sonst kann der Code nicht gelinkt werden. Näheres zur Hardwareprogrammierung unter LINUX siehe im Linux Kernel Hackers Guide.

Die Datei `mcu.h` stammt von der Datei `mcusrvr.h` ab und enthält Strukturen und Definitionen, die bei der Kommunikation mit dem MCU 6 Betriebssystem benötigt werden. Zahlreiche nicht benötigte Strukturen wurden entfernt. Die Funktionen aus `mcusrvr.c` sind nach `machine.cc` gewandert.

6.4.2 Die Maschinensteuerung in `machine.cc`

Die Datei `machine.cc` enthält die Klasse `Machine` und damit alle Funktionen, die mit der Maschine ausgeführt werden. Die Routinen, die direkt MCU 6 Befehle ausführen wurden von `mcusrvr.h` abgeschrieben, jedoch so modifiziert, daß sie in jedem Fall alle drei Achsen ansprechen. Auch wurden ausführlichere Namen verwandt. Konstruktor und Destruktor sind so ausgeführt, daß auch bei einem unerwarteten Programmabbruch die Verbindung ordnungsgemäß abgebrochen wird.

Die Funktion `Init()` Initialisiert due MCU 6, d.h. sorgt für ein brauchbares Betriebssystem, lädt dies erforderlichenfalls aus der Datei `rwtos.btl` nach und Initialisiert die Hardware mit den Daten aus `system.dat`.

`SearchReference()` fährt den Maschinennullpunkt an. Dies ist nicht unbedingt erforderlich, um mit der MCU 6 zu arbeiten.

`StartMove()` startet die Maschine in die bestimmte Richtung, `StopMove()` hält sie wieder an. Bevor die Maschine wirklich losfährt, prüft `StartMove()`, ob dies überhaupt möglich ist oder der Taster angefahren ist bzw. die Bewegung weiter ins Limit führen würde. Wegfahren aus dem Limit wird erlaubt. Alle Bewegungen der Maschine, außer dem Anfahren der Nullpunktes, werden mit den Jog... Befehlen vorgenommen. Ist die Maschine einmal gestartet, hält sie erst wieder an, wenn `StopMove()` ausgeführt wird. Es obliegt also dem ausführenden Programm, dies zu tun.

`CheckMove()` prüft alle diese Bedingungen, die einen Abbruch der Bewegung veranlassen, also erreichen eines Hardware Limits oder das anfahren des Tasters. Ist ein Limit erreicht, wird die Maschine einfach nur angehalten, ist jedoch der Taster angefahren, wird dessen Rückzug veranlasst und der Messpunkt als Parameter zurückgegeben. Im letzteren Fall ist der Rückgabewert `true`. Diese Routine eignet sich auch, um Messpunkte beim verfahren von Hand zu erfassen. In diesem Fall wird kein Rückzug eingeleitet. Da die Maschinenposition im Augenblick des Tasteransprechens von der MCU 6 aufgenommen und bis zum nächsten Ansprechen gespeichert wird, bringt ein *sehr* häufiges aufrufen dieser Funktion keine genaueren Messergebnisse, 5 – 10 Aufrufe je Sekunde sind völlig ausreichend.

`GetCurrentPos()` und `GetLatchedPos()` liefern die aktuelle Position der Maschine bzw. deren Position beim letzten auslösen des Messtasters.

Die Flags `debug`, `transdebug` und `reboot` enthalten die Einstellungen, die beim Aufruf des Programms mitgegeben werden.

Alle anderen Funktionen in machine.cc können nur von innerhalb der Klasse `Machine` verwendet werden und werden deshalb hier nicht besonders beschrieben.

6.5 Die Funktionsweise von MESSD

Die Hauptschleife von MESSD befindet sich in der Datei `messd.cc`. Zu Beginn des Programms werden die Parameter ausgewertet, die dem Programm beim starten mitgegeben

wurden. Dies beendet das Programm nach der Ausgabe von etwas Text oder setzt einige Flags. Danach wird ein Signal Handler gesetzt und die Maschine initialisiert. Dann geht das Programm in eine Endlosschleife, die nur noch durch ein Signal (z.B. `Ctrl-C`) abgebrochen werden kann.

In der Hauptschleife wird zunächst die Socketverbindung abgefragt, ob Nachrichten vorliegen. Ist dies nicht der Fall, wartet dieses Unterprogramm etwas, um die Belastung der Rechners nicht unnötig steigen zu lassen. Liegt jedoch eine Nachricht vor, wird diese ausgewertet. In der Regel ist das mit dem Aufruf der entsprechenden Routine der Maschine getan. Danach wird `CheckMove()` der Maschine aufgerufen um auf Messpunkte etc. zu prüfen. Zum Schluß wird noch dafür gesorgt, daß in regelmäßigen Abständen die aktuelle Position der Maschine bekannt gemacht wird und das Spiel beginnt von Neuem.

In der Datei `connection.cc` befindet sich die Klasse `Connection`, die sich um alles kümmert, was mit der Socketverbindung zu den anderen Programmen zu tun hat. Konstruktor und Destruktor kümmern sich um Öffnen und schließen des Socketports.

Wichtigste Funktion der Klasse ist `WaitGetMessage()`, denn hier wird der Port auf eingehende Nachrichten abgehorcht. Zunächst wird geprüft, ob ein Programm versucht, eine Verbindung aufzunehmen. Sie wird in jedem Fall akzeptiert, der neue Partner bekommt sofort die aktuelle Maschinenposition zugesandt. Danach wird eine der anliegenden Nachrichten gelesen und an das aufrufende Programm zurückgegeben. Die restlichen Nachrichten bleiben im Puffer des Betriebssystems bis zum nächsten Aufruf der Routine. Geschlossene Verbindungen werden hier von der Liste der Verbindungen gestrichen.

Zweiter Teil der Klasse `Connection` ist die Funktion `Broadcast()`, die eine Nachricht an alle verbundenen Programme sendet. Da das Protokoll keine Unterschiede in der Art der angeschlossenen Programme macht, ist die einzige Möglichkeit, Nachrichten zu versenden.

6.6 Die Funktionsweise von QUICKMESS

Das Hauptprogramm von QUICKMESS befindet sich in der Datei `quickmess.cc`. Hier werden zunächst ähnlich MESSD die Startparameter geprüft. Dann wird mit wenigen Zeilen das Grafikfenster aufgebaut un die Verbindung zum Server, in der Regel MESSD, aufgenommen. Die Hauptschleife tun dann nichts weiter als abwechselnd die Kontrolle an das Grafikfenster und and die Socketverbindung abzugeben. Abgebrochen wird das Programm, wenn die Verbindung zum Server verloren geht, der User den Quit Button im Grafikfenster drückt oder das Programm mit einem Signal abgebrochen wird.

Die gesamte Socketverbindung zum Server wird in der Klasse in `MessmaConn` in der Datei `messmaconn.cc` behandelt. Hier passiert nichts außergewöhnliches, Konstruktor und Destruktor machen nichts. Die Funktion `GetServer()` stellt die Verbindung zum Server her oder bricht das Programm ab, falls dies nicht gelingt. Die Funktion `Send()` dient zum senden von Nachrichten. In der Funktion `IdleProc()` wird geprüft, ob eine eingehende Nachricht vorliegt bzw. das Programm abgebrochen, falls ein Verlust der Verbindung ausgemacht wird. Die Nachrichten werden hier auch gleich behandelt, ein eingehende Messpunkt oder eine eingehende aktuelle Position wird in das Display geschrieben, andere Nachrichten werden ignoriert. Die restlichen Funktionen beschränken sich weitgehend darauf, die Wünsche des Benutzers in entsprechende Nachrichten für den Server umzusetzen und abzuschicken.

Die grafische Benutzeroberfläche wird mit dem FORM DESIGNER erstellt, ein Programm, mit dem man die Oberfläche quasi malen kann und den einzelnen Buttons sog. Callbacks zuordnet. Jede Aktion in der grafischen Oberfläche ruft in dem fertigen Programm dann den zugeordneten Callback auf, um den Rest, z.B. Aufbau der Grafik braucht sich der Programmierer nur noch in Sonderfällen zu kümmern. Die Beschreibung der Oberfläche ist in der Datei `myforms.fd`, die vom FORM DESIGNER gelesen werden kann. Der FORM DESIGNER wirft dann eine Datei mit Quellcode, `myforms.c`, aus, die zusammen mit dem restlichen Programm kompiliert werden muß. Da QUICKMESS mit einem C++ Kompiler kompiliert wird, wird diese Datei jedoch vor der Kompilation nach `myforms.cc` umbenannt.

Die benötigten Callbacks befinden sich in der Datei `callbacks.cc`. Die meisten Callbacks rufen lediglich die entsprechende Funktion der Messmaschinenverbindung auf. Die Callbacks, die eine Bewegung auslösen, berücksichtigen außerdem noch, welche der Maustasten den Button gedrückt hat, um die Geschwindigkeit entsprechend einzustellen. Die Buttons für die Bewegung entlang der Achsen und die Bewegung von/zur markierten Position benutzen jeweils einen gemeinsamen Callback.

6.7 Die Funktionsweise des CATIA Moduls

Das CATIA Modul besteht im wesentlichen aus zwei Teilen: Einem IUA Programm, das von CATIA zur Laufzeit interpretiert wird und einem Load, dessen Funktionen von diesem IUA Programm aufgerufen werden.

Der Load befindet sich in der Datei `imess.c` und enthält einige Hinweise, wo in der CATIA Dokumentation die verwandten Routinen und Variablen beschrieben sind. Er wird an drei Stellen aufgerufen: `im_open()`, `im_close()` und `im_do()`.

`im_open()` initialisiert zunächst die Datenstruktur, die alle Unterprogramme gemeinsam nutzen. Hier wird auch eine Magic Number gesetzt, eine frei gewählte Zahl, die sicherstellen soll, daß die Daten zwischen zwei Aufrufen von Unterprogrammen nicht verloren gingen. Jede Routine prüft zunächst diese Magic Number und geht davon aus, daß die Daten in Ordnung sind, wenn sie unverändert blieb. Weiterhin nimmt `im_open()` die Verbindung zum Server auf und sucht das Ditto für die Visualisierung des Messwerkzeuges heraus.

In `im_close()` wird das Ditto wieder entfernt und die Verbindung zum Server geschlossen.

`im_do()` befragt die Serververbindung nach eingehenden Nachrichten ähnlich wie QUICKMESS. Auch hier werden nur die Nachrichten mit einer neuen Maschinenposition und gemessenen Punkten bearbeitet. Eine solche Nachricht wird an das aufrufende Programm zurückgegeben, dabei ist die Art der Nachricht im Rückgabewert und die Koordinaten in den Parametern enthalten. Liegt keine Nachricht vor, wird die Position der Messtastervisualisierung aufgefrischt. Bei sehr vielen eingehenden Nachrichten wäre es denkbar, daß keine Zeit mehr verbleibt, dieses Ditto zu verschieben, dies konnte jedoch beim austesten der Programme nicht beobachtet werden.

Die Funktion `VisuMove()` bewerkstelligt dieses auffrischen der Visualisierung bzw. erzeugt das erste Ditto, falls es noch nicht existiert. Das erzeugen einer Transformation und anwenden dieser Transformation auf das Ditto hat sich bei Versuchen als schnellste Methode herausgestellt. Es wurden auch andere Funktionen der CATIA Library ausprobiert, um die Transformation zu erstellen sowie die Möglichkeit das alte Ditto durch ein neues zu ersetzen.

Das IUA Programm ist in der Datei `IMESS.iuaproc` enthalten. Es setzt zunächst die Funktion `im_close()` als die Funktion, die im Falle eines Programmabbruches oder der normalen Beendigung des Programms aufgerufen werden soll. Dann wird der User nach der Zahl der zu erwartenden Punkte gefragt. Liegt diese Zahl fest, wird die Verbindung zur Messmaschine hergestellt und dann so lange immer wieder `im_do()` aufgerufen, bis die gewünschte Anzahl Messpunkte eingegangen ist. Der Rückgabewert von `im_do()` gibt Auskunft darüber, was erreicht wurde, die Aktionen des IUA Programms selbst sind jedoch gering: möglicht häufiges auffrischen des Displays, darstellen der aktuellen Koordinaten in Textform und mitzählen der gemessenen Punkte. Alles andere wird von `im_do()` selbst erledigt. Je häufiger das Display erneuert wird, desto ruckfreier kann man das Modell während der Messung bewegt und gezoomt werden.

Die hier erreichte Lösung halte ich nicht für besonders zufriedenstellend, wenn es auch die bestmögliche zu sein scheint. Wesentlich besser wäre es, wenn der Benutzer die Anzahl der zu messenden Punkte nicht zu Beginn festlegen müßte und jederzeit das Messprogramm verlassen könnte. Dazu wäre es jedoch notwendig, Maus bzw. Tastatur abzufragen ohne die Kontrolle an CATIA zurückzugeben, d.h. daß die Tastaturabfrage auch zum Programm zurückkehrt, wenn der Benutzer keine Eingabe getätigt hat. Es gibt aber weder eine solche IUA Funktion noch einen Load in der CATIA Library. Abfangen der X–Windows Events wäre

umständlich aber möglich, wird jedoch von den CATIA Programmierrichtlinien verboten. Alle anderen denkbaren Lösungen würden zusätzliche Aktionen des Users erfordern, da kann er ebensogut das Programm jedesmal erneut starten.

Kapitel 7

Programmlistings

Hier folgen nun die gesammelten Programmlistings. Die Dateien sind natürlich auch auf der beiliegenden Diskette. Dateien, die automatisch erzeugt werden und natürlich Binärdateien sind hier nur kurz beschrieben. Objektdateien und Programme sind hier gar nicht berücksichtigt.

7.1 Gemeinsamer Quellcode

7.1.1 libimess/improtocol.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: common library
 *
 * file: improtocol.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file defines all stuff related to the
 * protocol used between the programs participating
 * in the Interaktive Messsoftware project.
 */

#ifndef IMPROTOCOL_H
#define IMPROTOCOL_H


/* The port used for the socket connection. */
#define SOCKET_PORT 1066

/* The maximum length of a message. All messages must be
   terminated by a newline ('\n'). */
#define MAX_MSG_LEN 80

/*
 * define the various messages between the programs participated
 * at InterMess. Each message begins with a word (max. 8 chars),
```

```
 * parameters are optional and separated by blanks. Parameters may
 * be any text but no binary information.
 */

/* This package contains current coordinates:
   message followed by 3 numbers (doubles). */
#define MSG_CURR_COORD "CurrCor"

/* This package contains a measured Point.
   message followed by 3 numbers (doubles). */
#define MSG_MEASURED_PT "MeasPt"

/* Start/Stop the machine.
   StartMv followed by 3 speeds (0 to 100) */
#define MSG_START_MOVE "StartMv"
#define MSG_STOP_MOVE "StopMv"

/* Say the Machine, the User wants a point exact here. */
#define MSG_FORCE_MEAS "ForcePt"


#endif /* IMPROTOCOL_H */
```

7.1.2 libimess/lib3d.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: common library
 *
 * file: lib3d.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file provides a little library to make handling
 * of 3D coordinates and vectors easier. There is no .cc file, because
 * the code part is very small.
 */

#ifndef LIB3D_H
#define LIB3D_H


#include <math.h>

#ifdef __cplusplus

class Double3d
{
  public:
    double x;
    double y;
    double z;

    void SetAll (double val) {
```

```
    x = y = z = val;
}

/* Calculate vector length. */
double Len (void) {
    return sqrt (x*x + y*y + z*z);
}

Double3d Abs (void) {
    Double3d a;

    a.x = fabs (x);
    a.y = fabs (y);
    a.z = fabs (z);
    return a;
}

Double3d operator + (Double3d a) {
    Double3d b;

    b.x = x + a.x;
    b.y = y + a.y;
    b.z = z + a.z;
    return b;
}

Double3d operator - (Double3d a) {
    Double3d b;

    b.x = x - a.x;
    b.y = y - a.y;
    b.z = z - a.z;
    return b;
}

Double3d operator * (double a) {
    Double3d b;

    b.x = x * a;
    b.y = y * a;
    b.z = z * a;
    return b;
}

Double3d operator / (double a) {
    Double3d b;

    b.x = x / a;
    b.y = y / a;
    b.z = z / a;
    return b;
}

bool operator != (Double3d a) {
    if (x != a.x || y != a.y || z != a.z)
        return true;
    else
        return false;
}
```

```
  private:
    /* nothing */
};


class Bool3d
{
  public:
    bool x;
    bool y;
    bool z;

    void SetAll (bool val) {
        x = y = z = val;
    }

  private:
    /* nothing */
};


#else /* __cplusplus */

typedef struct {
    double x;
    double y;
    double z;
} Double3d;

#endif /* __cplusplus */

#endif /* LIB3D_H */
```

7.1.3 libimess/error.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: common library
 *
 * file: error.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

#ifndef ERROR_H
#define ERROR_H


extern void WarningMsg (const char *message);
extern void ErrorMsg (const char *message);


#endif /* ERROR_H */
```

7.1.4 libimess/error.cc

```
/*
 * project: Interaktive Messsoftware
 *
 * program: common library
 *
 * file: error.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file provides two functions for handling Error
 * and Warning messages. A Warning returns while a Error aborts.
 */


#include <stdio.h>
#include <errno.h>


void
WarningMsg (const char *message)
{
    fprintf (stderr, "messd: Warning: ");
    fprintf (stderr, message);
    fprintf (stderr, "\t\n");
    if (errno)
        perror ("system means");
}


void
ErrorMsg (const char *message)
{
    fprintf (stderr, "messd: fatal Error: ");
    fprintf (stderr, message);
    fprintf (stderr, "\t\n");
    if (errno)
        perror ("system means");

    exit (1);
}
```

7.2 Der Quellcode von MESSD

7.2.1 messd/Makefile

```
#
# project: Interaktive Messsoftware
#
# program: messd
#
# file: Makefile
#
# Copyright: Markus Hitter, FH Trier 1996
```

```
#

#
# This file holds the OS independant part
#


.SUFFIXES: .o .c .cc .h

include Makefile.$(UNAME)



HEADERS = connection.h machine.h tpu.h mcu.h ../libimess/improtocol.h \
          ../libimess/3dlib.h
OBJECTS = messd.o machine.o connection.o tpu.o error.o

CFLAGS += -DVERSION=\"1.0\"


all: messd

messd: $(OBJECTS)
    $(CC) $(OBJECTS) $(LIBS) $(LDFLAGS) -o messd

error.o: ../libimess/error.h ../libimess/error.cc
    $(CXX) $(CFLAGS) $(INCLUDES) -c ../libimess/error.cc

tpu.o: tpu.c tpu.h
    $(CC) -c -O2 $(CFLAGS) tpu.c


.c.o: $(HEADERS)
    $(CC) $(CFLAGS) $(INCLUDES) -c $<

.cc.o: $(HEADERS)
    $(CXX) $(CFLAGS) $(INCLUDES) -c $<


clean:
    rm -f *.o core messd
```

7.2.2 messd/Makefile.Linux

```
#
# project: Interaktive Messsoftware
#
# program: messd
#
# file: Makefile.Linux
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the OS specific part for Linux
#
```

```
CC = gcc
CXX = g++
CFLAGS = -Wall
LDFLAGS = -lm -s

INCLUDES = -I$(HOME)/diplomarbeit/libimess
LIBS =
```

7.2.3 messd/tip-ball-4.0.rc

Dies ist ein Beispiel für eine `messdrc` Datei.

```
#
# project: Interaktive Messsoftware
#
# program: messd
#
# file: tip-ball-4.0.rc
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the configuration for messd when a
# mechanical tip with a ball diameter of 4mm is used.
# Link or copy this file to 'messdrc' in the same
# directory as the messd program to use it.
#


# currently valid commands are:
# SpeedLevel <multipier for moving speed>
# TipReversed <true or false>
# TipRadius <R>          (in mm, >= 0)

SpeedLevel 0.8
TipReversed false
TipRadius 2.0
```

7.2.4 messd/messd.cc

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: messd.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This is the file with the main loop. It coordinates the
 * machine with the incoming messages, which isn't too complicated.
 */
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

#include "lib3d.h"
#include "error.h"
#include "machine.h"
#include "connection.h"


static void sigHandler (int);

Machine m;
Connection conn;


int
main (int argc, char *argv[])
{
    struct timeval lastround, thisround;
    bool profiling = false;
    Double3d position;
    static Double3d oldPosition;
    static time_t sendtime, now;
    char msgbuf[MAX_MSG_LEN + 1], cmdbuf[9];

    /*
     * Parse the arguments.
     */
    for (int i=1; i<argc; i++) {
        if (! strcmp (argv[i], "-dc"))
            Connection::debug = true;
        else if (! strcmp (argv[i], "-dm"))
            Machine::debug = true;
        else if (! strcmp (argv[i], "-dt"))
            Machine::transdebug = true;
        else if (! strcmp (argv[i], "-r"))
            Machine::reboot = true;
        else if (! strcmp (argv[i], "-p"))
            profiling = true;
        else if (! strcmp (argv[i], "-v")) {
            printf ("Project: Interaktive Messsoftware.\n");
            printf ("Program: messd, Version %s\n", VERSION);
            exit (0);
        } else {
            printf ("Usage: messd [-dc] [-dm] [-p] [-h]\n\n");
            printf ("  -dc    Enable socket connection debug messages.\n");
            printf ("  -dm    Enable machine action debug messages.\n");
            printf ("  -dt    Enable transputer debug messages.\n");
            printf ("  -r     Force reboot of the transputer unit.\n");
            printf ("  -p     Turn profiling information on\n");
            printf ("  -v     Print Version info\n");
            printf ("  -h     Print this help.\n");
            exit (0);
```

```
    }
}

/*
 * Set our own signal handler. We have to stop the machine
 * the program is left. Most signals force to leave the
 * program as fast as possible.
 */
for (int i=0; i<16; i++)
    signal (i, sigHandler);

/*
 * Initialize the machine.
 */
m.Init ();
m.SearchReference ();

/*
 * MAIN LOOP
 */
while (true) {
    if (profiling) {
        gettimeofday (&thisround, NULL);
        if (lastround.tv_usec > thisround.tv_usec)
            lastround.tv_usec -= 1000000L;
        printf ("%ld us\n", thisround.tv_usec - lastround.tv_usec);
        lastround = thisround;
    }

    /*
     * Check for a incoming message and wait a short time.
     * If there is a message, handle it.
     */
    conn.WaitGetMessage (msgbuf);

    if (msgbuf[0] != 0) {
        /* There is a message, do the requested work. */
        if (! strncmp (msgbuf, MSG_STOP_MOVE, sizeof (MSG_STOP_MOVE)-1))
            /* Stop the machine. */
            m.StopMove ();
        else if (! strncmp (msgbuf, MSG_START_MOVE,
                            sizeof (MSG_START_MOVE) - 1)) {
            /* Start the machine. */
            Double3d speed;

            sscanf (msgbuf, "%s %lf %lf %lf", cmdbuf,
                    &speed.x, &speed.y, &speed.z);
            m.StartMove (&speed);
        } else if (! strncmp (msgbuf, MSG_FORCE_MEAS,
                              sizeof (MSG_FORCE_MEAS) - 1)) {
            /*
             * Force a measurement in the current position.
             * This means, just broadcast the current postion
             * as a measured point.
             */
            m.GetCurrentPos (&position);
            sprintf (msgbuf, "%s %.3f %.3f %.3f", MSG_MEASURED_PT,
                     position.x, position.y, position.z);
            conn.Broadcast (msgbuf);
```

```
            }
        }

        /*
         * Check for a measured point.
         */
        if (m.CheckMove (&position)) {
            /* A point was measured. */
            sprintf (msgbuf, "%s %.3f %.3f %.3f", MSG_MEASURED_PT,
                     position.x, position.y, position.z);
            conn.Broadcast (msgbuf);
        }

        /*
         * Write the current position to all open connections,
         * but at most once per second and only if it
         * has changed since the last time sended.
         */
        if (sendtime != time (&now)) {
            sendtime = now;
            m.GetCurrentPos (&position);
            if (position != oldPosition) {
                sprintf (msgbuf, "%s %.3f %.3f %.3f", MSG_CURR_COORD,
                         position.x, position.y, position.z);
                conn.Broadcast (msgbuf);
                oldPosition = position;
            }
        }
    }
    return 0;
}


void
sigHandler (int num)
{
    char str[30];

    sprintf (str, " %s signal (%d)", sys_siglist[num], num);
    ErrorMsg (str);
}
```

7.2.5 messd/tpu.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: tpu.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

#ifndef TPU_H
#define TPU_H
```

```
#define RWCOUNT 20000

#ifdef __cplusplus
extern "C" {
#endif

int InitLink ();
int ReadLink (char *Buffer, unsigned int Count);
int WriteLink (char *Buffer, unsigned int Count);
int ResetLink ();

#ifdef __cplusplus
}
#endif


#endif /* TPU_H */
```

7.2.6 messd/tpu.c

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: tpulink.c
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file contains the code which does the direct access to the
 * Hardware. It must be compiled with the same compiler and the
 * same optimisation as the kernel was compiled. Compilation
 * under DOS will probably work no more, but the needed changes
 * should be no problem.
 *
 * original file:
 * * Project:MCU-6 TOOLSET Software (User)
 * *
 * * Function:Link module for TPU-6002 in IBM (and compatible) PCs
 * *    Copyright (c) INMOS Ltd., 1988.
 * *    Copyright (c) Rîsch & Walter GmbH 1994
 * *    All Rights Reserved.
 * *
 * * Revision History:
 * * -----------------
 * * 14/12/93 V1-1RRcreated
 *
 * modified for the needs of Linux:
 * - get the IO permissions to be able to read/write directly
 *   to the hardware.
 * - modified io-calls (outb(), inb()).
 *
 */
```

```
#include <stdio.h>

#ifndef DOS
#include <unistd.h>
#endif

#ifdef linux
#include <asm/io.h>
#endif

#include "tpu.h"


/*
 * Offsets of the registers on the TPU-6002 card.
 */
/* Link input data-register */
#define C012_IDR 0x300
/* Link output data-register */
#define C012_ODR 0x301
#define C012_ISR 0x302 /* Link inbut status-register,
                          if bit 0 is "1", data can be read */
#define C012_OSR 0x303 /* Link output status-register,
                          if bit 0 is "1", data can be written */
#define C012_RESET 0x310 /* TPU-6002 reset register (output) */
#define C012_ERROR 0x310 /* TPU-6002 Error Flag register (input) */
#define C012_ANALYSE 0x311 /* TPU-6002 analyse register (output) */


/*
 * Initialise the Link.
 */
int
InitLink ()
{
#ifndef NO_HARDWARE

    if ((ioperm (C012_IDR, 1, 1)) || (ioperm (C012_ODR, 1, 1)) ||
        (ioperm (C012_ISR, 1, 1)) || (ioperm (C012_OSR, 1, 1)) ||
        (ioperm (C012_RESET, 1, 1)) || (ioperm (C012_ANALYSE, 1, 1))) {
        fprintf (stderr, "Can't get IO permissions\n");
        return 1;
    }

    /*
     * Clear the status registers.
     */
    /*Markus cli ();*/
    outb (0, C012_ISR);
    outb (0, C012_OSR);
    /*Markus sti();*/

#endif /* ! NO_HARDWARE */

#ifdef DEBUG
    printf ("Link initialized.\n");
#endif
    return 0;
```

```
}


/*
 * Read bytes from the Link.
 */
int
ReadLink (char *buffer, unsigned int count)
{
    register int n, l ;

    if (count < 1) return -1;

#ifndef NO_HARDWARE

#if DEBUG
    printf ("Reading %d Byte:", count);
    fflush (stdout);
#endif

    /*Markus cli ();*/
    for (n=0; count--; n++) {
        for (l=0; l<RWCOUNT; l++) {
            if (inb (C012_ISR) == 1) {
                *buffer = inb (C012_IDR) & 0xff;
                buffer++;
                break;
            }
#ifdef DOS
        kbhit ();
#endif
        }
        if (l >= RWCOUNT) break;
#ifdef DEBUG
        printf (" %02X", *(buffer-1) & 0xff);
        fflush (stdout);
#endif
    }
    /*Markus sti ();*/
#ifdef DEBUG
    printf (" (%d read)\n", n);
#endif
    return n;

#else /* NO_HARDWARE */
    return count;
#endif /* NO_HARDWARE */
}


/*
 * Write bytes to the Link.
 */
int
WriteLink (char *buffer, unsigned int count)
{
    register int n, l;

    if (count < 1) return -1;
```

```
#if DEBUG
    printf ("Writing %d Byte:", count);
    fflush (stdout);
#endif

#ifndef NO_HARDWARE

    /*Markus cli ();*/
    for (n=0; count--; n++) {
        for (l=0; l<RWCOUNT; l++) {
            if (inb (C012_OSR) == 1) {
                outb ((*buffer & 0xff), C012_ODR);
                buffer++;
                break;
            }
#ifdef DOS
          kbhit ();
#endif
        }
        if (l >= RWCOUNT) break;
#ifdef DEBUG
        printf (" %02X", *(buffer-1) & 0xff);
        fflush (stdout);
#endif
    }
    /*Markus sti ();*/

#ifdef DEBUG
    printf (" (%d written)\n", n);
#endif
    return n;

#else /* NO_HARDWARE */
    return count;
#endif /* NO_HARDWARE */
}


/*
 * Reset Link and Transputer(s),
 * only required in some bad situations,
 * or before booting.
 */
int
ResetLink ()
{
#ifndef NO_HARDWARE

    /*Markus cli ();*/
    outb (0, C012_ANALYSE); /* deassert analyse */
    usleep (100000);
    outb (0, C012_RESET); /* deassert reset */
    usleep (100000);
    outb (1, C012_RESET); /* assert reset */
    usleep (100000);
    outb (0, C012_RESET); /* deassert reset */
    usleep (100000);
    /*Markus sti ();*/
```

```
#endif /* ! NO_HARDWARE */

    return 1;
}
```

7.2.7 messd/mcu.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: mcu.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

 /*
  * most stuff copied from mcusrvr.h-1.71.
  */

#ifndef MCU_H
#define MCU_H


/*
 * The bit numbers used at the Output of the ASM 2003.
 */
/* The reference switches are connected here. */
#define REFERENCE_BIT 7L
/* The hardware limit switches */
#define LIMIT_SWITCH_BIT 8L
/* The measure instrument line */
#define MEAS_INSTR_BIT 9L

/*
 * The bit numbers used at the Input of the ASM 2003.
 */
/* The lines where the limit switches are connected. positive side,
   negative side and both together. */
#define POSITIVE_LIMIT_BIT 2L
#define NEGATIVE_LIMIT_BIT 4L
#define ALL_LIMIT_BIT 6L
/* The line where the switch in the measurement tip is connected. */
#define TIP_BIT 8L

/*
 * Some data needed for checking/loading the MCU-6 operating system.
 */
/* Current revision of transputer operating system. */
#define REV_NR "OS MCU-6 V1.71"
/* Size of this strung in the MCU-6's memory. */
#define SIZE_STRREV40
/* Name of the file containing the MCU-6 OS. */
#define BOOTFILENAME "rwtos.btl"
/* Name of the file containing the MCU-6 system settings,
   like acclerations, filter settings etc. */
```

```
#define SYSFILENAME"system.dat"

/* time in microseconds to wait for closing/opening a relay */
#define RELAYTIME 20000


char enum TSROPC {
    /* Don't change order !! */
    UF,         CTRU,
    WRJAC,      WRJVL,    WRJTVL,
    WRHAC,      WRHVL,
    WRRP,       WRDP,
    SHP,        WRASMO,   WRASMOB,
    RDRP,       RDDP,     RDTP,
    RDDV,
    RDASMI,     RDASMIB,
    RDASMS,     RDASMSB,
    RDASMEPC,
    RDASMO,     RDASMOB,
    RDLSM,
    RDAXST,     RDAXSTB,
    RDJAC,      RDJVL,    RDJTVL,
    RDHAC,      RDHVL,

    CL,         OL,
    RA,         RS,       AZO,
    JR,         JA,       JHI,
    JHL,        JHR,
    MLR,        MLA,
    MCR,        MCA,
    MHR,        MHA,
    SMLR,       SMLA,
    SMCR,       SMCA,
    SMHR,       SMHA,
    SSMS,
    SSTPS,      SDELS,
    TXBF,
    STARTCNCT, STOPCNCT, CONTCNCT,
    STEPCNCT,
    RDCNCTS,
    RDCBCNCT,  WRCBCNCT,
    RDCI,RDCD,
    WRCI,WRCD,
    RDMCP,WRMCP,

    WRLEDRD,   WRLEDYL,  WRLEDGN,
    RDLEDRD,   RDLEDYL,  RDLEDGN,
    RDIRQPC,

    MCUINIT,   DUMMY,
    WRJOVR,    RDJOVR,
    WRTROVR,   RDTROVR,
    UTROVR,
    RDF,       RDAP,
    RDSLL,     WRSLL,    RDSLR,
    WRSLR,     RDIPW,    WRIPW,
    RDMPE,     WRMPE,
    RDSDEC,    WRSDEC,
    SPD,
```

```
    WRGF,       RDGF,
    SPVTD,
    RDLP,       WRLP,
    MS,JS,
    LPS,
    RDAUX,      WRAUX
};

union AXST
{
    longstatus_word;
    struct {
        /* axst: Error- & Status-Bits */
        unsigned toasm : 1; /* Bit 0:  Timeot ASM (communication) */
        unsigned eo    : 1; /* Bit 1:  Emergency out */
        unsigned dnr   : 1; /* Bit 2:  Drive not ready */
        unsigned lslh  : 1; /* Bit 3:  Limit switch left hardware */
        unsigned lsrh  : 1; /* Bit 4:  Limit switch right hardware */
        unsigned lsls  : 1; /* Bit 5:  Limit switch left software */
        unsigned lsrs  : 1; /* Bit 6:  Limit switch right software */
        unsigned mpe   : 1; /* Bit 7:  Maximum Position Error */
        unsigned dhef  : 1; /* Bit 8:  data handling error flag */
        unsigned cef   : 1; /* Bit 9:  configuration error flag */
        unsigned       : 2; /* Bit 10..11: currently not used */

        /* Status-Bits */
        unsigned pe   : 1; /* Bit 12: Profile end */
        unsigned cl   : 1; /* Bit 13: Closed loop */
        unsigned ip   : 1; /* Bit 14: In Position */
        unsigned ui    : 1; /* Bit 15: user input */
        unsigned lpsf  : 1; /* Bit 16: latched position synchronous flag */
    } bit;
};

struct TOSI
{
    /* Transputer Operating System Informations */
    char    revision[SIZE_STRREV];  /* software-revision-string */
    long    Number_Axis;            /* number for axis */
    long    SysFile_Loaded;         /* status flag for sysfile loaded */
};


#endif /* MCU_H */
```

7.2.8 messd/connection.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: connection.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */
```

```
#ifndef CONNECTION_H
#define CONNECTION_H


#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/time.h>

#include "improtocol.h"


class Connection
{
  public:
    Connection ();
    ~Connection ();

    void WaitGetMessage (char buffer[MAX_MSG_LEN]);
    void Broadcast (const char *);

    static bool debug;

  private:
    int sock;
    int fdWidth;
    struct sockaddr_in server;
    fd_set connSet;
};


#endif /* CONNECTION_H */
```

7.2.9 messd/connection.cc

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: connection.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file contains the code for handling the socket connection
 * to the other programs. Since messd is the server in this project,
 * the routines here are somewhat more complicated than in the other
 * programs.
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>

#include "error.h"
#include "lib3d.h"

#include "connection.h"
#include "machine.h"


extern Machine m;

bool Connection::debug = false;


/*
 * Constructor: open the socket port. This is the server
 * and usually must be started first. For an Example similar to this
 * routine see AIX InfoExplorer: How to Reuse a Socket Port...
 */
Connection::Connection ()
{
    int on = 1;

    /* Get the maximum number of open file descriptors. */
    fdWidth = getdtablesize ();
    /* Do not overdrive: */
    if (fdWidth > 50) fdWidth = 50;

    /* Create a virtual-circuit, TCP, socket. */
    sock = socket (AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
        ErrorMsg ("Could not open socket");

    /* Allow the socket to reuse address if already exists. */
    if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, (char *)&on,
                    sizeof (on)) < 0)
        ErrorMsg ("Could not set socket for reusing");

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl (INADDR_ANY);
    server.sin_port = htons ((unsigned short)SOCKET_PORT);
#ifndef linux
    server.sin_len = sizeof (server);
#endif

    if (bind (sock, (sockaddr *)&server, sizeof (server)) < 0)
        ErrorMsg ("Could not bind socket");

    if (listen (sock, 5)) {
        char msg[100];
        sprintf (msg, "Unable to listen on port %d", SOCKET_PORT);
        ErrorMsg (msg);
    }

    if (debug) printf ("Listening INET on port %d\n", SOCKET_PORT);

    /* Initialize the file descriptor set. */
```

```
    FD_ZERO (&connSet);
    FD_SET (sock, &connSet);
}


/*
 * Destructor: close the socket.
 */
Connection::~Connection ()
{
    shutdown (sock, 2);
    close (sock);
}


/*
 * Checks all Clients for incoming messages and for new clients.
 * New Clients are connected, one message is given back to the
 * calling procedure. Additional messages are held for the next
 * time called. This routine also waits for a 10th of a second.
 */
void
Connection::WaitGetMessage (char buffer[])
{
    struct timeval sleeptime = {0, 100000};
    fd_set readySet;
    int fd, len, ret;

    /* Delete a old message */
    buffer[0] = 0;

    memcpy (&readySet, &connSet, sizeof (readySet));

    /* Check for messages and wait in one command. */
    select (fdWidth, &readySet, 0, 0, &sleeptime);

    /* Did we get a connection request ? If so, add it to
       the list of open connections */
    if (FD_ISSET (sock, &readySet)) {
        if (debug) printf ("New Connection.\n");
        fd = accept (sock, (struct sockaddr *)0, (int *)0);
        if (fd < 0)
            WarningMsg ("Couldn't accept new connection");
        else {
            char msgbuf[45];
            Double3d pos;

            FD_SET (fd, &connSet);
            if (debug) printf ("New Connection %d accepted.\n", fd);
            /* Here we could open a stream */

            /* Send the current position to the beginner */
            m.GetCurrentPos (&pos);
            sprintf (msgbuf, "%s %.3f %.3f %.3f", MSG_CURR_COORD,
                     pos.x, pos.y, pos.z);
            Broadcast (msgbuf);
        }
    }
```

```
    /*
     * This code should be modified if there are continuously more
     * than 1 message per call. For now the program connected first has
     * the highest priority.
     */
    for (fd=0; fd<=fdWidth; fd++) {
        if ((fd != sock) && FD_ISSET (fd, &readySet)) {
            /* Read the line character by character to find
               the end of the line. There might be more than one
               line in the queue. */
            if (debug) printf ("Reading (%d): ", fd);
            for (len=0; len<MAX_MSG_LEN; len++) {
                ret = read (fd, &buffer[len], 1);
                if (ret == 1) {
                    if (debug) {
                        putchar (buffer[len]);
                        fflush (stdout);
                    }
                    if (buffer[len] == '\n') {
                        buffer[len] = 0;
                        break;
                    }
                } else if (ret == 0 && len == 0) {
                    if (debug) printf ("Closing connection %d\n", fd);
                    close (fd);
                    FD_CLR (fd, &connSet);
                    break;
                } /* else error while reading or truncated message.
                     - ignore it. */
            }

            /* Handle only messages here, which relate to a specific
               connection. Most messages are not of this type and
               are handled in the main loop. */

            return; /* 'cuz we have a message or message is empty. */
        }
    }
}


/*
 * Send a message to all connected programs. Do not confuse this
 * with an generic IP broadcast.
 */
void
Connection::Broadcast (const char *message)
{
    int fd, msglen;
    char newline = '\n';

    if (debug)
        printf ("Broadcasting: %s (%d bytes)\n", message, strlen (message));

    msglen = strlen (message) + 1;

    for (fd=0; fd<=fdWidth; fd++)
        if ((fd != sock) && FD_ISSET (fd, &connSet)) {
            write (fd, message, msglen);
```

```
            write (fd, &newline, 1);
        }
  }
```

7.2.10 messd/machine.h

```
  /*
   * project: Interaktive Messsoftware
   *
   * program: messd
   *
   * file: machine.h
   *
   * Copyright: Markus Hitter 1996
   */

  #ifndef MACHINE_H
  #define MACHINE_H


  #include "improtocol.h"
  #include "lib3d.h"
  #include "mcu.h"


  class Machine
  {
    public:
      Machine ();
      ~Machine ();

      void Init ();
      void SearchReference ();

      /* Begin the axles moving with three explicit speeds for each axis */
      void StartMove (Double3d *speed);
      void StopMove ();

      /* Check for tip, limits, etc... Returns wether a point was measured.
         The coordinates contain the measured point. */
      bool CheckMove (Double3d *measPoint);

      void GetCurrentPos (Double3d *position);
      void GetLatchedPos (Double3d *position);

      static bool debug;
      static bool transdebug;
      static bool reboot;

    private:
      /* Variables for the transputer I/O. Fromerly this stuff was declared
         in mcusrvr.h, this and mcu.h is the small extract we need. */
      union AXST xAxst, yAxst, zAxst;
      long xAsmi, yAsmi, zAsmi;

      /* The time at the last stop, used for automatic
         opening of the loops after 5 seconds. */
```

```
    time_t stoptime;

    /* Tells wether the machine is moved by motor. */
    bool running;
    /* Tells wether the position hold loops are closed. */
    bool loopsClosed;
    /* Tells wether the machine moves currently in the limits. */
    Bool3d inLimit;

    /* Current target moving direction. */
    Double3d movingDir;

    /* Multiplicator for real speed calculation. */
    double speedLevel;
    /* Type of switch used in the tip. */
    bool tipReversed;
    /* Radius of the tip ball. */
    double tipRadius;

    void ReBoot ();

    void OpenLoops ();
    void CloseLoops ();

    void JogAbsolute (Double3d *targetPosition);
    void JogRelative (Double3d *targetPosition);

    void WaitProfileEnd ();

    void SetAsmOutput (long bitnr, long value);
#define HARDWARE_LIMIT_SWITCH_ON SetAsmOutput (LIMIT_SWITCH_BIT, true);
#define HARDWARE_LIMIT_SWITCH_OFF SetAsmOutput (LIMIT_SWITCH_BIT, false);
#define REFERENCE_SWITCH_ON SetAsmOutput (REFERENCE_BIT, true);
#define REFERENCE_SWITCH_OFF SetAsmOutput (REFERENCE_BIT, false);
#define MEAS_INSTR_ON SetAsmOutput (MEAS_INSTR_BIT, true);
#define MEAS_INSTR_OFF SetAsmOutput (MEAS_INSTR_BIT, false);
    void ReadAsmInput ();
    void ReadAsmStatus ();

    void ReadAxesStatus ();

    void ResetAxes ();
    void GoHomePos (long axis);
    void GoHomeNeg (long axis);
    void GoIndex (Double3d *targetPosition);
    void SetHomePosition ();

    void SetHomeVelocities (Double3d *speed);
    void SetJogVelocities (Double3d *speed);
    void SetJogTargetVelocities (Double3d *speed);

    /* some useful functions, formerly in tpulink.c */
    inline void OpCodeToLink (char opCode);
    inline void SendAxTab ();
    inline void LongToLink (long value);
    inline void DoubleToLink (double value);
    inline long LongFromLink ();
    inline double DoubleFromLink ();
    inline char ByteFromLink ();
```

```
    int WriteBootFile (char *filename);
    int WriteSystemFile (char *filename);
    void ReadResourceFile ();

    void DebugStatusReport (bool header);
    void SpeedReport ();
};


#endif /* MACHINE_H */
```

7.2.11 messd/machine.cc

```
/*
 * project: Interaktive Messsoftware
 *
 * program: messd
 *
 * file: machine.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file holds all the code for controlling the
 * machine via the MCU-6.
 *
 * A lot of the routines here have a direct counterpart
 * in mcusrvr.c, which was delivered with the MCU-6 but
 * is no more in use.
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#include "machine.h"
#include "tpu.h"
#include "error.h"

#define X_AXIS 0
#define Y_AXIS 1
#define Z_AXIS 2


bool Machine::debug = false;
bool Machine::transdebug = false;
bool Machine::reboot = false;


/*
```

```
 * Constructor: Initialise the Link to the MCU-6 and a lot
 * of other 'nearly constant' variables.
 */
Machine::Machine ()
{
    if (InitLink ())
        ErrorMsg ("Could not open MCU 6 Link");

    /* Initialize configuration variables. */
    speedLevel = 0.1;
    tipReversed = false;
}


/*
 * Destructor: Stop the Machine and switch all off.
 */
Machine::~Machine ()
{
    HARDWARE_LIMIT_SWITCH_OFF
    MEAS_INSTR_OFF
    ResetAxes ();
}


/*
 * Init the MCU system. If it fails or a reboot should be done, reboot the
 * unit and try it again. There is also guessed, about the unit is powered
 * on. Reading the system file and the resource file is also done here.
 */
void
Machine::Init ()
{
    int lderr;
    struct TOSI tosi;

    /* clear all fields */
    for (int i=0; i<(int)sizeof (tosi.revision); i++)
        tosi.revision[i] = 0;
    tosi.Number_Axis = 0;
    tosi.SysFile_Loaded = 0;

    if (reboot) {
        ReBoot ();
        reboot = false;
    }

    while (true) {
        /* Get info from MCU about it's TOS */
        OpCodeToLink (0xFF);
        ByteFromLink ();
        ReadLink (tosi.revision, sizeof (tosi.revision));
        tosi.Number_Axis = LongFromLink ();
        tosi.SysFile_Loaded = LongFromLink ();
        if (transdebug)
            printf ("%s, %ld Axes\n", tosi.revision, tosi.Number_Axis);

        /* check for the correct TOS */
        if (strcmp (tosi.revision, REV_NR)) {
```

```
            WarningMsg ("Wrong TOS in MCU 6. Will reboot and try again..");
            ReBoot ();

            /* Get info from MCU again */
            OpCodeToLink (0xFF);
            ByteFromLink ();
            ReadLink (tosi.revision, sizeof(tosi.revision));
            tosi.Number_Axis = LongFromLink ();
            tosi.SysFile_Loaded = LongFromLink ();
            if (transdebug)
                printf ("%s, %ld Axes\n", tosi.revision, tosi.Number_Axis);

            if (strcmp (tosi.revision, REV_NR))
                ErrorMsg ("MCU 6 init failed again, giving up");
        }

        printf ("Transputer is running %s, %ld Axes\n",
                tosi.revision, tosi.Number_Axis);

        if (! tosi.SysFile_Loaded) {
            lderr = WriteSystemFile (SYSFILENAME);
            if (lderr)
                ErrorMsg ("Failed to load system file");
        }

        /* Check the error flags and about it makes sense to go on. */
        ReadAxesStatus ();
        if (xAxst.bit.toasm || yAxst.bit.toasm || zAxst.bit.toasm ||
            xAxst.bit.cef || yAxst.bit.cef || zAxst.bit.cef) {
            char answer;
            printf ("\nASM2003 seems not to work properly\n");
            printf ("Is the power switched ON and all else OK ?\n\n");
            printf ("retry (r), abort (a), ignore (i) ? [r] ");
            fflush (stdout);
            answer = getchar ();
            answer = tolower (answer);
            if (answer == 'i') break;
            else if (answer == 'a') exit (1);
        } else
            break;

    }

    /* Initialize acclerations, which should probably never be changed.
       Velocities are set when needed. */
    ResetAxes ();
    ReadResourceFile ();
}


/*
 * Move the axes to the home position. This has to test a lot of
 * possibilities where the machine is BEFORE it reaches the home position.
 * If the machine is already near the home position, move it somewhat
 * outside to be sure.
 */
void
Machine::SearchReference ()
{
```

```
Double3d vvv;
char answer;
bool xOK, yOK, zOK;
time_t start, xStop, yStop, zStop;

ResetAxes ();

/* Set reasonable speeds */
vvv.SetAll (30); /* mm/s */
SetHomeVelocities (&vvv);
SetJogVelocities (&vvv);

/* Activate end switches */
HARDWARE_LIMIT_SWITCH_ON

/*
 * Under some circumstances is might be useful NOT to move the
 * machine to the home position. Ask the user what to do.
 */
printf ("\nShould we move the machine to the home position ?\n");
printf ("This requires unmounting the mesurement tip.\n\n");
printf ("yes (y), no (n) ? [n] ");
fflush (stdout);
answer = getchar ();
answer = tolower (answer);
if (answer == 'y') {
    /*
     * Prepare the axes.
     */
    CloseLoops ();
    REFERENCE_SWITCH_ON
    ReadAsmInput ();
    ReadAxesStatus ();
    if (transdebug) DebugStatusReport (true);
    xOK = yOK = zOK = false;

    printf ("messd: Reference move, please wait...\n");

    /*
     * If the machine is in the negative hardware limit, move
     * 100 mm to the positive side. 100 mm seems to be enough for
     * moving at least between the positive and negative side of
     * the home position. This is required for the next step.
     */
    vvv.SetAll (0);
    if (xAsmi & NEGATIVE_LIMIT_BIT) {
        vvv.x = 180;
    }
    if (yAsmi & NEGATIVE_LIMIT_BIT) {
        vvv.y = 180;
    }
    if (zAsmi & NEGATIVE_LIMIT_BIT) {
        vvv.z = 180;
    }
    JogRelative (&vvv);
    WaitProfileEnd ();

    /*
     * If the machine is in the positive hardware limit,
```

```
 * we are already on the positive side of the home position.
 *
 * Search now in negative direction. We'll find the positive side
 * of the Home position or run into negative hardware limit.
 * If the search takes less than 2 seconds, we were to near or
 * inside the home position and have to research from a better
 * position.
 */
start = time (NULL);
xStop = yStop = zStop = 0;
GoHomeNeg (X_AXIS);
GoHomeNeg (Y_AXIS);
GoHomeNeg (Z_AXIS);
do {
    usleep (200000);
    ReadAsmInput ();
    ReadAxesStatus ();
    if (transdebug) DebugStatusReport (false);
    /* Reverse direction in case of reaching hardware limit. */
    if (xAsmi & NEGATIVE_LIMIT_BIT) {
        GoHomePos (X_AXIS);
        xStop = start; /* Misuse xStop as a flag... */
    }
    if (yAsmi & NEGATIVE_LIMIT_BIT) {
        GoHomePos (Y_AXIS);
        yStop = start;
    }
    if (zAsmi & NEGATIVE_LIMIT_BIT) {
        GoHomePos (Z_AXIS);
        zStop = start;
    }
    /* Remember the time when a axis stopped. */
    if (! xStop && xAxst.bit.pe)
        xStop = time (NULL);
    if (! yStop && yAxst.bit.pe)
        yStop = time (NULL);
    if (! zStop && zAxst.bit.pe)
        zStop = time (NULL);
} while (! (xStop & yStop & zStop));
/*
 * The search in reverse direction might need additional time.
 */
WaitProfileEnd ();

/*
 * Drive now all questionable axes out and retry the task.
 * Driving 100 mm out seems to be enough for re-searching
 * the positive side of the home position.
 */
vvv.SetAll (0);
if ((xStop - start) < 2)
    vvv.x = 100;
if ((yStop - start) < 2)
    vvv.y = 100;
if ((zStop - start) < 2)
    vvv.z = 100;
JogRelative (&vvv);
WaitProfileEnd ();
GoHomeNeg (X_AXIS);
```

```
        GoHomeNeg (Y_AXIS);
        GoHomeNeg (Z_AXIS);
        WaitProfileEnd ();

        /*
         * Now we are on the positive side of the home position.
         * Begin the Index search loop.
         */

        /*
         * Look in negative direction util zero track or
         * hardware limit reached.
         */
        vvv.SetAll (5); /* mm/s */
        SetHomeVelocities (&vvv);
        vvv.SetAll (20); /* mm/s */
        GoIndex (&vvv);
        do {
            usleep (200000);
            ReadAxesStatus ();
            ReadAsmInput ();
            if (transdebug) DebugStatusReport (false);
        } while (! (xAxst.bit.pe & yAxst.bit.pe & yAxst.bit.pe));

        /*
         * Now move very slowly in positve direction
         * (the slower, the exacter).
         */
        vvv.SetAll (0.1); /* mm/s */
        SetHomeVelocities (&vvv);
        vvv.SetAll (-160); /* mm/s */
        GoIndex (&vvv);
        do {
            usleep (200000);
            ReadAxesStatus ();
            ReadAsmInput ();
            if (transdebug) DebugStatusReport (true);
        } while (! (xAxst.bit.pe & yAxst.bit.pe & yAxst.bit.pe));

    } /* if (answer == 'y') */

    /* OK, the job should be finished.
       Set this position to zero, then leave. */
    usleep (200000);
    SetHomePosition ();
    OpenLoops ();
    REFERENCE_SWITCH_OFF
    /* HARDWARE_LIMIT_SWITCH is left on. */
    MEAS_INSTR_ON

    printf ("messd: OK\n");
}


/*
 * Start the machine moving. The speeds for the three axes are given
 * in percent of the maximum speed. Do not move the machine if we're
 * in a hardware limit and the movement command tries to get deeper
 * in this limit. Do not move at all, if the tip is touched, this means
```

```
 * the back-movement from the last measured point was unsuccessful.
 */
void
Machine::StartMove (Double3d *speed)
{
    Double3d jogSpeed, jogTarget;

    /*
     * Check the tip.
     */
    ReadAsmInput ();
    if ((xAsmi & TIP_BIT) != tipReversed) {
        WarningMsg ("Tip is touched, can't move");
        WarningMsg ("Move the tip by hand to untouch the tip");
        return;
    }

    /*
     * Check the hardware limits in all possible directions. If the
     * given speed whould lead deepre into the limit, set the speed to zero.
     */
    /* Without closed loops we'll have no valid hardware limit bits. */
    CloseLoops ();

    /* Check about we are in the Hardware limits and test about it
       might be allowed to move anyway (away from the Hardware limit). */
    ReadAsmInput ();
    if (transdebug) DebugStatusReport (true);
    if (xAsmi & POSITIVE_LIMIT_BIT) {
        inLimit.x = true;
        if (speed->x > 0)
            speed->x = 0;
    } else if (xAsmi & NEGATIVE_LIMIT_BIT) {
        inLimit.x = true;
        if (speed->x < 0)
            speed->x = 0;
    }
    if (yAsmi & POSITIVE_LIMIT_BIT) {
        inLimit.y = true;
        if (speed->y > 0)
            speed->y = 0;
    } else if (yAsmi & NEGATIVE_LIMIT_BIT) {
        inLimit.y = true;
        if (speed->y < 0)
            speed->y = 0;
    }
    if (zAsmi & POSITIVE_LIMIT_BIT) {
        inLimit.z = true;
        if (speed->z > 0)
            speed->z = 0;
    } else if (zAsmi & NEGATIVE_LIMIT_BIT) {
        inLimit.z = true;
        if (speed->z < 0)
            speed->z = 0;
    }

    /*
     * Calculate the needed speeds.
     */
```

```
    jogSpeed = speed->Abs () * speedLevel; /* Speed, but not direction. */

    /* Setting a speed to zero and start this axis might cause
       variable overflow in the TPU. */
    if (jogSpeed.x == 0) jogSpeed.x = 0.01;
    if (jogSpeed.y == 0) jogSpeed.y = 0.01;
    if (jogSpeed.z == 0) jogSpeed.z = 0.01;
    if (transdebug)
        printf ("Starting Machine: %f %f %f\n",
                speed->x, speed->y, speed->z);

    /*
     * Calculate the needed directions.
     * Note: Using target velocities whould prevent the
     *       the transputer from using moving profiles.
     */
    if (speed->x < 0) {
        jogTarget.x = -2000.0; /* Infinite far in negative direction. */
    } else if (speed->x == 0) {
        jogTarget.x = 0; /* Not really needed, but might prevent
                            variable overflow inside the TPU. */
    } else {
        jogTarget.x = 2000.0; /* Infinite positive. */
    }
    if (speed->y < 0) {
        jogTarget.y = -2000.0;
    } else if (speed->y == 0) {
        jogTarget.y = 0;
    } else {
        jogTarget.y = 2000.0;
    }
    if (speed->z < 0) {
        jogTarget.z = -2000.0;
    } else if (speed->z == 0) {
        jogTarget.z = 0;
    } else {
        jogTarget.z = 2000.0;
    }

    /*
     * Do a StopMove () to discard all previous move commands.
     */
    StopMove ();

    /*
     * Actually, start the machine.
     */
    SetJogVelocities (&jogSpeed);
    JogRelative (&jogTarget);
    running = true;

    /*
     * Remember the moving direction for later use.
     */
    movingDir = *speed;
}


/*
```

```
 * Stop the machine. If the machine moves currently fast,
 * it moves perhaps somewhat further.
 */
void
Machine::StopMove ()
{
    running = false;

    /* Formerly js() */
    OpCodeToLink (JS);
    SendAxTab ();

    stoptime = time (NULL);
}


/*
 * Checks for everything that can happen while the machine
 * is moving or stopped:
 * Hardware limits, measured points etc.
 *
 * Also checks while stopped about it's time to open the loops.
 *
 * This Routine should be called as often as possible.
 */
bool
Machine::CheckMove (Double3d *position)
{
    Double3d thisPos, distance;
    static Double3d lastPos;
    static time_t t, measureTime = time (NULL);

    ReadAsmInput ();
    ReadAxesStatus ();

    if (transdebug && loopsClosed) {
        if (t != time (NULL)) {
            DebugStatusReport (false);
            t = time (NULL);
        }
    }

    if (running) {
        /*
         * Check the hardware limits.
         *
         * If we've started inside a limit, 'inLimit' were set,
         * so check if we're already outside the limits and clear the flag.
         *
         * If we've run into hardware limits, just stop the machine.
         * This notifies the user, he will be able to restart the
         * machine and StartMove() will check that this is in a
         * usable direction.
         */
        if (inLimit.x) {
            if (! (xAsmi & ALL_LIMIT_BIT))
                inLimit.x = false;
        } else {
            if (xAsmi & ALL_LIMIT_BIT)
```

```
                StopMove ();
        }
        if (inLimit.y) {
            if (! (yAsmi & ALL_LIMIT_BIT))
                inLimit.y = false;
        } else {
            if (yAsmi & ALL_LIMIT_BIT)
                StopMove ();
        }
        if (inLimit.z) {
            if (! (zAsmi & ALL_LIMIT_BIT))
                inLimit.z = false;
        } else {
            if (zAsmi & ALL_LIMIT_BIT)
                StopMove ();
        }
    } else { /* if (running) */
        if (loopsClosed) {
            /*
             * Open the loops after some time to allow measurement 'by hand'.
             */
            if (time (NULL) > stoptime + 7) {
                OpenLoops ();
                GetCurrentPos (&lastPos);
            }
        } else {
            /*
             * Try to calculate the current moving Direction, even if moving
             * by hand. Do this by comparing the current position with the
             * position a short time before. Accept the calculation only if
             * the distance from this last Position is at least 0.5 mm.
             *
             * This means that radius correction for measurement by
             * hand is only valid if the moving distance before a measurement
             * is at least 0.5 mm.
             */
            GetCurrentPos (&thisPos);
            distance = lastPos - thisPos;
            if (distance.Len () >= 1) {
                movingDir = distance;
                lastPos = thisPos;
            }
        }
    }

    /*
     * Check for a latched (= measured) position. The latched position
     * is configured to be triggered by the measurement tip. The
     * corresponding bit is only cleared by reading out the values.
     * NOTE: The electronics must be wired to trigger all three axes
     *       with the one signal from the tip. Otherwise the non-connected
     *       axes won't get results.
     */
    if (xAxst.bit.lpsf) {
        if (transdebug) DebugStatusReport (false);

        GetLatchedPos (position);
        if (debug) printf ("measured position: %.3f, %.3f, %.3f\n",
                           position->x, position->y, position->z);
```

```
/*
 * Even after a StopMove() the machine might move a little bit,
 * this could cause to activate the tip. 'if (running)' were the
 * wrong indicator, because we have to move back in this case.
 */
if (loopsClosed) {
    measureTime = time (NULL);

    /*
     * Move the machine 0.5 mm back behind the measured position.
     * The movingDir value is valid even after a StopMove().
     */
    JogAbsolute (&(*position - movingDir * 0.5 / movingDir.Len ()));

    /*
     * We don't need to wait for the back-movement to be finished,
     * but we have to wait at least until the tip has released.
     * Otherwise we whould allow to move with an unreleased tip.
     *
     * In rare cases the back-movement fails because the
     * moving direction is nearly the same as the side of the
     * measurement object. Moving back doesn't give then really
     * a distance between the object and the tip. In this case
     * there is no sure way to release the tip; the user has to
     * do this by hand.
     */
    do {
        usleep (100000);
        ReadAsmInput ();
    } while (((xAsmi & TIP_BIT) != tipReversed) &&
             (time (NULL) <= measureTime + 5));

    /*
     * Check for failure.
     */
    if ((xAsmi & TIP_BIT) != tipReversed) {
        WarningMsg (
            "Could not free the tip automatically, do it by hand");
        OpenLoops ();
        GetCurrentPos (&lastPos);
        /* Do not return here, the measurement is poor but valid. */
    }

    /*
     * Read the latched position again to be sure to have the
     * latch mechanism cleared.
     */
    usleep (200000);
    GetLatchedPos (&thisPos);
} else if (time (NULL) <= measureTime + 1) {
    /*
     * Point was measured 'by hand'.
     * Allow only one measurement 'by hand' in 1 - 2 seconds
     * to avoid "noise" on a bad surface.
     */
    if (debug) printf (" - ignored\n");
    return false;
} else {
```

```
            /*
             * Point was measured valid 'by hand'.
             */
            measureTime = time (NULL);
        }

        /* This is common to all valid measurements. */

        /*
         * Inform the user with a beep.
         */
        printf ("\a");
        fflush (stdout);

        /*
         * Do the radius correction.
         */
        *position = *position + movingDir / movingDir.Len () * tipRadius;
        if (debug)
            printf ("After radius correction: %.3f %.3f %.3f\n",
                    position->x, position->y, position->z);

        return true;
    }

    return false;
}


/*
 * Get the current position of the machine. Don't care where
 * it should be or which instrument we use.
 *
 * Formerly: void rdrp (struct TSRP *tsrp)
 */
void
Machine::GetCurrentPos (Double3d *pos)
{
    OpCodeToLink (RDRP);
    LongToLink (X_AXIS);
    ByteFromLink ();
    pos->x = DoubleFromLink ();
    OpCodeToLink (RDRP);
    LongToLink (Y_AXIS);
    ByteFromLink ();
    pos->y = DoubleFromLink ();
    OpCodeToLink (RDRP);
    LongToLink (Z_AXIS);
    ByteFromLink ();
    pos->z = DoubleFromLink ();
}


/*
 * Get the by a latch measured position.
 *
 * Formerly rdlp (struct TSRP *tsrp);
 */
void
```

```
Machine::GetLatchedPos (Double3d *pos)
{
    OpCodeToLink (RDLP);
    LongToLink (X_AXIS);
    ByteFromLink ();
    pos->x = DoubleFromLink ();
    OpCodeToLink (RDLP);
    LongToLink (Y_AXIS);
    ByteFromLink ();
    pos->y = DoubleFromLink ();
    OpCodeToLink (RDLP);
    LongToLink (Z_AXIS);
    ByteFromLink ();
    pos->z = DoubleFromLink ();
}



/*
 * Reset the axes: Stop them, Open the loops etc.
 *
 * Formerly ra (struct AS *as)
 */
void
Machine::ResetAxes ()
{
    if (debug) printf ("Axes reset\n");
    OpCodeToLink (RA);
    SendAxTab ();
    loopsClosed = false;
    running = false;
}


/*
 * Close the position hold loops.
 *
 * Formerly void cl(struct AS *as)
 */
void
Machine::CloseLoops ()
{
    if (loopsClosed) return;

    if (debug) printf ("Closing loops\n");
    OpCodeToLink (CL);
    SendAxTab ();
    usleep (RELAYTIME);
    loopsClosed = true;
}


/*
 * Open the position hold loops.
 *
 * Formerly void ol(struct AS *as)
 */
void
Machine::OpenLoops ()
```

```
{
    if (! loopsClosed) return;

    if (debug) printf ("Opening loops\n");
    OpCodeToLink (OL);
    SendAxTab ();
    usleep (RELAYTIME);
    running = false;
    loopsClosed = false;
}


/*
 * Set the current position to the root of the machine
 * coordinate system.
 *
 * Formerly: void shp(struct TSRP *tsrp)
 */
void
Machine::SetHomePosition ()
{
    OpCodeToLink (SHP);
    LongToLink (X_AXIS);
    DoubleToLink (0);
    OpCodeToLink (SHP);
    LongToLink (Y_AXIS);
    DoubleToLink (0);
    OpCodeToLink (SHP);
    LongToLink (Z_AXIS);
    DoubleToLink (0);
}


/*
 * Set the jog velocities, means the top moving speeds
 * while moving a normal profile.
 *
 * Formerly: void wrjvl(struct TSRP *tsrp)
 */
void
Machine::SetJogVelocities (Double3d *speed)
{
    if (debug) printf ("Setting JOG Velocities: %.3f %.3f %.3f\n",
                       speed->x, speed->y, speed->z);
    OpCodeToLink (WRJVL);
    LongToLink (X_AXIS);
    DoubleToLink (speed->x);
    OpCodeToLink (WRJVL);
    LongToLink (Y_AXIS);
    DoubleToLink (speed->y);
    OpCodeToLink (WRJVL);
    LongToLink (Z_AXIS);
    DoubleToLink (speed->z);
}


/*
 * Set the jog target velocities. This is the velocity AFTER
 * the end of the moving profile and is usually zero.
```

```
 *
 * Formerly: void wrjtvl(struct TSRP *tsrp)
 */
void
Machine::SetJogTargetVelocities (Double3d *speed)
{
    if (debug) printf ("Setting JOG TARGET Velocities: %.3f %.3f %.3f\n",
                       speed->x, speed->y, speed->z);
    OpCodeToLink (WRJTVL);
    LongToLink (X_AXIS);
    DoubleToLink (speed->x);
    OpCodeToLink (WRJTVL);
    LongToLink (Y_AXIS);
    DoubleToLink (speed->y);
    OpCodeToLink (WRJTVL);
    LongToLink (Z_AXIS);
    DoubleToLink (speed->z);
}


/*
 * Set the home velocities (= top speed while moving to home position).
 *
 * Formerly: void wrhvl(struct TSRP *tsrp)
 */
void
Machine::SetHomeVelocities (Double3d *speed)
{
    if (debug) printf ("Setting HOME Velocities: %.3f %.3f %.3f\n",
                       speed->x, speed->y, speed->z);
    OpCodeToLink (WRHVL);
    LongToLink (X_AXIS);
    DoubleToLink (speed->x);
    OpCodeToLink (WRHVL);
    LongToLink (Y_AXIS);
    DoubleToLink (speed->y);
    OpCodeToLink (WRHVL);
    LongToLink (Z_AXIS);
    DoubleToLink (speed->z);
}


/*
 * Move to a absolute position without interpolation.
 *
 * Formerly: void ja(struct AS *as, struct TSRP *tsrp)
 */
void
Machine::JogAbsolute (Double3d *target)
{
    if (debug) printf ("Jogging absolute: %.3f %.3f %.3f\n",
                       target->x, target->y, target->z);
    OpCodeToLink(JA);
    SendAxTab ();
    DoubleToLink (target->x);
    DoubleToLink (target->y);
    DoubleToLink (target->z);
}
```

```
/*
 * Move a relative distance without interpolation.
 *
 * Formerly: void jr(struct AS *as, struct TSRP *tsrp)
 */
void
Machine::JogRelative (Double3d *target)
{
    if (debug) printf ("Jogging relative: %.3f %.3f %.3f\n",
                       target->x, target->y, target->z);
    OpCodeToLink (JR);
    SendAxTab ();
    DoubleToLink (target->x);
    DoubleToLink (target->y);
    DoubleToLink (target->z);
}


/*
 * Wait for the end of the profile on the specified axis.
 * Mostly for SearchReference ()
 */
void
Machine::WaitProfileEnd ()
{
    do {
        usleep (200000);
        ReadAxesStatus ();
        if (transdebug) DebugStatusReport (false);
    } while (! (xAxst.bit.pe & yAxst.bit.pe & zAxst.bit.pe));
}


/*
 * Go to the index.
 *
 * Formerly void jhi(struct AS *as, struct TSRP *tsrp)
 */
void
Machine::GoIndex (Double3d *target)
{
    OpCodeToLink (JHI);
    SendAxTab ();
    DoubleToLink (target->x);
    DoubleToLink (target->y);
    DoubleToLink (target->z);
}


/*
 * Find the home position, positive direction.
 * Axes must be prepared, no Error checking is done.
 *
 * Formerly jhl(struct AS *as)
 */
void
Machine::GoHomePos (long axis)
{
```

```
    OpCodeToLink (JHR);
    LongToLink (1);
    LongToLink (axis);
}


/*
 * Same in negative direction.
 *
 * Formerly jhr(struct AS *as)
 */
void
Machine::GoHomeNeg (long axis)
{
    OpCodeToLink (JHL);
    LongToLink (1);
    LongToLink (axis);
}


/*
 * Set one of the ASM 2003 output bits for all axes.
 * Formerly wrasmob (long an, long bitnr, long value)
 */
void
Machine::SetAsmOutput (long bitnr, long value)
{
    OpCodeToLink (WRASMOB);
    LongToLink (X_AXIS);
    LongToLink (bitnr);
    LongToLink (value);
    OpCodeToLink (WRASMOB);
    LongToLink (Y_AXIS);
    LongToLink (bitnr);
    LongToLink (value);
    OpCodeToLink (WRASMOB);
    LongToLink (Z_AXIS);
    LongToLink (bitnr);
    LongToLink (value);
}


/*
 * Read the status of all axes.
 *
 * Formerly rdaxst(struct TSRP *tsrp)
 */
void
Machine::ReadAxesStatus ()
{
    OpCodeToLink (RDAXST);
    LongToLink (X_AXIS);
    ByteFromLink ();
    xAxst.status_word = LongFromLink ();
    OpCodeToLink (RDAXST);
    LongToLink (Y_AXIS);
    ByteFromLink ();
    yAxst.status_word = LongFromLink ();
    OpCodeToLink (RDAXST);
```

```
    LongToLink (Z_AXIS);
    ByteFromLink ();
    zAxst.status_word = LongFromLink ();
}


/*
 * Read the ASM 2003 input bits.
 *
 * Formerly rdasmi(), rdasmib()
 */
void
Machine::ReadAsmInput ()
{
    OpCodeToLink (RDASMI);
    LongToLink (X_AXIS);
    ByteFromLink ();
    xAsmi = LongFromLink ();
    OpCodeToLink (RDASMI);
    LongToLink (Y_AXIS);
    ByteFromLink ();
    yAsmi = LongFromLink ();
    OpCodeToLink (RDASMI);
    LongToLink (Z_AXIS);
    ByteFromLink ();
    zAsmi = LongFromLink ();
}


/*
 * Send an opcode to the Transputer.
 */
inline void
Machine::OpCodeToLink (char opcode)
{
    WriteLink (&opcode, sizeof (char));
}


/*
 * Send the axes table to the Transputer.
 */
inline void
Machine::SendAxTab ()
{
    long axTab[4] = {3L, 0L, 1L, 2L};

    WriteLink ((char *)axTab, sizeof (axTab));
}


/*
 * Send a long to the Transputer.
 * (for the Transputer this is the standard integer)
 */
inline void
Machine::LongToLink (long value)
{
    WriteLink ((char *)&value, sizeof (long));
```

```
}


/*
 * Send a double to the Transputer.
 */
inline void
Machine::DoubleToLink (double value)
{
    WriteLink ((char *)&value, sizeof (double));
}


/*
 * Get a long from Transputer.
 */
inline long
Machine::LongFromLink ()
{
    long value;

    ReadLink ((char *)&value, sizeof (long));
    return value;
}


/*
 * Get a double from Transputer.
 */
inline double
Machine::DoubleFromLink ()
{
    double value;

    ReadLink ((char *)&value, sizeof (double));
    return value;
}


/*
 * Get a byte from Transputer.
 */
inline char
Machine::ByteFromLink ()
{
    char byte;

    ReadLink (&byte, 1);
    return byte;
}


/*
 * Try to reboot the MCU-6. Do not handle errors, the calling code
 * will remember this when checking for a valid system after rebooting.
 * Templates for this routine see mcuboot.c.
 */
void
Machine::ReBoot ()
```

```
{
    time_t start;
    long size, count, *statusP;
    char buffer[100];

    if (debug) printf ("Rebooting MCU_6.\n");

    ResetLink ();

    if (WriteBootFile (BOOTFILENAME)) return;

    /*
     * Get {a,the} request from the Transputer, but allow a
     * relative big time limit.
     */
    start = time (NULL);
    while (! (count = ReadLink (buffer, 2)))
        if (time (NULL) > start + 60) break;
    if (count != 2) {
        WarningMsg ("Could not get request after writing bootfile");
        return;
    }

    /*
     * Calculate the size of the request.
     */
    size = ((unsigned char)buffer[0]) + ((unsigned char)buffer[1]) * 256;
    if (size < 6 || size > 98) {
        WarningMsg ("Unexpected request size");
        return;
    }

    /*
     * Read the rest of the request, overwrite the size bytes.
     */
    count = ReadLink (buffer, size);
    if (count != size) {
        WarningMsg ("Could not read complete request");
        return;
    }

    /*
     * Check the contents of the request. mcuboot.c checks here
     * more, but trashes the results anyway.
     */
    statusP = (long *)buffer;
    if (*statusP != 35) /* 35 = SP_EXIT */
        WarningMsg ("Request status not SP_EXIT");

    /*
     * Send Acknowledge to TPU.
     */
    buffer[0] = 1; /* Length LSB. */
    buffer[1] = 0; /* Length MSB. */
    buffer[2] = 0; /* SP_SUCCESS, one byte message. */
    WriteLink (buffer, 3);
}
```

```
/*
 * Write the boot file (usually rwtos.btl) to the link.
 * Usually used only once per session.
 */
int
Machine::WriteBootFile (char *filename)
{
    char  fileBuffer[128];
    long  length;
    FILE  *fp;

    /*
     * Open file for binary reading.
     */
    fp = fopen (filename, "rb");
    if (! fp) {
        char str[100];

        sprintf (str, "Could not open file <%s>\n", filename);
        WarningMsg (str);
        return 7;
    }

    /*
     * Transmit data until file end. Data must be written in exact
     * filesize.
     */
    while ((length = fread (fileBuffer, 1, sizeof (fileBuffer), fp)))
        if (WriteLink (fileBuffer, length) !=  length) {
            WarningMsg ("Transferred file incomplete");
            fclose (fp);
            return 1;
        }

    fclose (fp);
    return 0; /* success */

}


/*
 * Write the system file (usually system.dat) to the link.
 * Usually used only once per session. Formerly txbf().
 * The changes to WriteBootFile () are to much to do it in one routine.
 */
int
Machine::WriteSystemFile (char *filename)
{
    char  result, fileBuffer[128];
    long  length;
    FILE  *fp;

    /*
     * Open file for binary reading.
     */
    fp = fopen (filename, "rb");
    if (! fp) {
        char str[100];
```

```
        sprintf (str, "Could not open file <%s>\n", filename);
        WarningMsg (str);
        return 7;
    }

    /*
     * Examine file lenght.
     */
    fseek (fp, 0L, SEEK_END);
    length = ftell(fp);
    fseek (fp, 0L, SEEK_SET);

    /*
     * Copy the header to the Link.
     */
    fread (fileBuffer, 1, sizeof (fileBuffer), fp);

    OpCodeToLink (TXBF);
    LongToLink (length);
    WriteLink (fileBuffer, sizeof (fileBuffer));

    /*
     * Check if File-Type and File-Length o.k.
     */
    ByteFromLink ();
    result = ByteFromLink (); /* receive Status */
    if (result) {
        WarningMsg ("MCU 6 can't receive system file\n");
        fclose (fp);
        return (result);
    }

    /*
     * Transmit data until file end. Data must be written in multiples
     * of the filebuffer size (128 bytes) in opposite to WriteBootFile().
     */
    while (fread (fileBuffer, 1, sizeof (fileBuffer), fp))
        if (WriteLink (fileBuffer, sizeof (fileBuffer)) !=
            sizeof (fileBuffer)) {
            WarningMsg ("Transferred file incomplete");
            fclose (fp);
            return 1;
        }

    fclose (fp);
    return 0; /* success */
}


/*
 * Reads the resource file where acclerations and velocities
 * might be configured. A line beginning with a '#' is a comment.
 * Lines must be smaller than 80 characters.
 */
void
Machine::ReadResourceFile ()
{
    Double3d vvv;
    FILE *fd;
```

```
short linenr = 0;
char line[90], command[30];

/*
 * Jog target velocity must be zero anyway.
 * Perform infinite movement by setting the target position
 * out of the physical range of the machine.
 */
vvv.SetAll (0);
SetJogTargetVelocities (&vvv);

speedLevel = 1.0;

/* Open file for reading. */
fd = fopen ("messdrc", "r");
if (! fd) {
    WarningMsg ("Could not open resource file <messdrc>\n");
    return;
}

while (fgets (line, 90, fd)) {
    linenr++;
    /*
     * Cut off comments.
     */
    for (unsigned long i=0; i<sizeof (line); i++) {
        if (line[i] == '#')
            line[i] = 0;
    }

    command[0] = 0;
    sscanf (line, "%s", command);
    if (command[0] == 0) {
        continue;
    } else if (! strcmp (command, "SpeedLevel")) {
        sscanf (&line[sizeof ("SpeedLevel")], "%lf", &speedLevel);
        printf (" Set speed level to: %.3f\n", speedLevel);
        continue;
    } else if (! strcmp (command, "TipReversed")) {
        sscanf (line, "%s %s", command, command);
        if (! strncmp ("true", command, 4)) {
            tipReversed = true;
            printf (" Set tipReversed to true\n");
        } else {
            tipReversed = false;
            printf (" Set tipReversed to false\n");
        }
        continue;
    } else if (! strcmp (command, "TipRadius")) {
        sscanf (line, "%s %lf", command, &tipRadius);
        printf (" Set tip radius to: %.3f\n", tipRadius);
        continue;
    } else {
        sprintf (line, "Unknown command in line %d: <%s> -- ignored",
                 linenr, command);
        WarningMsg (line);
    }
}
fclose (fd);
```

```
}


/*
 * Get a axis status report for debugging purposes.
 */
void
Machine::DebugStatusReport (bool header)
{
    char str[28];

    ReadAxesStatus ();
    ReadAsmInput ();
    if (header) {
        printf ("Axes Status:                                        ASM 2003
InputStatus:\n");
        printf ("toasm dnr lslh lsrh dhef cef pe  cl  ip  ui  lpsf  0  1  2
3  4  5  6  7  8\n");
    }

    printf (" %1d%1d%1d", xAxst.bit.toasm, yAxst.bit.toasm,
            zAxst.bit.toasm);
    printf ("  %1d%1d%1d", xAxst.bit.dnr, yAxst.bit.dnr, zAxst.bit.dnr);
    printf (" %1d%1d%1d ", xAxst.bit.lslh, yAxst.bit.lslh, zAxst.bit.lslh);
    printf (" %1d%1d%1d ", xAxst.bit.lsrh, yAxst.bit.lsrh, zAxst.bit.lsrh);
    printf (" %1d%1d%1d ", xAxst.bit.dhef, yAxst.bit.dhef, zAxst.bit.dhef);
    printf (" %1d%1d%1d ", xAxst.bit.cef, yAxst.bit.cef, zAxst.bit.cef);
    printf ("%1d%1d%1d", xAxst.bit.pe, yAxst.bit.pe, zAxst.bit.pe);
    printf (" %1d%1d%1d", xAxst.bit.cl, yAxst.bit.cl, zAxst.bit.cl);
    printf (" %1d%1d%1d", xAxst.bit.ip, yAxst.bit.ip, zAxst.bit.ip);
    printf (" %1d%1d%1d ", xAxst.bit.ui, yAxst.bit.ui, zAxst.bit.ui);
    printf ("%1d%1d%1d  ", xAxst.bit.lpsf, yAxst.bit.lpsf, zAxst.bit.lpsf);
    short pos = 0;
    for (int i=0; i<9; i++) {
        str[pos] = '0';
        str[pos] += (xAsmi >> i) & 1;
        pos++;
        str[pos] = '0';
        str[pos] += (yAsmi >> i) & 1;
        pos++;
        str[pos] = '0';
        str[pos] += (zAsmi >> i) & 1;
        pos++;
    }
    str[27] = 0;
    printf ("%s\n", str);
}


/*
 * Read and display the velocities (= top speeds for movement).
 */
void
Machine::SpeedReport ()
{
    printf ("Jog Velocities.: ");
    OpCodeToLink (RDJVL);
    LongToLink (X_AXIS);
    ByteFromLink ();
```

```
    printf (" %7.3f ", DoubleFromLink ());
    OpCodeToLink (RDJVL);
    LongToLink (Y_AXIS);
    ByteFromLink ();
    printf (" %7.3f ", DoubleFromLink ());
    OpCodeToLink (RDJVL);
    LongToLink (Z_AXIS);
    ByteFromLink ();
    printf (" %7.3f   ", DoubleFromLink ());

    printf ("Motor: ");
    OpCodeToLink (RDMCP);
    LongToLink (X_AXIS);
    ByteFromLink ();
    printf (" %5ld ", LongFromLink ());
    OpCodeToLink (RDMCP);
    LongToLink (Y_AXIS);
    ByteFromLink ();
    printf (" %5ld ", LongFromLink ());
    OpCodeToLink (RDMCP);
    LongToLink (Z_AXIS);
    ByteFromLink ();
printf (" %5ld\n", LongFromLink ());
}
```

7.2.12 messd/rwtos.btl

Dies ist eine Binärdatei und enthält das Betriebssystem für die MCU 6. Sie wird auf einer Diskette mit der Hardware geliefert.

7.2.13 messd/system.dat

Dies ist auch eine Binärdatei und enthält die Konfigurationsdaten der MCU 6. Sie wird beim speichern der Konfiguration mit dem Konfigurationsprogramm erstellt.

7.3 Der Quellcode von QUICKMESS

7.3.1 quickmess/Makefile

```
#
# project: Interaktive Messsoftware
#
# program: quickmess
#
# file: Makefile
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the OS independant part
#


.SUFFIXES: .o .c .cc .h

include Makefile.$(UNAME)
```

```
HEADERS = myforms.h messmaconn.h ../improtocol.h
OBJECTS = quickmess.o myforms.o callbacks.o messmaconn.o error.o

CFLAGS += -DVERSION=\"1.0\"


quickmess: $(OBJECTS)
    $(CC) $(OBJECTS) $(LIBS) $(LDFLAGS) -o quickmess

error.o: ../libimess/error.h ../libimess/error.cc
    $(CXX) $(CFLAGS) $(INCLUDES) -c ../libimess/error.cc


.c.o: $< $(HEADERS)
    mv $*.c $*.cc
    $(CXX) $(CFLAGS) $(INCLUDES) -c $*.cc

.cc.o: $< $(HEADERS)
    $(CXX) $(CFLAGS) $(INCLUDES) -c $<



clean:
    rm -f *.o core quickmess
```

7.3.2 quickmess/Makefile.AIX

```
#
# project: Interaktive Messsoftware
#
# program: quickmess
#
# file: Makefile.AIX
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the OS specific part for AIX
#


CC = gcc
CXX = gcc
CFLAGS = -D_BSD=44 -Wall -Wno-implicit
LDFLAGS = -s

INCLUDES = -I$(HOME)/$(UNAME)/include -I../libimess/
LIBS = -L$(HOME)/$(UNAME)/lib -lforms -lX11 -lm -lbsd
```

7.3.3 quickmess/Makefile.Linux

```
#
# project: Interaktive Messsoftware
#
```

```
# program: quickmess
#
# file: Makefile.Linux
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the OS specific part for Linux
#


CC = gcc
CXX = g++
CFLAGS = -Wall
LDFLAGS = -s

INCLUDES = -I../libimess
LIBS = -lforms -lX11 -lm
```

7.3.4 quickmess/quickmess.cc

```
/*
 * project: Interaktive Messsoftware
 *
 * program: quickmess
 *
 * file: quickmess.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This is the main part of the quickmess program. This does not
 * mean that here is the most of the code, but here is the coordination done.
 */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>

#include "forms.h"
#include "myforms.h"
#include "error.h"
#include "messmaconn.h"


MessmaConn mmco;
FD_quickmess *fd_main;


int
main (int argc, char *argv[])
{
```

```
struct timeval lastround, thisround;
bool profiling = false;
char hostName[100];

/*
 * Parse the arguments.
 */
/* Set the default value. */
strcpy (hostName, "localhost");

/* Read each argument. */
for (int i=1; i<argc; i++) {
    if (! strncmp (argv[i], "-h", 2)) {
        printf ("\n  Usage: quickmess [name] [-h] [-d]\n\n");
        printf ("  name   Name of the host the server is running on.\n");
        printf ("         default: localhost\n");
        printf ("  -h     Print this help.\n");
        printf ("  -v     Print Version info\n");
        printf ("  -d     Turn debugging messages on\n");
        printf ("  -p     Turn profiling information on\n\n");
        exit (0);
    } else if (! strcmp (argv[i], "-d")) {
        MessmaConn::debug = true;
    } else if (! strcmp (argv[i], "-v")) {
        printf ("\n  Project: Interaktive Messsoftware\n");
        printf ("  Program: QUICKMESS, Version %s\n\n", VERSION);
        exit (0);
    } else if (! strcmp (argv[i], "-p")) {
        profiling = true;
    } else {
        strncpy (hostName, argv[i], 99);
    }
}

/*
 * Initialize the user interface, the fl_check_forms() is needed
 * to be sure it's there if we want to access it in the main loop.
 */
fl_initialize (argv[0], "quickmess", 0, 0, &argc, argv);
fd_main = create_form_quickmess ();
fl_set_bitmap_file (fd_main->pict, "pict-mb.xbm");
fl_check_forms ();
fl_check_forms ();
fl_check_forms ();
fl_check_forms ();

/*
 * Get the server. Any failure will abort the program.
 */
mmco.GetServer (hostName);

/*
 * Build the user Interface: This is just one window
 * composed with the x-forms designer. The output of this designer
 * will do all the work.
 */
fl_show_form (fd_main->quickmess, FL_PLACE_CENTER,
              FL_FULLBORDER, "quickmess");
```

```
    /*
     * MAIN LOOP
     */
    for (;;) {
        if (profiling) {
            gettimeofday (&thisround, NULL);
            if (lastround.tv_usec > thisround.tv_usec)
                lastround.tv_usec -= 1000000L;
            printf ("%ld ms\n", thisround.tv_usec - lastround.tv_usec);
            lastround = thisround;
        }

        /*
         * Check the display. This will also call the Callbacks, if needed.
         */
        fl_check_forms ();

        /*
         * Check for incoming messages. The required work is done there, too.
         */
        mmco.IdleProc ();
    }

    /* Never reached. */
    return 0;
}
```

7.3.5 quickmess/myforms.fd

Diese Datei ist eine Binärdatei und kann vom FORM DESIGNER gelesen werden. Sollte sie je rekonstruiert werden müssen, kann man mit dem FORM DESIGNER eine neue Datei anlegen und dabei das Aussehen an das Bild im Kapitel mit der Bedienung anlehnen. Die notwendigen Callbacks kann man aus der Datei `callbacks.cc` im Vergleich mit der Bedienungsanleitung erhalten.

7.3.6 quickmess/myforms.h

Diese Datei wird vom FORM DESIGNER automatisch beim bauen des Programms aus der Datei `myforms.fd` erzeugt und sollte nicht editiert werden. Ein Abdruck scheint daher nicht sinnvoll.

7.3.7 quickmess/myforms.cc

Auch diese Datei wird automatisch beim bauen des Programms erzeugt. Der FORM DESIGNER produziert aus der Datei `myforms.fd` zunächst `myforms.c`, diese Datei wird dann vom Makefile nach `myforms.cc` umbenannt um von C++ Kompiler bearbeitet werden zu können. Auch sie sollte nicht editiert werden.

7.3.8 quickmess/pict-mb.xbm

Diese Datei enthält das Maschinenbau Logo, das im QUICKMESS Fenster zu sehen ist, im X–Bitmap format. Die Datei trägt nicht zur Funktion des Programms bei und kann mit einem Bildbearbeitungsprogramm editiert oder erstellt werden.

7.3.9 quickmess/callbacks.cc

```
/*
 * project: Interaktive Messsoftware
```

```
 *
 * program: quickmess
 *
 * file: callbacks.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This file holds all the callbacks used in the Window. Each
 * click or valid keyboard interaction in the window triggers
 * one of these callbacks. Type and Name of the callbacks are
 * defined when the window form is designed (fdesign program).
 *
 * The callbacks just pass the needed information to a other object.
 * Any calculations are avoided here as far as possible.
 */


#include "forms.h"
#include "myforms.h"
#include "lib3d.h"
#include "messmaconn.h"

extern MessmaConn mmco;


/*
 * Callback for the Quit Button. Just exit.
 */
void
Myexit (FL_OBJECT *, long)
{
    exit (0);
}


/*
 * Common callback for all the movement Buttons. 'which' tells
 * which Button was pressed. Also called when a Button was released.
 */
void
StartStop (FL_OBJECT *button, long which)
{
    int mouse;
    double speed;
    Double3d doSpeed;

    /*
     * Check wether the Callback was triggerd by a Button
     * release. If so, stop the machine.
     */
    if (! fl_get_button (button)) {
        /* Button was released */
        mmco.StopMove ();
        return;
    }

    /*
```

```
     * Check which Mouse Button was used to press the Button.
     * The left Button sets the speed to 2 mm/s, the middle to 20 mm/s
     * and the right one to 100 mm/s (= rapid = maximum allowed speed).
     */
    mouse = fl_get_button_numb (button);
    if (mouse == 1) {
        speed = 2;
    } else if (mouse == 2) {
        speed = 20;
    } else if (mouse == 3) {
        speed = 100;
    } else {
        speed = 0;
    }

    /*
     * Set the direction of the speed depending on 'which'.
     */
    doSpeed.SetAll (0);
    switch (which) {
      case 1:
        doSpeed.z = speed;
        break;
      case 2:
        doSpeed.y = -speed;
        break;
      case 3:
        doSpeed.x = -speed;
        break;
      case 4:
        doSpeed.z = -speed;
        break;
      case 5:
        doSpeed.y = speed;
        break;
      case 6:
        doSpeed.x = speed;
        break;
    }

    /*
     * Finally, do the request.
     */
    mmco.StartMove (&doSpeed);
}


/*
 * The 'measure here' Button was pressed. Releasing the Button
 * doesn't trigger a callback.
 */
void
MeasureHere (FL_OBJECT *, long)
{
    /* This will send a request,
       the incoming response will do the rest. */
    mmco.MeasurePos ();
}
```

```
/*
 * Same as above for the 'mark here' Button.
 */
void
MarkHere (FL_OBJECT *, long)
{
    mmco.MarkPos ();
}


/*
 * One of the two Distance Buttons were pressed or released.
 * mmco.StartMove() will figure out in which direction to move,
 * because there is the marked position known.
 * The rest is similar to the StartStop() routine.
 */
void
DistMark (FL_OBJECT *button, long which)
{
    int mouse;
    double speed;

    if (! fl_get_button (button)) {
        /* Button was released */
        mmco.StopMove ();
        return;
    }

    mouse = fl_get_button_numb (button);
    if (mouse == 1) {
        speed = 2;
    } else if (mouse == 2) {
        speed = 20;
    } else if (mouse == 3) {
        speed = 100;
    } else {
        speed = 0;
    }

    switch (which) {
      case 1:
        mmco.StartMove (speed);
        break;
      case 2:
        mmco.StartMove (-speed);
        break;
    }
}
```

7.3.10 quickmess/messmaconn.h

```
/*
 * project: Interaktive Messsoftware
 *
 * program: quickmess
 *
```

```
 * file: messmaconn.h
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

#ifndef MESSMACONN_H
#define MESSMACONN_H


#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>

#ifdef _AIX
#include <sys/select.h>
#endif

#ifdef linux
#include <sys/time.h>
#include <sys/socketvar.h>
#endif

#include "improtocol.h"
#include "lib3d.h"
#include "forms.h"
#include "myforms.h"


class MessmaConn
{
  public:
    MessmaConn ();
    ~MessmaConn ();

    void GetServer (const char *serverName);

    /* Begin the axles moving with three explicit speeds for each axis. */
    void StartMove (Double3d *speed);
    /* Begin the axles moving with the direction beginning from the last
       marked Position. */
    void StartMove (double speed);
    void StopMove ();

    void IdleProc ();

    void MeasurePos ();
    void MarkPos ();

    static bool debug;

  private:
    Double3d currentPos;
    Double3d markedPos;
    Double3d measuredPos;

    struct sockaddr_in server;
    struct hostent *hostInfo;
    int sock, count;
    int fdWidth;
```

```
    char *serverName;

    bool connected;

    void Send (const char *);
};


#endif /* MESSMACONN */
```

7.3.11 quickmess/messmaconn.cc

```
/*
 * project: Interaktive Messsoftware
 *
 * program: quickmess
 *
 * file: messmaconn.cc
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * This class does all communication with the daemon handling
 * the real machine. Some machine related data, like e.g. the
 * marked position is also stored here.
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#ifdef _AIX
#include <sys/timers.h>
#endif
/* Markus: This covers a link failure.
           Curious: messd uses same code but do not need this define. */
#ifdef linux
#define htons(x) __htons(x)
#endif

#include "forms.h"
#include "myforms.h"
#include "error.h"
#include "messmaconn.h"


extern FD_quickmess *fd_main;

bool MessmaConn::debug = false;
```

```
/*
 * The constructor: initialize some variables.
 */
MessmaConn::MessmaConn ()
{
    /*
     * Get the maximum number of open file descriptors,
     * but do not overdrive.
     */
    fdWidth = getdtablesize ();
    if (fdWidth > 20) fdWidth = 20;
}


/*
 * The destructor: nothing to do.
 */
MessmaConn::~MessmaConn ()
{
}


/*
 * Open the connection to the server on SOCKET_PORT. This is the
 * standard handling of BSD sockets and the same as in the imess
 * catia load module.
 * Currently the program aborts if it's unable to open a connection.
 * It might be useful, but not easy to implement to open the connection
 * later in the program.
 */
void
MessmaConn::GetServer (const char *serverName)
{
    sock = socket (AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
        ErrorMsg ("Could not open socket");

    hostInfo = gethostbyname (serverName);
    if (hostInfo == NULL) {
        char msg[50];
        sprintf (msg, "Unknown host: %s", serverName);
        ErrorMsg (msg);
    }

    server.sin_family = hostInfo->h_addrtype;
    memcpy ((char *)&server.sin_addr, hostInfo->h_addr, hostInfo->h_length);
    server.sin_port = htons ((unsigned short)SOCKET_PORT);
#ifndef linux
    server.sin_len = sizeof (server);
#endif

    if (connect (sock, (sockaddr *)&server, sizeof (server)) < 0) {
        char msg[50];
        sprintf (msg, "Could not connect on port %d", SOCKET_PORT);
        ErrorMsg (msg);
    }
}
```

```
/*
 * Send a message to the server. The server will recognize the
 * end of the message in the newline.
 */
void
MessmaConn::Send (const char *message)
{
    char newline = '\n';

    if (debug)
        printf ("Sending: <%s> (%d Bytes)\n", message, strlen (message));
    write (sock, message, strlen (message));
    write (sock, &newline, 1);
}


/*
 * This routine is usually called once in the main loop. It looks
 * if there is something to read from the connection and delays
 * for a short time if there is nothing to do. If there WAS something
 * to read, the message is handled here as well. The handling of the
 * connection is here similar to the catia load, too.
 */
void
MessmaConn::IdleProc ()
{
    struct timeval sleeptime;
    fd_set readySet;
    int len, ret;
    char buffer[MAX_MSG_LEN], buf[20];

    /*
     * Prepare a new empty file desciptor set, it's
     * discarded at return.
     */
    FD_ZERO (&readySet);
    FD_SET (sock, &readySet);
    sleeptime.tv_sec = 0;
    sleeptime.tv_usec = 0;

    /*
     * Check if there is something to read but do not wait.
     * If there was nothing to read, just return.
     */
    select (fdWidth, &readySet, 0, 0, &sleeptime);
    if (FD_ISSET (sock, &readySet)) {
        /* Read the line character by character to find
           the end of the line. There might be more than one
           line in the queue. */
        if (debug) printf ("Reading: ");
        for (len=0; len<MAX_MSG_LEN; len++) {
            ret = read (sock, &buffer[len], 1);
            if (ret == 1) {
                if (debug) {
                    putchar (buffer[len]);
                    fflush (stdout);
                }
                if (buffer[len] == '\n') {
                    buffer[len] = 0;
```

```
                    break;
                }
            } else if (ret == 0 && len == 0) {
                shutdown (sock, 2);
                close (sock);
                connected = false;
                /* Markus: we should not abort here, but try to
                   reconnect to the server when he's started again. */
                ErrorMsg ("Connection closed by server");
                return;
            } /* else error while reading or truncated message.
                 - ignore it. */
        }
    } else
        return;

    /*
     * We reach this place only if there is a
     * message, we should handle.
     */
    if (! strncmp (buffer, MSG_CURR_COORD,
                   sizeof (MSG_CURR_COORD) - 1)) {
        /* We have a new current position. */
        sscanf (buffer, "%s %lf %lf %lf", buf, &currentPos.x,
                &currentPos.y, &currentPos.z);

        /* write the position to the appropriate text fields. */
        sprintf (buf, "X:   %.3f", currentPos.x);
        fl_set_object_label (fd_main->cpx, buf);
        sprintf (buf, "Y:   %.3f", currentPos.y);
        fl_set_object_label (fd_main->cpy, buf);
        sprintf (buf, "Z:   %.3f", currentPos.z);
        fl_set_object_label (fd_main->cpz, buf);
    } else if (! strncmp (buffer, MSG_MEASURED_PT,
                          sizeof (MSG_MEASURED_PT) - 1)) {
        /* We have a new measured position. */
        sscanf (buffer, "%s %lf %lf %lf", buf, &measuredPos.x,
                &measuredPos.y, &measuredPos.z);

        /* write the position to the appropriate text fields. */
        sprintf (buf, "X:   %.3f", measuredPos.x);
        fl_set_object_label (fd_main->lmpx, buf);
        sprintf (buf, "Y:   %.3f", measuredPos.y);
        fl_set_object_label (fd_main->lmpy, buf);
        sprintf (buf, "Z:   %.3f", measuredPos.z);
        fl_set_object_label (fd_main->lmpz, buf);
    } /* else unknown message - ignore it. */

    /* Support for more Messages comes in here */
}


/*
 * Send a request to the daemon for moving the axes with
 * three explicit speeds for each axis.
 */
void
MessmaConn::StartMove (Double3d *speed)
{
```

```
    char str[MAX_MSG_LEN];

    /* Send the request. */
    sprintf (str, "%s %.3f %.3f %.3f", MSG_START_MOVE,
             speed->x, speed->y, speed->z);
    Send (str);
}


/*
 * Begin the axles moving with the direction beginning from the last
 * marked Position.
 */
void
MessmaConn::StartMove (double speed)
{
    char str[MAX_MSG_LEN];
    Double3d l;

    /* Calulate the moving direction. */
    /* Markus: It whould be better to send first a position
               request to the machine. The current position we
               remember here might be up to 1 second old. */
    l = markedPos - currentPos;
    l = l / l.Len () * speed;

    /* Send the request. */
    sprintf (str, "%s %.3f %.3f %.3f", MSG_START_MOVE, l.x, l.y, l.z);
    Send (str);
}


/*
 * Send a request to stop to the machine.
 */
void
MessmaConn::StopMove ()
{
    Send (MSG_STOP_MOVE);
}


/*
 * Send a request to measure the current position.
 */
void
MessmaConn::MeasurePos ()
{
    /*
     * Just sending the request is sufficient, because the incoming
     * response will do all the rest of the work.
     */
    Send (MSG_FORCE_MEAS);
}


/*
 * mark the position here, this means remember the position
 * for later use.
```

```
 */
void
MessmaConn::MarkPos ()
{
    char buf[20];

    /* Markus: Here it whould be better first to request
               the current position, too. */
    markedPos = currentPos;

    /* Update the Display, too. */
    sprintf (buf, "X:   %.3f", markedPos.x);
    fl_set_object_label (fd_main->mpx, buf);
    sprintf (buf, "Y:   %.3f", markedPos.y);
    fl_set_object_label (fd_main->mpy, buf);
    sprintf (buf, "Z:   %.3f", markedPos.z);
    fl_set_object_label (fd_main->mpz, buf);
 }
```

7.4 Der Quellcode des CATIA Moduls

7.4.1 catia/iua/IMESS.iuaproc

```
* Interaktive Messsoftware 1.0
*
* project: Interaktive Messsoftware
*
* program: catia modul
*
* file: IMESS.iuaproc
*
* Copyright: Markus Hitter 1996
*

*-------------------------------------------------------------------------
                              INTERNAL
*-------------------------------------------------------------------------
* Number of points to be measured.
  INTEGER REPTS

* String for keyboard input
  CHAR*72 ANSWER

*-------------------------------------------------------------------------
                              EXTERNAL
*-------------------------------------------------------------------------
* For coordinate receiving from im_do() and showing
  REAL X FORMAT 4/3
  REAL Y FORMAT 4/3
  REAL Z FORMAT 4/3

*-------------------------------------------------------------------------
                              PROC
*-------------------------------------------------------------------------

* Prepare for failure.
  LOADEXIT imess ENTRY im_close
```

```
  MSGCNTL '(c) Markus Hitter, FH Trier 1996'

* Ask the user how many points we want
  LABEL START
  MSG 'YES:SINGLE PT // KEY NUMBER PTS' ANSWER, KEY, YES, NO
  IF (KODE EQ NO) EXIT
  BLOCKIF (KODE EQ YES) THEN
    LET REPTS = 1
  ELSE
    BLOCKIF (ANSWER EQ ' ') THEN
      MSGCNTL 'INVALID INPUT'
      BEEP
      BRANCH START
    ENDIF
    LET REPTS = CONV (ANSWER)
    BLOCKIF ((IRET NE 1) OR (REPTS LT 0)) THEN
      MSGCNTL 'INVALID INPUT'
      BEEP
      BRANCH START
    ENDIF
  ENDIF
  IF (REPTS EQ 0) EXIT

* Open the connection (add a blank after the host name).
  LOAD imess ENTRY im_open 'mbpc18 '

  MSG 'IMESS WAITING FOR INCOMING POINTS: MOVE MACHINE'
  DO
    LOAD imess ENTRY im_do X, Y, Z
    BLOCKIF (IRET EQ 0) THEN
*     Nothing to do, just allow to refresh the display
      DISPLAY
    ELSE
      BLOCKIF (IRET EQ 1) THEN
*       Position has changed, show the new coordinates
        MSGCNTL 'X:' // CHCONV (X) //
        %       '  Y:' // CHCONV (Y) //
        %       '  Z:' // CHCONV (Z)
        DISPLAY
      ELSE
        BLOCKIF (IRET EQ 2) THEN
*         A point was measured an already created.
          BEEP
          DISPLAY
          LET REPTS = REPTS - 1
        ELSE
          BLOCKIF (IRET GT 2) THEN
*           Some sort of failure, but usually 'endexe' is
*           used in the load's code.
            MSGCNTL 'Failure ' // CHCONV (IRET)
            DUMP
            EXIT
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  WHILE (REPTS GT 0)
```

```
END
```

7.4.2 catia/load/Makefile

```
#
# project: Interaktive Messsoftware
#
# program: catia load modul
#
# file: Makefile
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the OS independant part
#


.SUFFIXES: .o .c .h

include Makefile.$(UNAME)


OBJECTS = imess.o
HEADERS =

CFLAGS += -DVERSION=\"1.0\"


# Allgemeine Vorgehensweise beim erzeugen von loads:
#
# Fuer load mit nur einem entry (name wie load):
# 'shared -c -p $(OBJECTS) -o LOADNAME'
#   -> erzeugt "shared.o" und "LOADNAME.ed".
#      Nur notwendig bei Aenderung des Namens oder des environments
# 'LOADNAME.ed'
#   -> erzeugt load
#
# Fuer loads mit mehreren entries:
# Namen der entries in file schreiben (z.B. "imessentries")
# 'shared -c -p $(OBJECTS) -e imessentries -o LOADNAME'
#   -> erzeugt "shared.o" und "LOADNAME.ed".
# 'LOADNAME.ed'
#   -> erzeugt load


all: IMESS

IMESS: $(OBJECTS) IMESS.ed
    IMESS.ed

IMESS.ed: imessentries
    shared -c -e imessentries -p $(OBJECTS) -o IMESS


.c.o: $< $(HEADERS)
    $(CXX) $(CFLAGS) $(INCLUDES) -c $<
```

```
.f.o:
    xlf -c $<


clean:
    rm -f core IMESS.ed IMESS.lk *.o
```

7.4.3 catia/load/Makefile.AIX

```
#
# project: Interaktive Messsoftware
#
# program: catia load modul
#
# file: Makefile
#
# Copyright: Markus Hitter, FH Trier 1996
#

#
# This file holds the OS specific part for AIX3 and AIX4
#


CC = gcc
CXX = gcc
CFLAGS = -D_BSD=44 -Wall -Wno-implicit
LDFLAGS = -s

INCLUDES = -I/u/mb4/mbpool/hitterm/diplomarbeit/libimess/
LIBS = -lm -lbsd
```

7.4.4 catia/load/imess.c

```
/*
 * project: Interaktive Messsoftware
 *
 * program: catia load modul
 *
 * file: imess.c
 *
 * Copyright: Markus Hitter, FH Trier 1996
 */

/*
 * These are the selfdefined CATIA loads for the Interaktive
 * Messsoftware project. The entries are called by IMESS.iuaproc.
 * Most of the work is done here, the IUA proc does only the direct
 * user interaction.
 * All loads used here are documented in the BookManager books:
 *  - CATIA Solutions API and
 *  - CATIA 3D Wireframe API
 * except where noticed.
 *
 * There are some standard variables needed when calling loads:
```

```
 *  - 'mnum' is the model number and must be initialized with 1,
 *    see BookManager: CATIA Solutions API: 1.1.4.1 Multi model concepts.
 *  - 'ier' is the output error code, it's not evaluated here.
 *    See BookManager: CATIA Solutions API: 1.2.2 Subroutine Documentation.
 *  - '*LAB' is the label address which should be activated in case of error,
 *    also not used here. Errors lead here only to non-behaviour, not to
 *    mis-behaviour.
 *    see BookManager: CATIA Solutions API: 1.2.2 Subroutine Documentation.
 *
 * Unfortunatly all Keyboard and Display handling must be done with
 * IUA commands (DISPLAY, SELECT ...). Writing a message or updating
 * the Display is impossible by using loads. Using GII whould not
 * change this situation. For this reason, the program has to switch
 * as often as possible between this load and the IUA program. In other
 * words, im_do() is called from a IUA in a (nearly) endless loop.
 */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netdb.h>
#include <netinet/in.h>

#include "improtocol.h"
#include "lib3d.h"

//#define DEBUG

typedef struct {
    long magic;
    struct sockaddr_in server;
    struct hostent *hostInfo;
    int sock, count;
    int fdWidth;
    char serverName[82];
    long toolWsp;
    long toolDitto;
    char message[70];
} imessData;


int im_open (const char *server);
int im_close ();
int im_do (double *x, double *y, double *z);

static void VisuMove (Double3d *newCoord);


/*
 * Global variables work only if automatic unloading is set to false!
 * To unload the load explicitly (e.g. after compilation), use
 * the '/unloadm imess' command.
 * Note that all entries first check for a valid magic number
 * to protect against misbehaviour in case of data loss.
```

```
 */
imessData data;
char* me = " IMESS: ";
long mnum = 1, ier;


/*
 * This entry opens the connection to the "messd" server. The
 * parameter must hold the Internet name of the machine on which
 * messd runs on. Any failure will stop the whole task by calling
 * endexe(). After successful connection the Measurement Tool visualisation
 * Ditto is searched. If there is no fitting Detail, the visualisation
 * is just left out but everything else will work fine.
 */
int
im_open (const char *server)
{
    short i;
    long nextWsp, wspType, lastWsp;
    long identLen;
    char wspIdent[18];

    printf ("Interaktive Messsoftware Version %s\n", VERSION);

    /* Set our security magic number. */
    data.magic = 020466;

    /*
     * Open the connection to messd.
     * Mostly the same as in the "quickmess" program.
     */

    /* This is a OS constant (the maximum number of
       open file descriptors). */
    data.fdWidth = getdtablesize ();

    /* Copy the Server name. Because the name is a FORTRAN
       alphanumeric constant, the calling procedure has to
       add a blank as the last character. */
    strncpy (data.serverName, server, 80);
    for (i=0; i<80; i++) {
        if (data.serverName[i] == ' ') {
            data.serverName[i] = 0;
            break;
        }
    }

#ifdef DEBUG
    printf ("im_open (%s)\n", data.serverName);
#endif

    data.sock = socket (AF_INET, SOCK_STREAM, 0);
    if (data.sock < 0) {
        data.sock = 0;
        endexe (" Could not open socket;", me);
    }

    data.hostInfo = gethostbyname (data.serverName);
    if (data.hostInfo == NULL) {
```

```
        char msg[50];
        sprintf (msg, " Unknown host: %s;", data.serverName);
        endexe (msg, me);
    }

    data.server.sin_family = data.hostInfo->h_addrtype;
    memcpy ((char *)&data.server.sin_addr, data.hostInfo->h_addr,
            data.hostInfo->h_length);
    data.server.sin_port = htons (SOCKET_PORT);
    data.server.sin_len = sizeof (data.server);

    if (connect (data.sock, (struct sockaddr *)&data.server,
                 sizeof (data.server)) < 0) {
        char msg[50];
        sprintf (msg, " No partner on port %d;", SOCKET_PORT);
        endexe (msg, me);
    }

    /*
     * OK, the connection is up.
     * Now find the Measure Tool in the CATIA model for visualisation.
     */
    /* Scan the workspaces to find the Tool (name = MEASURE TOOL).
       Begin with 0 = master workspace. */
    data.toolWsp = 0;
    do {
        giswsp (&mnum, &data.toolWsp, &nextWsp, &wspType, &lastWsp, &ier);
        if (wspType == 3 || wspType == 4) {
            /* It's a SPACE detail, check the name. */
            giride (&mnum, &data.toolWsp, &identLen, wspIdent, &ier);
            if (identLen >= 12 && ! strncmp (wspIdent, "MEASURE TOOL", 12)) {
                lastWsp = 0;
                break;
            }
        }
        data.toolWsp = nextWsp;
    } while (lastWsp != 1);

    /* If we don't find a tool we just leave out the visualisation. */
    if (lastWsp == 1)
        data.toolWsp = 0;

    data.toolDitto = 0;

    return 0;
}


/*
 * This entry cleans up as well as possible. Any error leaves
 * us in a hopeless situation and is ignored.
 */
int
im_close ()
{
#ifdef DEBUG
    printf ("im_close ()\n");
#endif
```

```
    if (data.magic != 020466)
        /* using endexe() here whould lead to a recursion. */
        return 0;

    /* If we have a tool Ditto, remove it. */
    if (data.toolWsp && data.toolDitto)
        gieras (&mnum, &data.toolDitto, &ier);

    shutdown (data.sock, 2);
    close (data.sock);

    return 0;
}


/*
 * This routine does all the work. It should be called as often
 * as possible from the IUA procedure. Depending on the return value
 * the IUA procedure should do additional things like Display updating.
 *
 * Table of return values:
 *
 * 0   ok but nothing to do for the IUA proc.
 *
 * 1   ok, the machine position has changed, position in x, y, z.
 *
 * 2   ok, a point is measured and created.
 *
 * 99  fatal failure, connection is closed.
 *     usually is called endexe() is such circumstances.
 *
 * Handling the Tool visualisation Ditto is also done here. The mechanism
 * leaves out visualisation updates when lots of communication traffic
 * is pending, in other words updates are only done when no message pends.
 */

int
im_do (double *x, double *y, double *z)
{
    /* For the communication. */
    struct timeval sleeptime;
    fd_set readySet;

    /* read buffers etc. */
    int len, ret;
    Double3d coord;
    char buffer[MAX_MSG_LEN], command[20];

    /* Tool visualisation handling. */
    static Double3d visuCoord;


    if (data.magic != 020466)
        endexe (" Data lost. Load reloaded?;", me);

    if (data.sock <= 0)
        endexe (" Not connected to server;", me);

#ifdef DEBUG
```

```
    printf (".");
    fflush (stdout);
#endif

    /* prepare a new empty file desciptor set. */
    FD_ZERO (&readySet);
    FD_SET (data.sock, &readySet);
    sleeptime.tv_sec = 0;
    sleeptime.tv_usec = 0;

    /* Check if there is something to read, wait otherwise. */
    select (data.fdWidth, &readySet, 0, 0, &sleeptime);
    if (FD_ISSET (data.sock, &readySet)) {
        /* Read the line character by character to find
           the end of the line. There might be more than one
           line in the queue. */
#ifdef DEBUG
        printf ("Reading: ");
#endif
        for (len=0; len<MAX_MSG_LEN; len++) {
            ret = read (data.sock, &buffer[len], 1);
            if (ret == 1) {
#ifdef DEBUG
                putchar (buffer[len]);
                fflush (stdout);
#endif
                if (buffer[len] == '\n') {
                    buffer[len] = 0;
                    break;
                }
            } else if (ret == 0 && len == 0) {
                shutdown (data.sock, 2);
                close (data.sock);
                endexe (" Connection closed by server;", me);
            } /* else error while reading or truncated message.
                 - ignore it. */
        }
    } else {
        /* Nothing to do. */
        VisuMove (&visuCoord);
        return 0;
    }

    /* React to the Messages */
    if (! strncmp (buffer, MSG_CURR_COORD, sizeof (MSG_CURR_COORD)-1)) {
        /*
         * The machine has moved, here are the new position coordinates.
         */
        sscanf (buffer, "%s %lf %lf %lf", command, x, y, z);
        visuCoord.x = *x;
        visuCoord.y = *y;
        visuCoord.z = *z;

        /* VisuMove() will be called when no messages are pending. */

        return 1;
    } else if (! strncmp (buffer, MSG_MEASURED_PT,
                          sizeof (MSG_MEASURED_PT) - 1)) {
        /*
```

```
         * A point was measured! Create the point.
         */
        long measpt;

        sscanf (buffer, "%s %lf %lf %lf", command,
                &coord.x, &coord.y, &coord.z);
        giwpt (&mnum, &coord, &measpt, &ier);

        return 2;
    }

    /* Support for more Messages comes in here */

    /* Ignore unknown messages. */

    return 0;
}


/*
 * This routine actually moves the Tip visualisation Ditto.
 * If there is not yet a Ditto, it's created.
 * It's not a entry callable by IUA.
 */
void
VisuMove (Double3d *newCoord)
{
    static Double3d oldCoord;
    static double transM[12];
    double scale;
    long identLen, transf;
    char ident[8];

    if (! data.toolWsp) return;

    /* If we have not yet a Ditto, create one and return. */
    if (! data.toolDitto) {
        /* Calculate the first transformation Matrix. */
        transM[0] = transM[4] = transM[8] = 1;
        transM[1] = transM[2] = transM[2] = 0;
        transM[5] = transM[6] = transM[7] = 0;
        transM[9] = newCoord->x;
        transM[10] = newCoord->y;
        transM[11] = newCoord->z;

        /* Create a new Ditto. */
        scale = 1;
        gcwdit (&mnum, &data.toolWsp, &scale, transM,
                &data.toolDitto, &ier);

        oldCoord = *newCoord;
        return;
    }

    /* Move the Ditto, if needed. */
    if (newCoord->x != oldCoord.x ||
        newCoord->y != oldCoord.y ||
        newCoord->z != oldCoord.z) {
        /* Calculate the new transformation Matrix. */
```

```
        transM[9] = newCoord->x - oldCoord.x;
        transM[10] = newCoord->y - oldCoord.y;
        transM[11] = newCoord->z - oldCoord.z;

        /* Create a space transformation. */
        identLen = 0;
        giwtra (&mnum, &identLen, ident, transM, &transf, &ier);

        /* Apply the transformation in replace mode. */
        gtsrp1 (&mnum, &data.toolDitto, &transf, &ier);
    }

    oldCoord = *newCoord;
    return;
}
```

7.4.5 catia/load/imessentries

Dies ist eine Hilfsdatei mit einem Verzeichnis der Entries im Load. Sie wird benötigt, um mehrere Entries in einem Load unterzubringen.

```
im_open
im_close
im_do
```

7.4.6 catia/load/imess.lk

Diese Datei enthält den Assembler Header des Loads und wird automatisch beim bauen desselben erzeugt. Er ändert sich nicht, solange sich Zahl und Art der Entries nicht ändern.

7.4.7 catia/load/imess.ed

Diese Datei enthält ein Hilfsscript zum linken des Loads. Auch dieses wird erforderlichenfalls beim bauen des Loads automatisch erzeugt und ändert sich nicht, solange Zahl und Namen der Entries gleich bleiben.

Kapitel 8

Anhang

8.1 Inhalt der beiliegenden Diskette

Bei der beiliegenden Diskette handelt es sich um eine UNIX Diskette, die mit dem `tar` Befehl bespielt wurde. `tar` ist auch für DOS und MacOS (z.B. `suntar`) frei verfügbar. Beim Auspacken erzeugt sie ein Verzeichnis `diplom`, in dem folgendes zu finden ist:

- Ein Verzeichnis `libimess` mit dem gemeinsamen Quellcode.
- Ein Verzeichnis `quickmess` mit QUICKMESS Quellcode und Programmen. `quickmess.AIX` und `quickmess.Linux` ist das Bedienprogramm jeweils in einer Version für RS6000/AIX und PC/Linux.
- Ein Verzeichnis `messd` mit dem MESSD Quellcode samt Programm (`messd`).
- Ein Verzeichnis `catia` mit Quellcode und Binärdatei (`IMESS`) für den CATIA Load und das IUA Programm (`IMESS.iuaproc`) in jeweils einem eigenen Unterverzeichnis.
- Ein Verzeichnis `frame` mit den FrameMaker Dokumenten dieser Diplomarbeit. Es ist FrameMaker oder FrameViewer Version 5 zum lesen dieser Dokumente notwendig. Eine Konvertierung nach Version 3 bzw. 4 war nicht ohne weiteres möglich. Bilder und Listings sind nur per Referenz mit dem Dokument verbunden und müssen sich daher im Verzeichnis `Bilder` bzw. `Source` befinden.